Singapore Management University

Institutional Knowledge at Singapore Management University

Dissertations and Theses Collection (Open Access)

Dissertations and Theses

8-2020

Novel deep learning methods combined with static analysis for source code processing

Duy Quoc Nghi BUI
Singapore Management University

Follow this and additional works at: https://ink.library.smu.edu.sg/etd_coll

Part of the OS and Networks Commons, Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation

BUI, Duy Quoc Nghi. Novel deep learning methods combined with static analysis for source code processing. (2020). 1-131.
Available at: https://ink.library.smu.edu.sg/etd_coll/306

This PhD Dissertation is brought to you for free and open access by the Dissertations and Theses at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Dissertations and Theses Collection (Open Access) by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

NOVEL DEEP LEARNING METHODS COMBINED WITH STATIC ANALYSIS FOR SOURCE CODE PROCESSING

BUI DUY QUOC NGHI

SINGAPORE MANAGEMENT UNIVERSITY

2020

Novel Deep Learning Methods Combined with Static Analysis for Source Code Processing

Bui Duy Quoc Nghi

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

Lingxiao Jiang (Supervisor/Chair)
Associate Professor
Singapore Management University

David Lo
Associate Professor
Singapore Management University

Jiang Jing
Associate Professor
Singapore Management University

Yijun Yu
Associate Professor
The Open University, UK

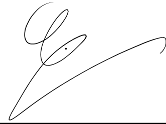
Singapore Management University

2020

Copyright (2020) Bui Duy Quoc Nghi

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in this thesis. This thesis has also not been submitted for any degree in any university previously.

Signed: 

Date: 19/08/2020

Abstract

It is desirable to combine machine learning and program analysis so that one can leverage the best of both to increase the performance of software analytics. On one side, machine learning can analyze the source code of thousands of well-written software projects that can uncover patterns that partially characterize software that is reliable, easy to read, and easy to maintain. On the other side, the program analysis can be used to define rigorous and unique rules that are only available in programming languages, which enrich the representation of source code and help the machine learning to capture the patterns better.

In this dissertation, we aim to present novel code modeling approaches to learn the source code better and demonstrate the usefulness of such approaches in various software engineering tasks. The methods developed for the aims to utilize the advantages of both deep learning techniques and static code analysis techniques.

The contributions in this dissertation are as follows:

1. **Hierarchical representation of source code**: a novel approach to model the source code as a hierarchical representation of different code levels, e.g., token level, statement level, function level, etc., to better capture the semantic information of source code from finer-grained to coarser-grained level for the API mapping task.

2. **Enhanced AST with semantic information**: a novel tree representation of source code, so-called the Dependency Tree, which is the enriched version of the Abstract Syntax Tree with def-use chain information. This is a

framework of *bilateral* neural networks (Bi-NN), an idea adapted from the area of neural machine translation and Siamese neural networks, aiming to generalize a previous study on cross-language program classification.

3. **Unsupervised API mapping**: a deep learning-based approach that can adapt the two domains with almost no parallel data, namely SAR. The underlying goal of prior API mapping techniques is essential to find a transformation that can align two different domains (in our context, the two vector spaces for APIs in two different languages). The key idea is to adapt the Generative Adversarial Network to align the vector spaces without the need for parallel data for the API mapping task.

4. **Interpretability for code modeling**: Towards the interpretability of the neural network for code classification, we propose to use the attention mechanism to quantify the importance of input elements based on their effects on the outputs of the network. The quantified attention scores and effects of the perturbed elements help to provide explanations on how the network classify programs.

5. **Capsule Network for AST Processing**: a novel tree-based capsule networks (TreeCaps) for processing program code . TreeCaps is a fusion between capsule networks with tree-based convolutional neural networks, to achieve learning accuracy higher than existing graph-based techniques while it is based only on trees. TreeCaps introduces novel *variable-to-static routing* algorithms into the capsule networks to compensate for the loss of previous routing algorithms. The evaluation on programs written in different programming language shows that TreeCaps outperforms other approaches in two use case scenarios, which are code classification and function name prediction.

Contents

1 Introduction **1**

1.1 Main Contributions 3

1.2 Thesis Structure 4

1.3 Declaration of Previous Work 7

2 Hierarchical Representation of Software Program **8**

2.1 Our Approach 10

2.1.1 Overview 10

2.1.2 Token Normalization 11

2.1.3 The Bilingual skip-gram Model 12

2.1.4 Hierarchical Models 12

2.2 Empirical Evaluation 13

2.2.1 Data 13

2.2.2 Evaluation Tasks 13

2.2.2.1 Element mappings 13

2.2.2.2 API mappings 14

2.2.3 Threats to Validity 15

2.3 Conclusion and Future Work 16

3 Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification **17**

3.1 Framework of Bilateral Neural Networks 20

3.1.1 Work Flow 20

3.1.2 The Design of Bi-NN 21

3.1.3 Single-Language Classification 22

3.2 Instantiations of Bi-NN 22

3.2.1 AST and Tree-Based Convolutional Neural Networks . . 23

3.2.2 Dependency Trees and TBCNN 24

3.2.3 Gated Graph Neural Networks (GGNN) 26

3.2.4 Token and Sequence based Neural Networks 27
3.3 Empirical Evaluation 28
3.3.1 Datasets 28
3.3.2 Implementation and Research Questions 29
3.3.2.1 Research Questions 30
3.3.3 Summary of Classification Results 31
3.3.4 Details of Classification Results 32
3.3.5 Threats to Validity and Discussions 35
3.3.5.1 Threats to Validity 35
3.3.5.2 Justification for baseline results 36
3.3.5.3 Comparison between neural-network based models 36
3.4 Conclusions and Future Work 38

4 Learning Cross Language API Mapping with Little Knowledge 39
4.1 Background 41
4.1.1 Seed-based Domain Adaptation 42
4.1.2 Unsupervised Domain Adaptation 42
4.2 Our Approach 44
4.2.1 Code Embedding via Word Embedding 45
4.2.2 Domain Adaptation 46
4.2.2.1 Seeding 47
4.2.2.2 Adversarial Learning 47
4.2.2.3 Refinement for Better Alignment 48
4.3 Empirical Evaluation 50
4.3.1 Dataset 50
4.3.2 Implementation 50
4.3.2.1 Evaluation Metrics 51
4.3.2.2 Code Embedding 51
4.3.2.3 Domain Adaptation 52
4.3.3 Evaluation 53
4.3.3.1 RQ1. Effectiveness of SAR in Mining API Mapping 53
4.3.3.2 RQ2. Effect of Different Refinement Approaches 56
4.3.3.3 RQ3: Effect of Each Component 58
4.3.4 Explainability Analysis of the Results 58
4.3.4.1 Effect of Refinement on Frequent vs Rare tokens 59

4.3.4.2 Retrieved Results Comparison 60
4.4 Threats to Validity and Limitations 61
4.5 Conclusion 61

5 Interpretability for Program Representation Learning 63
5.1 Overview 64
5.2 Approach Details 66
5.2.1 Building Attention Neural Networks 66
5.2.1.1 Aggregation Using Attention Mechanism 68
5.2.1.2 Objective Function 68
5.2.2 Deriving Statement-Level Attention Scores 68
5.2.3 Finding Decision-Changing Subset of Code Component . 68
5.2.4 Definition of Code Component 68
5.2.5 Different Types of Code Components 69
5.2.5.1 Single Code Component 69
5.2.5.2 Dependency Code Component 70
5.2.5.3 Finding a Subset of Decision-Change Code Components 71
5.2.5.4 Problem Formulation 71
5.2.5.5 Greedy 1-best Search 73
5.2.5.6 Attention-based Beam Search 74
5.3 Evaluation 77
5.4 Future Work 78

6 Tree-based Capsule Network for Code Processing 80
6.1 Introduction 80
6.2 Tree-based Capsule Networks 82
6.2.1 Tree-based Convolutional Neural Networks 82
6.2.2 The Primary Variable Capsule Layer (PVC) 83
6.2.3 The Secondary Capsule Layer (SC) 84
6.2.3.1 Sharing Weight across Child Capsules with Dynamic Routing (DRSW) 84
6.2.3.2 Variable-to-Static Routing (VTS) 85
6.2.4 The Code Capsules layer 86
6.2.4.1 Code (Functionality) Classification 86
6.2.4.2 Function (Method) Name Prediction 87
6.3 Empirical Evaluation 87
6.3.1 Set up for Code Classification 88

6.3.2 Set up for Function Name Prediction 89
6.3.3 Model Analysis 90
6.3.3.1 Robustness of Models 90
6.3.3.2 Comparison between the Two Routing Algorithms 91
6.3.3.3 Effect of the Secondary Capsule (SC) Layer . . 92
6.3.3.4 Effect of the Number of Capsules (N_{sc}) in the Secondary Capsule (SC) Layer 92
6.3.3.5 Effect of the Variable-To-Static (VTS) Routing Algorithm 93
6.3.3.6 Effect of the Dimension of Capsules (D_{cc}) in the Code Capsule (CC) Layer 93
6.3.3.7 Sensitivity to Input Code Sizes 94
6.4 Conclusion 95

7 Related work **96**
7.1 Learning Code as Natural Languages Tokens 96
7.2 Learning Programs with Structures 97
7.3 Combining syntactical and semantic information 98
7.4 Bi-lateral representations for cross-language learning 98
7.5 Intepretability for Code Processing 99

8 Conclusion **100**
8.1 Summary 100
8.2 Future Directions 101

Bibliography **103**

List of Figures

2.1 Abstract syntax trees — illustration of code hierarchical structure and composability. 10
2.2 An overview of hierarchical learning approach to find mappings across Java and C# 11

3.1 Overview of the cross-language algorithm classification work flow 20
3.2 Setting of single-language algorithm classification 20
3.3 Structure of a TBCNN, adapted from [81] 23
3.4 A Sample AST Expanded into a Dependency Tree. 24

4.1 Overview of SAR: Domain Adaptation for API Mappings 44
4.2 Domain adaptation steps to align two vector spaces 46
4.3 Unsupervised Model Selection Criteria 52

5.1 Overview of AutoFocus approach 65
5.2 Attention mechanism as the aggregation layer for the neural network 66
5.3 Example of different component types 70
5.4 Example of a program after broken down into smaller statement sub-trees 71
5.5 Search Graph Example 73
5.6 Example of Greedy 1-best Search 73
5.7 Example of Beam Search, beam size $b = 2$ 74
5.8 Histogram of Pearson Correlation Coefficients of all test data . . 77
5.9 AutoFocus visualization of attention scores in Visual Studio Code 78

6.1 Overview of TreeCaps: Source codes are parsed, vectorized and fed into the TBCNN to extract node features, then the node features are combined through the TreeCaps network. 82
6.2 Comparisons between the Two Routing Algorithms 91
6.3 Sensitivity of Input Code Sizes 94

List of Tables

2.1 Overview of the training data set for cross language mappings task 13
2.2 Average MAP scores for element mappings at various granularity levels . . . 14
2.3 Precision of API method mappings . . . 15

3.1 Results for cross-language binary classification by different code learning techniques. . . . 32
3.2 Single-language algorithm classification results . . . 33
3.3 Sensitivity Analysis . . . 34
3.4 Results of cross-language algorithm classification with different dependency trees. . . . 34

4.1 Example of Seeds from the Signature-based Matching Heuristic 50
4.2 API Mapping Results . . . 53
4.3 Accuracy of 1-1 Class-Level Mapping when compares with StaMiner and MAM . . . 53
4.4 Accuracy of 1-1 Method-Level Mapping when compares with StaMiner and MAM . . . 54
4.5 Examples of newly found APIs in Java and C# . . . 56
4.6 Accuracy of the filtered mappings using various similarity thresholds . . . 57
4.7 Different ways to combine refinement heuristics . . . 58
4.8 Ablation Study – effects of each component . . . 59
4.9 Effect of Refinement on Frequent vs. Rare Tokens . . . 59
4.10 Retrieved API mapping results from sample queries produced by SAR and Api2Api. . . . 60

6.1 Experimental Settings . . . 88
6.2 Performance in Code Functionality Classification compared. A '-' means that the model is not suited to use the relevant node representation or the parser and thus not evaluated. . . . 89

6.3 Performance of TreeCaps and the baselines for Function (Method) Name Prediction 90
6.4 Model robustness, measured as percentage of predictions changed wrt. semantic-preserving program transformations. The lower the more robust. 91
6.5 Performance of TreeCaps after removing the Secondary Capsule Layer 92
6.6 Effect on N_{sc} on the classification accuracy for the OJ Dataset . 93
6.7 Effect of different model variants for the OJ Dataset 94

Acknowledgements

It is a great pleasure to express my respect to the many people who have supported me throughout my doctoral study at SMU.

First, this Ph.D. thesis would not have been possible without the constant help from my Ph.D. advisor, Prof. Lingxiao Jiang. We spent hundreds of hours discussing research projects and emailing them, while he patiently taught me how to deal with hard problems and acquire a ”good taste” for research problems. Ever since I started working with Lingxiao, he has been pointing me to important research questions while also giving me enough freedom to explore my own interests. This dissertation would not have been in its present state without his visionary understanding of the field, and his belief that great research impact is possible.

I would also like to thank Prof. Yijun Yu from The Open University, who although not officially related to my Ph.D., acted as a remote advisor. We frequently chatting about new ideas, while he patiently explained to me concepts that I have not known before. He also supported me in the engineering parts, which helped me a lot to increase the experiment's progress. Although being at the Open University, his support was vital throughout this Ph.D. Furthermore, I want to thank Prof. Yijun Yu together with Prof. Meng Wang from the University of Bristol for their generous sponsorship to invite me to visit them for three months in the UK.

I would also like to thank all the other thesis committee members, Prof. David Lo and Prof. Jiang Jing, for their generous time and commitment. Their constructive comments and suggestions made this thesis complete.

My thesis research would not be possible without financial support from SMU. Finally, I am grateful to my parents, whose unceasing love and support have helped me go through all the difficulties to complete this venture. I dedicate this thesis to them.

Chapter 1

Introduction

Software is pervasive in present-day society. Many parts of life, such as health services, energy, transportation, etc., relies upon high-quality software. Unfortunately, building software is an expensive procedure: software engineers need to handle the complexity of software while at the same time evading bugs, and delivering highly functional software products on schedule. There are ongoing interests for developments in software tools that help make programming progressively solid and viable. New techniques are continually looked for, to alleviate the complexity and help engineers to build better software.

Machine learning is one of the techniques that have been utilized to mine knowledge from existing artifacts as a stage forward to reduce software maintenance costs and detect bugs. The application of Machine Learning to solve software engineering problems and has been one of the new, hotly debated issues in software engineering. The mined knowledge can be utilized for understanding big systems, reducing software maintenance costs, recognizing bugs, code refactoring, and so forth. With the increasing availability of open-source projects on centralized code hosting services, such as Github, Bitbucket, Gitlab, this new direction is now possible. Concretely, this field is located at the intersection of machine learning, software engineering, and programming language research. The goal is to create probabilistic source code models that learn how developers use code artifacts within the existing code while taking into account the highly structured information within code (aka Program Representation Learning). These models are then used to augment existing tools with statistical information and enable new machine learning-based software engineering.

In this dissertation, we aim to propose machine learning models that can capture the rich structural representation of the source code. Such representations can be generated from traditional program analysis and contain lots of

patterns that can be learned by using machine learning.

Moreover, developers usually work with multiple languages or platforms at the same time to address the different requirements for the business. In general, the process to adapt the source code written in one language (or platform) to another is called domain adaptation and it is pervasive in software engineering. The key challenge when working on multiple domains is to find the similarity across them as a stage forward to adapt the domains together. For example, software migration can be considered as a domain adaptation problem as the goal is to migrate the code written in one language to another language. The domains, in this case, are the software artifacts written in different languages (or different platforms). As such, the goal of the domain adaptation for software is to automatically find the similarity across the domains to assist the developer in the process of adapting them.

The first problem we aim to tackle is the API mapping task. To address the different requirement for business, software companies often develop software in one language and then migrate them to another language, or migrate the software to work on multiple platforms and devices. The process of migrating software between languages and platforms is called software migration. The domains, in this case, are the code written in different languages (or different platforms). The goal of the domain adaptation for software migration is to automatically find the similarity across the languages to assist the migration process of the developers. Toward helping developers in the process of code migration, there exist semi-automatic approaches and supporting tools [46, 48, 88, 137]. Those tools and methods require users to define the migration rules between the respective program constructs and the mappings between the corresponding Application Programming Interfaces (APIs) that are used in two languages. The existing tools and methods expect programmers to specify such API mappings. There are usually a large number of API mappings and many of them are newly introduced from time to time as well. Thus, existing tools can support only a subset of needed API mappings. As such, we need a method to automatically mine the API mapping across the language to reduce the human effort to manually collect and label the mapping.

Algorithm classification is another source code processing problem for code learning that we aim to tackle in this dissertation. Algorithm classification is to automatically identify the classes of a program based on the algorithm(s) and/or data structure(s) implemented in the program. It can be useful for various tasks, such as code reuse, code theft detection, and malware detection. It can

be even a greater challenge to bring the benefit of algorithm classification across different programming languages, to facilitate program reuse and synthesis across languages, reducing the need of reimplementing the same algorithms in different languages repeatedly.

Thesis Statement: In this dissertation, we aim to (1) propose novel representations of programs; (2) propose novel machine learning techniques that can model such representations demonstrate the usefulness of such approaches in various source code processing tasks in software engineer, such as API mapping, method name prediction, and algorithm classification; and (3) propose a novel mechanism to interpret such code modeling techniques. For (1), we aim to leverage the well-established foundation of program analysis to represent the code in a more meaningful way, (2) is about the learning algorithms that can extract useful patterns from the representations; and (3) aims to interpret the output of the learning algorithms.

1.1 Main Contributions

Our contributions in this dissertation are as follows:

- **Hierarchical representation of source code**: We present a novel approach to model the source code as a hierarchical representation of different code levels, e.g., token level, statement level, function level, etc., to better capture the semantic information of source code from finer-grained to coarser-grained level for the API mapping task

- **Enhanced AST with semantic information**: We propose a novel tree structure representation of source code, so-called the Dependency Tree, which is the enriched version of the Abstract Syntax Tree with def-use chain information. We also present a framework of *bilateral* neural networks (Bi-NN), an idea adapted from the area of neural machine translation [21, 50, 119] and Siamese neural networks [21, 84], aiming to generalize a previous study on cross-language program classification.

- **Unsupervised API Mapping**: We propose a deep learning-based approach that can adapt the two domains with almost no parallel data, namely SAR (Seeding, Adversarial training, and Refinement). We realize that the underlying goal of state-of-the-art API mapping techniques is essential to find a transformation that can align two different domains (in our context, the two vector spaces for APIs in two different languages).

Our key idea is to adapt the Generative Adversarial Network to align the vector spaces without the need for parallel data. We demonstrate the success of our model for the API mapping task.

- **Interpretability for code modeling**: Since we have proposed several deep learning-based techniques to tackle different domain adaptation problems, the key challenge for such techniques to become practical in the real world usage is due to the lack of interpretability. AutoFocus is an automated approach towards the interpretability of the neural network for code learning. Our key idea to understand the neural network on source code is that we want to quantify the importance of input elements based on their effects on the outputs of the networks We apply the AutoFocus approach to the algorithm classification task. Specifically, an attention mechanism is incorporated in the neural networks for classifying a program according to the algorithm implemented by the program, and attention scores are generated to differentiate various code elements in the program. More importantly, AutoFocus identifies and perturbs code elements in the program systematically, and quantifies the effects of the perturbed elements on the networks' capability in classifying the program.

- **Capsule Network for Code Processing**: Although Autofocus can identify important parts of the program, we realize that the interpretability of AutoFocus depends on the way the neural networks encode the programs. Most of the encoding does not capture dependencies inside the program or one needs to specify the dependency explicitly through program analysis, which may result in noisy and inaccurate information. As such, we propose a novel approach, so-called TreeCaps by applying the capsule principle that is widely used in image recognition to process the source code. From the experiments, we show that the TreeCaps can reach state-of-the-art performance for two interesting tasks in software engineering, which are code classification and function (method) name prediction.

1.2 Thesis Structure

The thesis are presented as follow:

- In Chapter 2, we present a novel approach to model the source code as a hierarchical representation of different code levels, e.g., token level, statement

level, function level, etc., to better capture the semantic information of source code from finer-grained to coarser-grained level for the API mapping task. The API mapping is a very important stepping stone for automated code migration, which reduces the cost of manual effort to migrate programs in different languages. The evaluation results show that this novel way to model the source code produces significantly better results than the existing techniques for API mapping. Inspired by Neural Machine Translation (NMT) and limitations in existing program translation techniques, our goal is to produce a new way of representing code in distributed vectors for any kind of code elements across languages.

- In Chapter 3, we explore the idea to model the code across the languages by incorporating the semantic information of the source code into the tree representation of the code, e.g. Abstract Syntax Tree. We also present a framework of *bilateral* neural networks (Bi-NN), an idea adapted from the area of neural machine translation [21, 50, 119] and Siamese neural networks [21, 84], aiming to generalize a previous study on cross-language program classification [12], to encode code syntactic and semantic information for programs written in two different languages, and train the bilateral neural networks to recognize code implementing the same algorithms across languages. Such a framework enables us to explore different ways to use different kinds of code intermediate representations with different kinds of neural networks to search for optimal algorithm classification solutions. Every instance of Bi-NN can be trained with bilateral programs that implement the same algorithms and/or data structures in two different languages. The trained Bi-NN models can then be applied to recognize code implementing the algorithms and/or data structures in different languages.

- In Chapter 4, we propose a deep learning-based approach that can map APIs across languages without the need for parallel data based on the idea of Adversarial Learning. We realize that the underlying goal of state-of-the-art API mapping techniques is essential to find a transformation that can align two different domains (in our context, the two vector spaces for APIs in two different languages). Given large codebases in two languages, certain similarities between the code bases can likely be exploited to discover APIs of similar functionality across languages, without manually specifying parallel corpora. Such knowledge of similar functionalities may not be big enough for a complete mapping model, but it is small enough to afford

human validation. Once validated, the knowledge can be *transferred* through *adversarial training* techniques to maximize the alignment between the two languages which results in better API mappings.

- In Chapter 5, we propose AutoFocus, an automated approach to rating and visualizing the importance of input elements based on their effects on the outputs of the networks. The rationale for such an approach is to tackle the interpretability for code learning models. Since we have been proposed several deep learning-based techniques to tackle different tasks, the key challenge for such techniques to become practical in the real world usage is due to the lack of interpretability. People treat the model as a black box and expect good output from it. As a result, no one understands how the techniques for source code representation learning work, which results in the hesitation to adopt such techniques to solve real-world problems. The problem is even more severe in the software engineering field, where most of the learning model is built based on some human heuristics with trial and error. Because of such reasons, understanding the reasons and able to interpret the predictions is quite important if one plans to take action based on a prediction. We apply the AutoFocus approach to the algorithm classification task. Specifically, an attention mechanism is incorporated in the neural networks for classifying a program according to the algorithm implemented by the program, and attention scores are generated to differentiate various code elements in the program. More importantly, AutoFocus identifies and perturbs code elements in the program systematically, and quantifies the effects of the perturbed elements on the networks' capability in classifying the program. Our evaluation shows that the attention scores are highly correlated to the effects of the perturbed code elements. Such a correlation provides a strong basis for the uses of attention scores to interpret the relations between inputs and outputs of the algorithm classification neural networks, and visualizing the code elements in the input programs ranked according to the attention scores can facilitate faster program comprehension with reduced code.

- In Chapter 6, we propose a novel deep learning technique called TreeCaps to process source code in an automated way that captures relationships between different parts inside the AST without the need to use program analysis. In our empirical evaluation, we evaluate TreeCaps on two tasks to demonstrate the usefulness of TreeCaps in different use case scenarios.

The two tasks are code classification and function name prediction. Across codebases in C/C++ and Java with respect to commonly compared program comprehension tasks such as code functionality classification and function name prediction, our empirical evaluation shows that TreeCaps achieves better classification accuracy and better F1 score in prediction compared to other code learning techniques such as GGNN, Code2vec, ASTNN, and TBCNN.

1.3 Declaration of Previous Work

This thesis contains work that has been previously published in conferences that have been co-authored with different people. The author of this thesis has been the first author and main contributor to all these publications. Specifically, Chapter 2 contains work published in "Hierarchical Learning of Cross-Language Mappings through Distributed Vector Representations for Code" (Nghi et al., ICSE 2018). Chapter 3 contains work published in "Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification" (Nghi et al., SANER 2019). Chapter 4 contains the work published in "SAR: Learning Cross-Language API Mappings with Little Knowledge" (Nghi et al, FSE 2019). Chapter 5 contains work found in "AutoFocus: Interpreting Attention-based Neural Networks by Code Perturbation" (Nghi et al., ASE 2019). Finally, chapter 6 contains the work found in "TreeCaps: Tree-Structured Capsule Networks for Program Source Code Processing", this work has been published as a workshop paper in NeurIPS 2019 and is under review as a full conference paper in NeurIPS 2020.

Chapter 2

Hierarchical Representation of Software Program

Automated program translation (a.k.a. language migration) can be very useful for software development as it may help reduce developer coding time, especially for functionalities and library APIs that need to be implemented and maintained in various programming languages. Take the Apache Lucene as an example: it is a popular information retrieval library, providing many APIs for a third-party client program to access its core functionalities. Lucene was originally implemented in Java and not easy to be used by client programs in other languages. Due to popular demands for its functionalities, it has been ported to other languages (e.g., C#, C++, Python, Ruby, PHP, etc.) to support clients in those languages. Nevertheless, multiple versions of Lucene in different languages increase the cost on its maintenance and development, as new features or bug fixes in one version may need to be manually ported to another version for consistency. An automated approach to translate code among languages can help save much cost, and still be useful even if the approach cannot generate complete translations but can identify likely translation candidates in large code bases.

Existing studies on program translation may be classified in two categories. One is based on grammar rules (e.g., Java2CSharp at https://github.com/codejuicer/java2csharp), which can be very accurate, but inflexible in dealing with different languages or language evolutions as the translations need to be programmed repeatedly for different grammars. The other is based on statistical language models for selected code elements (e.g., for tokens [91], token phrases with contexts [92, 93], or APIs and API sequences [48, 49, 88, 90, 104, 137, 138]), which can deal with different languages but may need to incorporate various kinds of contexts (e.g.,

sequences/co-occurrence relations, data/control dependencies) for selected code elements [49, 93].

In the field of natural language processing (NLP), neural-network-based Neural Machine Translation (NMT) has emerged as an alternative to statistical language models, achieving good results for natural language translation [115]. NMT models use distributed vector representations of words as the basic unit to compose representations for more complex language elements, such as sentences and paragraphs. One prominent distributed vector representation is word2vec [78, 80], which uses neural networks to learn vector representations of words (a.k.a. *word embeddings*) from natural language articles to capture latent semantics with respect to a modeling objective, such as predicting the context given a word or predicting the next word given a context. Also, similarities among different natural languages can be exploited for machine translation [79], which can be applicable for programming languages as there are many across-language code clones too [27].

Inspired by NMT and limitations in existing program translation techniques, **our goal** is to produce a new way of *representing code in distributed vectors for any kind of code elements across languages*.

Our key idea is mainly based on two observations: (1) code clearly has hierarchical structures as illustrated in Figure 2.1 and is often composable, and (2) code structures (in addition to its textual appearance) often accurately reflect its semantics, which are different from natural languages. That means, NMT may be able to generate distributed vector representations that can closely reflect the code semantics if the code token streams can be enriched with its structural information, and generate vector representations for any composed code elements that are of higher levels of granularity. Therefore, **our approach** works by normalizing and enriching code token streams with structural (and some semantic) information extracted via code parsing, constructing a bilingual skip-gram model to generate distributed vectors for code tokens in two different languages (a.k.a. *shared embeddings*), and composing shared embeddings for low-level code elements into more complex ones according to code structures. Code elements in different languages but having similar shared embeddings will thus become mapping and translation candidates for each other.

Our preliminary evaluations using about 40,000 source files from 9 programs that have multiple versions in both Java and C# show that our approach can automatically learn shared embeddings from existing code across Java and C#, and achieve around 50% precision in recommending top-10

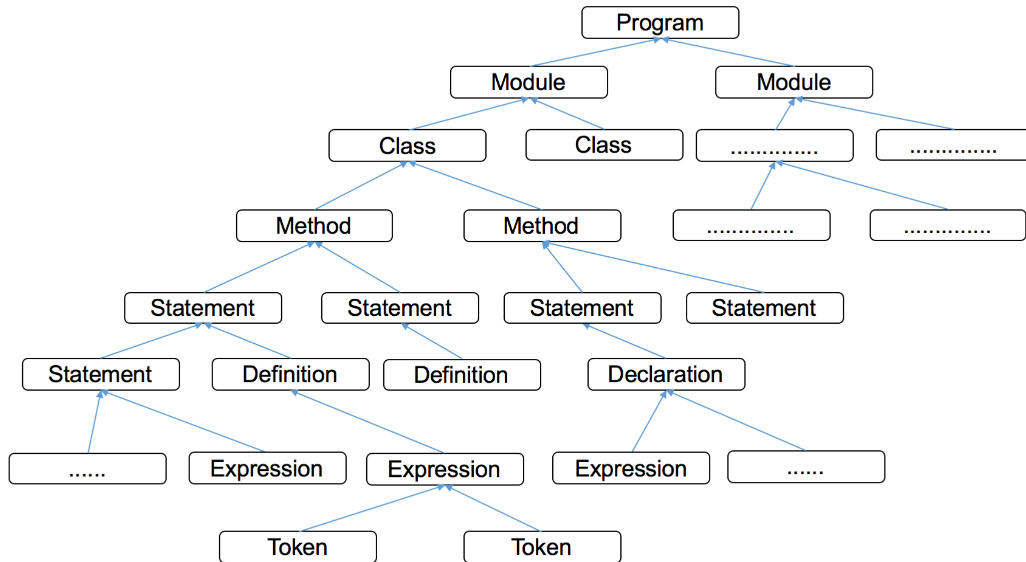


Figure 2.1: Abstract syntax trees — illustration of code hierarchical structure and composability.

cross-language code mappings at various levels of granularity. Compared with existing tools for identifying API mappings (StaMiner [88]), our approach can identify more than 400 more library API methods and classes accurately.

Our main contributions are as follows:

- We propose a new way to add structural information into source code token streams and adapt word embeddings to learn vector representations for code tokens across languages.
- We allow hierarchial compositions of vector representations for simpler code elements into more complex ones according to code structures, and thus can produce vector representations for any code structures across languages.

2.1 Our Approach

2.1.1 Overview

Figure 2.2 is the overview of our approach. We first collect parallel corpus across languages for training bilingual embedding models. A parallel corpus is a collection of source code in one language and their translation into another language. We utilize the similarity among file names to identify files in different languages that implement a same functionality. Taking Lucene as an example, the file `AbstractEncoder.cs` in its C# version has the same name as the file

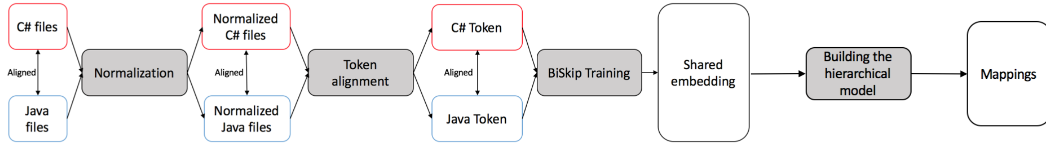


Figure 2.2: An overview of hierarchical learning approach to find mappings across Java and C#

`AbstractEncoder.java` in its Java version. Thus, the files in the parallel corpus are considered to be "semantically aligned" with each other and used for later steps. We then normalize the token streams in the files to remove semantic-irrelevant information (e.g., some variable names) and add more structural and some semantic information (e.g., syntactic node types, data types, and method signatures). The normalized token streams are then used as input for the Berkeley aligner [72] to generate token-level alignment information indicating potential synonyms across languages, and the token-aligned data is then used to learn bilingual vector representations for the tokens. Finally, vector representations for low-level tokens are composed together to form representations for code elements of higher levels of granularity. Code elements of similar vector representations across languages (i.e., shared embeddings) will be identified as mapping candidates for each other.

2.1.2 Token Normalization

This step (1) converts each raw token into its signature version and (2) adds structural keywords for the tokens based on ASTs.

Convert a raw token into its signature: This is to normalize the effects of various kinds of identifier names as some names are important for code semantics while some others are not. For example, `class Text` in Lucene and `class Text` in Java SDK are different types even though their lexical names are the same. Thus, we replace the names with their type signatures (including their package and class names) for differentiation. Similarly, function names are replaced by their full signatures. For variable names, if they are non-primitive types, they are replaced by the type signatures, similar to class names; if they are primitive types, they are replaced by a type-specific token. Tokens having no effect on code operational semantics, such as '', ',', ';', are removed. The below illustrates how three main kinds of tokens are normalized:

```
int i; ==> int int_id                         // 2 tokens
CommonTree ==> Antlr.Runtime.Tree.CommonTree  // 1 token
```

```
lexer.Emit();  ==> Antlr.Runtime.SlimLexer.Emit() // 1 token
```

Add structural keywords for the tokens: This is to add relevant AST node types for the tokens into the token streams so that the later learning steps may utilize more information implicit in raw code texts. The below snippet illustrates this step:

```
Console.WriteLine("out"); ==>

expr_stmt expr func_call System.Console.WriteLine(String)
argument literal_type string  // 7 tokens
```

2.1.3 The Bilingual skip-gram Model

Our goal here is to learn distributed vector representations for cross-language code tokens, which can then serve as the basis for more complex composed code elements. We use the bilingual skip gram model (BiSkip) [75] to achieve the goal. The motivation behind BiSkip is to learn shared embeddings between tokens cross-lingually rather than just monolingually: Rather than just predicting the tokens in one language, they use the tokens in one language to predict their aligned tokens in another language and vice versa. For example, from a large corpus of Java and C# code, the BiSkip model may be able to learn that the token `readonly` in C# is aligned to and has the same meaning as the token `final` in Java, and `final` often occurs together with `public` and `int` to define certain constants. Then, when given the token `readonly`, we can use the BiSkip model to substitute `final` for `readonly` and predict that its surrounding tokens are `int` and `public`. The BiSkip model has been shown to perform well for both bilingual and monolingual tasks [75]. We utilize the Berkeley aligner [72] to generate token alignments from the code token streams to be used by BiSkip.

2.1.4 Hierarchical Models

Once we get the vector representation for tokens across languages, we want to generate representations for more complex code elements, such as expressions, definitions, declarations, statements, methods, classes, and modules (Figure 2.1) so that code mappings and program translations can be done for more complex elements. Since all of the elements are hierarchical compositions of elements at lower levels of granularity including tokens, our intuition is to generate

Project	Java			C#		
	Ver	Files	Methods	Ver	Files	Methods
Antlr(AN)	4.0.0	276	3560	4.0.0	630	5049
db4o(DB)	8.0	5556	38525	8.0	3845	23248
fpml(FP)	1.7	130	727	1.7	135	1038
Itext(IT)	7.0.5	1147	10003	7.0.5	2647	18842
JGit(JG)	4.10.0	1394	13862	4.10.0	1079	9203
JTS(JT)	4.0.0	958	7883	4.0.0	1035	6640
Lucene(LC)	7.1.0	6098	48038	7.1.0	2930	19961
POI	4.0.0	3295	29172	4.0.0	2794	16717
Neodatis(ND)	2.1	960	10525	2.1	987	12153

Table 2.1: Overview of the training data set for cross language mappings task

representations for elements at higher levels of granularity by composing the shared embeddings of their constituent elements.

According to [63], simply averaging word embeddings of all words in a text can be a strong baseline for representing the whole text for the task of short text similarity comparison. Variants of this simple averaging strategy exist, such as averaging the embeddings with their weights measured in terms of term-frequency/inverse-document-frequency (TF-IDF) to decrease the influence of the most common words. As a preliminary exploration, we only consider 3 levels of granularity of this task: expressions, statements, and methods, and use the simple averaging operation to compose shared embeddings according to the structures of code abstract syntax trees.

2.2 Empirical Evaluation

2.2.1 Data

Table 2.1 is a summary of our training dataset. We consider the language pair C# and Java in all the evaluations. We collect the comparable dataset as in StaMiner [88]. We use the implementation of BiSkip from [15] to generate the vector representations of tokens.

2.2.2 Evaluation Tasks

2.2.2.1 Element mappings

As described in Section 2.1.4, we aim to build vector representation for compositional cases in order to find good mappings in a hierarchical model across languages. We extract all the expressions, the statements and the methods of each project in our data set by traversing the AST representation to identify

Levels	Expressions	Statements	Methods
Avg. MAP, k = 1	0.31	0.38	0.44
Avg. MAP, k = 5	0.43	0.50	0.58
Avg. MAP, k = 10	0.57	0.53	0.59

Table 2.2: Average MAP scores for element mappings at various granularity levels

the element type, then we manually defined ground truth elements pairs. We treat each element in Java as a query to retrieve the top-k elements of C#. Then we use Mean Average Precision (MAP) as the metric to evaluate this task. Due to the limitation of pages, instead of showing the MAP score for each type of element of each project, we calculate the *average MAP* of all project for each type of element. Table 2.2 shows the results of the evaluations, with k = 1, 5, 10, respectively.

2.2.2.2 API mappings

We use the task described in StaMiner [88] to evaluate how effective our approach. We consider 2 types of API names: classes and methods. For each *name* in Java, we get its vector representation and use it as a query. The query is used to find the top-k nearest neighbors among the shared embeddings for C# names. In this task, we only consider k = 1, which means we consider only the exact match of the query. Since there are too many APIs to build ground truth manually, we randomly select 100 APIs of each project for this task. Table 2.3 shows the precisions for class mappings and method mappings.

Compared to StaMiner [88] and DeepAM [49] that mine API mappings by using statistical machine translation techniques and deep learning, our work is more generalized in term of the kind of code elements supported. Their work only focuses on learning the mappings between language SDK APIs, while our approach allows mappings among any kind of structural code elements in a language beyond SDK APIs. Although finding the mappings for SDK APIs in different languages is a commonly needed and important task, developers often need more mappings for program elements (e.g., variable names, data structures, statements, method implementations, etc.). We believe that being able to find the mappings among program elements of any granularity is a important step to reach the goal of automated language migration.

We also found that our approach detects correctly about 400 more SDK API method mappings and 150 more SDK API class mappings that were not set in the latest mapping files in the Java2CSharp tool, while StaMiner detects

Level\Project	AN	FP	IT	JG	JTS	LC	POI	DB	ND
Class	0.85	0.82	0.88	0.69	0.83	0.82	0.78	0.86	0.79
Method	0.81	0.80	0.83	0.77	0.72	0.87	0.89	0.82	0.83

Table 2.3: Precision of API method mappings

120 more SDK mappings for both classes and methods and 84 of which are also in our mappings. In this evaluation, we only consider k=1 for the top-k nearest neighbors. We expect to find even more mappings if we consider a larger k and we can find more mappings for APIs that are *not* in the language SDK libraries. We leave these evaluations for future work.

2.2.3 Threats to Validity

For the model training, we use the same settings as described by Mikolov et al. [80], which may not be the best with respect to our dataset. We will do more empirical research to choose better hyper-parameters to improve training results. The normalization step is mostly based on srcML [30] to get the AST of source code. At this moment, srcML supports four languages (`C, C++, C#`, `Java`), and we only perform experiments on the Java – C# pair. In the future, we want to explore the generalizability of our approach with more programming languages supported by srcML and beyond.

The correctness of our API mappings results were checked by ourselves manually, which may be biased and incomprehensive. To evaluate the actual correctness and usefulness of our API mappings, we plan to do more large-scale evaluations by using Java2CSharp to see how our mappings can help reduce the compilation error rates when compared with StaMiner during actual migration of projects.

As described in [117], source code is very localized. Since we treat a whole file as a corpus, which means we ignore the localness of the tokens. The skip-gram model focuses on capturing the global regularities over the whole corpus, and neglects local regularities, thus ignoring the localness of software. A natural way to collect local-awareness parallel corpus is that we can slice the file into multiple slices based on dependence information, then we align the parallel corpus based on the similarity of the slices.

2.3 Conclusion and Future Work

This work proposes an approach to learn code element mappings across programming languages based on distributed vector representations. By utilizing the alignment of files in projects with multiple versions of implementations in different languages, we learn the alignment of tokens in an unsupervised manner, and generate the shared embeddings for tokens across languages. Then, the shared embeddings for more complex code elements are composed from the embeddings of their constituent elements and tokens from bottom up according to the hierarchical structures of the syntax trees of the code. Our evaluations show that our approach can map many code elements between Java and C# accurately. This can serve as a foundation for more complicated tasks, such as program translation, cross-language program classification, code clone detection, code reuse, and even synthesis.

Chapter 3

Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification

Algorithm classification is a long-standing problem related to program reuse and synthesis [18, 28, 116]. It aims to assign class labels or concepts to programs based on code structures and semantics [77]. Automated classification of a piece of code could ease a number of software engineering tasks, such as program comprehension [77], concept location [107], algorithm plagiarism detection [131], bug fix classification [73, 103] and malware detection [24]. The algorithm labels for the code can serve to some extent as the summary of the code [98], which help to modularize, abstract, analyze, and reuse the code.

Even though it is different from the problem of program equivalence checking [17], this problem remains challenging because what is considered to be the "same algorithm" can look different under different situations. An "appropriate" classification should not only take sufficiently detailed information about the code into consideration, but also ignore irrelevant details depending on the abstraction level of an algorithm class. For example, a program A implementing `bubblesort` for an integer array may not be the "same" as a second program B implementing `bubblesort` for integers stored in a linked list, and both of them may not be the "same" as a third program C implementing `mergesort`, while the three programs may all be considered the "same" as variant of sorting algorithms.

It can be even a greater challenge to bring the benefit of algorithm classification across different programming languages, so as to facilitate program reuse and synthesis across languages, reducing the need of reimplementing the

"same" algorithms in different languages repeatedly.

Past studies on algorithm classification can neglect differences in different programming languages by simply processing the programs as a bag or a sequence of tokens or simple call graphs[17, 107], which do not utilize the rich code syntactic structures and semantics, or by taking advantages of system-level APIs used and/or higher-level descriptions available in human languages[126, 129]. On the other hand, program classification and functional cloning studies utilizing code syntax structures are mostly limited to individual languages [59, 61, 81, 120], which have not been adapted to the classification problem across languages.

Our research goal here is to find a suitable representation for given pieces of code in different languages that can be used to identify the algorithm classes of the code. Specifically, we present a framework of *bilateral* neural networks (Bi-NN), an idea adapted from the area of neural machine translation [21, 50, 119] and Siamese neural networks [21, 84], aiming to generalize a previous study on cross-language program classification [12], to encode code syntactic and semantic information for programs written in two different languages, and train the bilateral neural networks to recognize code implementing the same algorithms across languages. Two technical aspects of the framework are important for the effectiveness of cross-language algorithm classification:

1. One is to build a bilateral structure of neural networks that consists of *two* (thus the name *bilateral*) underlying neural networks, each of which encodes code in one language, and another classification model on top of the two to link them together.

2. The other is to explicitly embed code dependencies (e.g., variable def-use relations) into the intermediate representations (IR) of code for the neural networks to learn code representations.

Such a framework enables us to explore different ways to use different kinds of code intermediate representations with different kinds of neural networks to search for optimal algorithm classification solutions. Every instance of Bi-NN can be trained with bilateral programs that implement the same algorithms and/or data structures in two different languages. The trained Bi-NN models can then be applied to recognize code implementing the algorithms and/or data structures in different languages.

We instantiate the Bi-NN framework with token-, sequence-, tree-, and graph-based machine learning techniques to train different Bi-NN models

on large code bases to classify programs into different algorithms. Empirical evaluations on two code bases, (1) 52000 C++ files from previous studies written by computer science students implementing 104 different algorithms and (2) 4932 unique Java and 4732 unique C++ single-file programs from GitHub implementing 50 different algorithms, show that the Bi-NN model trained by tree-based convolutional neural networks (TBCNN) using our custom-built dependency trees (DTs) representing code syntax and semantics achieves a reasonable accuracy of 86% in classifying cross-language programs according to the ground-truth algorithm class labels. The accuracy of this model (referred to as a Bilateral Dependency Tree Based CNN model, or Bi-DTBCNN in short) is the highest among several other Bi-NN models based on bags-of-words, n-gram, tf-idf, long-short term memeory (LSTM), gated graph neural network (GGNN), etc. Even for the simpler problem of algorithm classification in a single language, our evaluations show that DTBCNN models (without using the bilateral structure) achieve the highest classification accuracy of 93% among different models we have evaluated.

The main conceptual and empirical contributions are as follows:

- We generalize a bilateral neural network (Bi-NN) framework for the cross-language algorithm classification task;
- We adapt various learning techniques, including n-grams, bags-of-words, tf-idf, tree-based convolution neural networks (TBCNN), long short-term memory (LSTM), and gated graph neural networks (GGNN) to instantiate the bilateral code representation framework to represent both syntax and semantics for algorithm classification;
- We custom-build a *dependency* tree-based convolutional neural network (DTBCNN) as an extension to TBCNN to encode semantics for more accurate classification;
- We collect a benchmark of 9664 unique programs in Java and C++ implementing 50 algorithms, and evaluate the performance of various Bi-NN models. The results demonstrate the effectiveness of Bi-NN models for cross-language algorithm classification. In particular, Bi-DTBCNN achieves the highest classification accuracy in our evaluation.

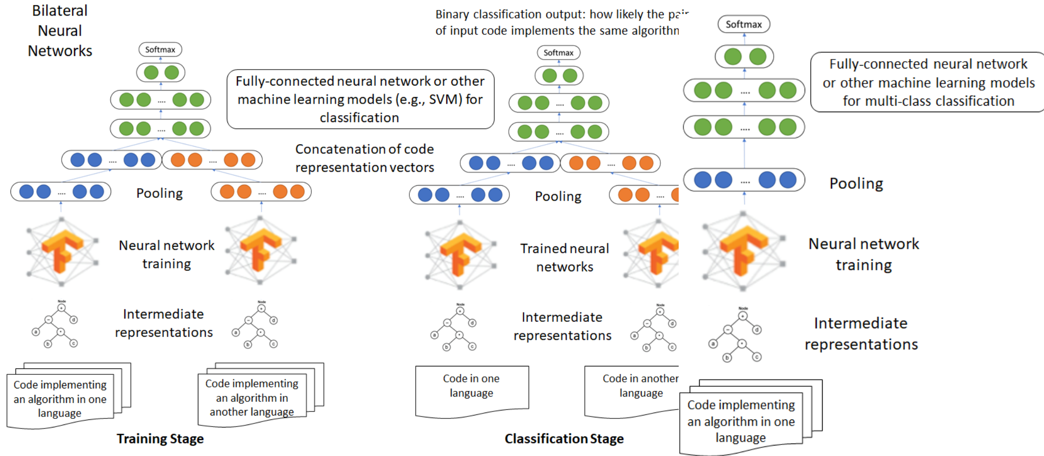


Figure 3.1: Overview of the cross-language algorithm classification work flow

Figure 3.2: Setting of single-language algorithm classification

3.1 Framework of Bilateral Neural Networks

3.1.1 Work Flow

For the purpose of cross-language algorithm classification, we want to align code representations for different programming languages so that a model learned from algorithms in one programming language can be used to recognize code implementing the algorithms in another language. To achieve this purpose, a work flow based on the Bilateral Neural Networks (Bi-NN) model can be used; it consists of two stages (see Figure 6.1): (1) training of an instance of Bi-NN with a set of input program pairs in two languages, and (2) classification by applying the trained Bi-NN models to pairs of test programs.

The training stage takes in pairs of programs in two programming languages as input. Each program in a pair has an algorithm class label, indicating whether the pair implements the same algorithm or not. Parsers are used to convert the input programs to certain intermediate representations (IR) that expose code syntax and semantics, which can be based on tokens, sequences, trees, or graphs. Then, the IR of all the input pairs, either implementing the same algorithm or not, is used to train a Bi-NN model that minimizes the classification errors for the inputs.

The classification stage takes in a pair of test programs in two programming languages without knowning their algorithm class labels, converts the inputs into the same kind of IR as those used in the training stage, and uses the trained Bi-NN model to predict the likelihood for the two test programs belonging to the same algorithm class. We call this classification a binary classification as

its output only tells whether or not two programs belong to the same algorithm class.

One can also utilize such binary classifications to determine the algorithm class for one given test program. For each known algorithm class, one can pick an arbitrary program from the class to form a pair of input programs with the given test program, and feed them into the trained Bi-NN model to predict the likelihood for the pair to be in the same algorithm class. Repeating this step for every known algorithm class produces a set of likelihoods; each indicates the likelihood of the given test program belonging to the corresponding algorithm class. From these predictions, the algorithm class label with the highest likelihood is assigned to the given test program. If none of the likelihood is high enough (e.g., above 0.5), one may choose to leave the test program as unknown.

The capability of the classification is naturally limited by the known algorithm classes used for training. In our evaluation later (Section 3.3), we show the effectiveness of such classifications for the numbers of algorithm classes ranging from 10 to 104.

3.1.2 The Design of Bi-NN

The key component of the proposed work flow is the Bi-NN. It is constructed as two underlying subnetworks and another classification model (which can be a neural network or other kinds of classifiers) on top of the two.

Each of the two underlying subnetworks can be any neural network, such as a LSTM, GGNN, or others, as we show later in this chapter. During the training, each subnetwork takes in code representations of one-specific language to recognize the code in that language. For our cross-language algorithm classification task, the two subnetworks are designed to take in code representations of different languages. If both subnetworks took in code representations of the same language, the Bi-NN could be suitable for single-language classification too (see the next subsection).

The classification model on top connects the two underlying subnetworks. It can be another neural network. As illustrated in Figure 6.1, It consists of (1) two pooling layers, each of which aggregates the code learning output from one of the two subnetworks, (2) a "joint feature representation layer" that concatenates the pooling outputs of the two subnetworks, (3) two or more fully connected hidden layers above the joint feature representation layer, and (4) a Softmax layer on the top to determine how likely the input code pair belongs to

the same algorithm class. The layers of (3) and (4) essentially form a classifier that can be trained for the inputs from the layers below. They do not have to be neural networks, and they may be substituted by any classifier, such as Support Vector Machines (SVM [58]) and Random Forests [56].

Note that, when we instantiate the Bi-NN framework with different machine learning models, it becomes essentially the same as Siamese networks in the literature [21]. Siamese networks are a class of neural network architectures that contain two (or more) subnetworks, which are merged via an energy cost function over a joint layer on top of the subnetworks. Its high-level architecture is similar to the Bi-NN framework illustrated in Figure 6.1. Many choices for the subnetworks and the loss function over the joint layer can be adopted for different tasks. Section 3.2 provides ways to instantiate the Bi-NN framework with different concrete code representations and machine learning models for the task of cross-language algorithm classification. For cross-language training specifically when node types are used, we implement an additional alignment step to ensure the shared node types multiple language are mapped to the same node.

3.1.3 Single-Language Classification

Besides the task of cross-language algorithm classification, we can also use the Bi-NN framework to classify programs in a single language. That is, the input programs are all in one language only. Although we may use both sides of the Bi-NN framework as mentioned in the previous subsection to train the classification model, it is more straightforward to use "half" of the Bi-NN framework. As illustrated in Figure 3.2, by disabling the "joint feature representation layer" and the right side of the Bi-NN framework, we can obtain a classification model that can be trained with programs in one language to classify algorithm classes in the same language too.

3.2 Instantiations of Bi-NN

The Bi-NN may be instantiated with different kinds of structures to represent various syntax and semantic information of given pieces of code and different kinds of neural networks to learn the code representations. This section presents several variants that we consider to be promising for learning and classifying algorithms.

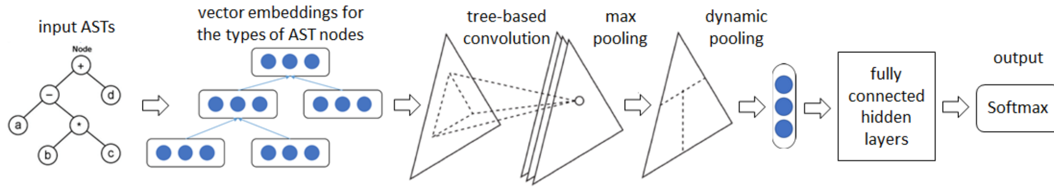


Figure 3.3: Structure of a TBCNN, adapted from [81]

3.2.1 AST and Tree-Based Convolutional Neural Networks

Abstract syntax trees (AST) are a very commonly used code representation that faithfully encodes the syntax of a program, and given a well-formed AST, the responding program can be regenerated. Thus, it is a natural way to use AST as the intermediate representation and a tree-based neural network to learn the representation in the Bi-NN framework.

Mou et al. [81] have proposed to use tree-based convolutional neural networks (TBCNN) to learn AST and classify C++ programs. Figure 3.3 illustrates the structure of a TBCNN. Each AST node is represented as a vector by using an encoding layer that basically embeds AST node types into a continuous vector space where contextually similar node types are mapped to nearby high-dimension points in the vector space. For example, the node types 'while' and 'for' are similar because they are both loops and thus their vectors will be close to each other. Given an AST where every node is turned into a vector representation, Mou et al. [81] uses a CNN and a set of fixed-depth subtree filters sliding over the AST to "convolute" structural information of the entire tree. A dynamic pooling layer [113] is applied to deal with varying numbers of children of AST nodes to generate one high-dimension vector to represent the whole tree. Finally, they use a hidden layer and an output layer, similar to the common neural network on top in our Bi-NN framework, to classify the programs.

The TBCNN was used to classify algorithms written in one language only and it only encodes code syntax without explicit semantics, but it inspires us to improve it for cross-language classification that encodes more code semantics. We can instantiate the two subnetworks in the Bi-NN framework with two TBCNN for different languages, say, one for Java, and the other for C++, and then we can train a bilateral tree-based convolutional neural network model (Bi-TBCNN) for cross-language algorithm classification.

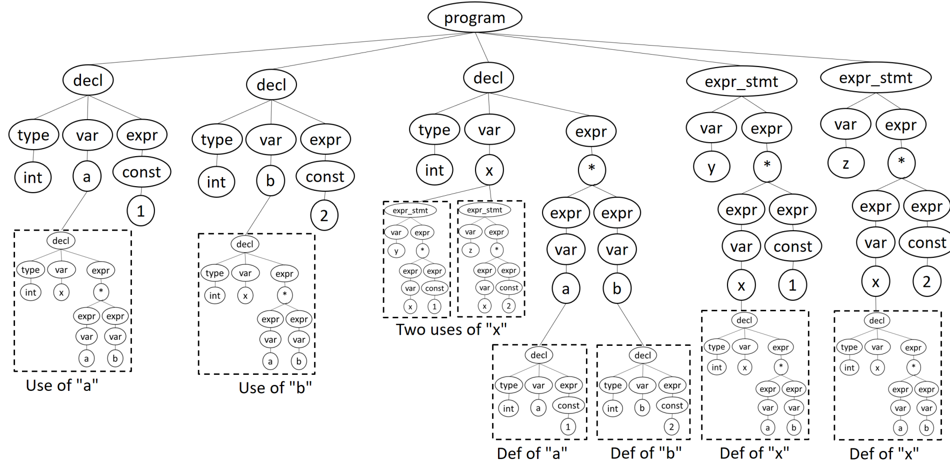


Figure 3.4: A Sample AST Expanded into a Dependency Tree.

3.2.2 Dependency Trees and TBCNN

Although abstract syntax trees can faithfully encode the syntax of a program, it may not be obvious or explicit about various kinds of semantic information in the program, such as def-use relations, call relations, class inheritances, etc. Therefore, we consider encoding such semantic dependency information directly into abstract syntax trees so that the *dependency tree* based code representation can help tree-based neural networks to learn code semantics more accurately to recognize code of different algorithms.

Our basic idea is to insert additional nodes that represent some semantic information relations into AST to form *dependency trees*: given a program or a code snippet, it is first parsed into AST represented in the Pickle format using our tool; then, dependency relations, especially def-use relations, are extracted from the code using srcML and srcSlice [8, 30], and the nodes related to a Def are appended to the nodes representing the uses of the Def, and vice versa.

For example, Figure 3.4 (ignoring the boxes with dashed lines first) represents the AST of the following piece of code:

```
int a = 1; int b = 2; int x = a * b;
y = x + 1; z = x + 2;
```

In this example, the defs "`int a = 1`" and "`int b = 2`" affect the definition of the variable `x` in "`int x = a * b`", and `x` is used to compute both `y` and `z`. Therefore, the expanded *dependency tree* is a tree shown in Figure 3.4 (including the boxes with dashed lines), where (1) the subtrees in the original AST representing "`int x = a * b`" are duplicated and inserted as children of the nodes representing the definitions of `a` and `b` respectively; (2) the subtrees representing the definitions of `y` and `z` are duplicated and inserted as children

of the node representing the definition of `x`; and symmetrically, (3) the subtrees representing the definitions of `a` and `b` are also duplicated and inserted as children of the nodes representing the uses of `a` and `b` respectively; and (4) the subtrees representing the definition of `x` are duplicated and inserted as children of the nodes representing the uses of `x`.

Notice that many subtrees are duplicated multiple times in our dependency trees. In the literature, program dependency *graphs* are typically used to avoid duplication, by referring to the dependencies as edges between dependers and dependees. However, our intuition is that representing the relations as trees and allowing such node duplication can have an advantage of separating the contexts of each use-def from each other to faciliatate context-sensitive learning while making it more efficient to train tree-based neural networks than graph-based ones. Therefore, we instantiate Bi-NN with dependency tree-extended tree-based convolutional neural networks to train cross-language algorithm classification models, which we simply call Bi-DTBCNN.

Apart from tree structure differences between DTBCNN and TBCNN, we also employ a different vector embedding strategy for the tree nodes to bootstrap the training of the tree-based neural networks with more code semantics.

Tree-node embedding for boostrapping the training

As illustrated in Figure 3.3, the training of TBCNN needs a vector representation for each tree node. Mou et al. [81] use the "coding criterion" from Peng et al. [102] to learn the vector for each AST node type. We adapt the skip-gram neural network model used for *word2vec* [78] to the context of AST nodes. The skip-gram model, given an input word in a sentence during training, looks at the words spatially nearby and picks one at random, and produces the probability for each word in the whole vocabulary to be a "nearby word" of the input word; i.e., it can be used to "predict the contextual words for an input word". When such a model is trained to produce the probabilities of nearby words, its hidden layers can produce numerical vectors representing the words, i.e., the word embeddings.

We apply this idea for the so-called *AST2vec* task. That is, we view AST node types as the vocabulary words and consider nodes to be "nearby" in the AST if they have parent-child or sibling relations, and train a skip-gram neural network model using all the AST generated from our code base to produce the probability for each node type in the whole vocabulary to be a child of any given node type. The size of the vocabulary of node types for a programming

language is rather small. Based on the unified grammar in SrcML, we estimate that the size is below 450, even when all AST node types from C/C++, C#, Objective-C, and Java are combined. After training the *AST2vec* for all node types, we obtain a numerical vector, i.e., an embedding, for each node type, and use the vectors to start the training of DTBCNN and Bi-DTBCNN models.

3.2.3 Gated Graph Neural Networks (GGNN)

GGNN and Gated Graph Sequence Neural Networks have been proposed as a general way to learn graphs [69]. Allamanis et al. [5] custom-build program graphs to encode both syntax and semantics of C# source code and extend GGNN to learn the graphs for several software maintenance tasks such as predicting misused variable names. Although our algorithm classification tasks are different from those published in previous GGNN-based tasks, we adopt the same schema for encoding the code as graphs, as shown in [5] that the graphs may encode more code semantics and graph-based neural networks may produce better code representations .

Each sample (i.e., program compilation unit) is represented by a graph as follows. The AST are encoded as a set of edges representing `Child` relations, whilst the ordering of children are kept by the `NextSibling` relations. Since our classification task does not care about the exact names of the identifiers compared to the variable misuse prediction task in [5], we can further reduce the number of nodes in the graph.

On the semantic side, the 'def-use' relations we obtained from program slicing are encoded directly into edge types `LastWrite` and `LastUse`. Following the schema used in [5], the 'returns' statements are recorded as a special relation `ReturnsTo`. Similarly, the `LastUse` relations are inferred from the otherwise discarded variable names, and the `ComputeFrom` relations are derived from the variables used in right-hand side and left-hand side of assignment expression AST. In this encoding, these semantic relations are rather agnostic to the concrete language syntax. It is therefore our hope that the graph representation can capture more commonalities between the structures. Compared to TBCNN encodings, such semantic edges are explicitly identified by static analysis tools, instead of learnt by the NN from the extracted features.

In this work, we have faithfully used the suggested approach in the original GGNN work [69] to aggregate the node-level embeddings learnt from graph propagation to the graph-level. Even though the schema from early work is adopted [5], there are still many configuration in the encoding to adjust,

e.g., whether or not to encode the backward edges for semantic relations, and for our algorithm classfication tasks the best configuration needs to be found empirically. [1]

3.2.4 Token and Sequence based Neural Networks

There are also many other techniques based on tokens or token sequences or others to learn code representations [2], many of which have originated from natural language processing (NLP). Most of the techniques have the common underlying idea that whatever code representations can be turned into feature vectors which can then be classified through various machine learning models. This common underlying idea also aligns well with the Bi-NN framework, and we can instantiate the framework with those techniques too.

Particularly, we instantiate the Bi-NN framework with the following commonly used models for our evaluation later.

BoW: The bag-of-words (BoW [65]), a.k.a. vector space model, counts the occurrences of each token as a feature to generate feature vectors for input. We adapt the model to generate vectors for source code.

N-gram: The BoW model does not consider the ordering among words, while n-gram models partially consider the ordering by using the count of n consecutive tokens in the input as a feature to generate feature vectors. Hellendoorn and Devanbu [51] found that a n-gram model (where n can be 3 or 5) can be a strong baseline for modeling large amount of source code. Notice that BoW can be seen as a special case of n-gram (i.e., unigram) models.

Tf-idf: The term frequency–inverse document frequency weighting scheme gives different weights to tokens of different occurrence frequencies: tokens appearing more frequently in one input are given higher weights while tokens appearing more frequently cross different inputs are given lower weights. As indicated in [117], such feature vectors may be better in identifying features that are more discriminative across programs, e.g., the token "bfs" for programs involving breadth-first searches.

LSTM-based: The above language models generate feature vectors by simply counting consecutive tokens, which may miss relations among separated

[1] The implementation of GGNN: `https://github.com/bdqnghi/ggnn_graph_classification`

tokens. Some neural network (NN) based models, such as long short-term memory (LSTM) [57], can take as input word embeddings, instead of token counts, to learn sequences better. In the literature, Siamese-LSTM [84] has been shown to achieve good performance for matching sentences in different natural languages. Therefore, we instantiate the Bi-NN framework with LSTMs to construct a Siamese-LSTM for comparison too. In our case, we use word2vec [80] to train the embedding vectors for tokens in C++ and Java corpus respectively. Then, the word2vec vectors for C++ tokens and Java tokens are fed as input into each of the two sub-LSTMs to train the whole Siamese-LSTM.

We have implemented various preprocessing steps to normalize tokens in source code as we know that not all tokens in programs are useful for determining code semantics. Sample preprocessing steps are removal of punctuation characters, identifier splitting based on CamelCase and underscores, converting all tokens into lower cases, and replacing single-letter variable names with a unified "id", etc.

For the classification model on top, we use fully-connected neuron layers with a Softmax output for the Siamese-LSTM. The Softmax classifier, which can be seen as the Multinominal Logistic Regression, is a classifier for multi-class classification. The Softmax layers can also be used as the classifier for the BoW, n-gram, tf-idf models where the feature vectors are based on token counting. Although we can also use any other classifier, such as SVM and Random Forests, instead of the Softmax layers, we use Softmax consistently for easier comparison and leave evaluation on the effectiveness of different classifiers to future work.

3.3 Empirical Evaluation

3.3.1 Datasets

We use two datasets for evaluation. The first dataset inherits from the TBCNN work by Mou et al. [81], let's call this dataset as Dataset A. This dataset includes samples of 104 programming problems used in university programming lectures, and each class comprises of 500 different C++ programs. The total of 52,000 samples in this dataset is used to evaluate whether our implementation of TBCNN can correctly reproduce the same level of performance compared to [81].

For cross-language algorithm classification, however, the first dataset is insufficient because it only has C++ samples. Therefore, the second dataset is crawled from the GitHub, which contains 50 distinct algorithms. We use GitHub Developer APIs to retrieve the algorithm instances for each given algorithm class in each programming language. We use the name of the algorithms as the keywords to search for the files whose names or contents contain such keywords.[2] For the evaluation purpose, we collect the same algorithms in both C++ and Java. To prevent from getting toy examples, we only retrieve the files of a size larger than 500 bytes.

The raw samples from GitHub, however, contain clones which may affect the training performance. To reduce the impact of duplicated data on classification results, we use NiCad [34] to detect clones among the data, and remove all of the Type 1, Type 2 and Type 3 clones with a dissimilarity lower than 10%. After the clone removal, we obtained 4,932 algorithm files in Java and 4,732 for C++. On average, each class contains from 90 to 100 programs.

For the training and testing purpose, we divide either the C++ or Java dataset into the training set and testing set with a split ratio of 80/20 for the programs per class, and use 80% of the data for training and 20% of the data for testing. To form the pairwise data for the cross-language settings, we take each program in one language and pair it up with another program in the other language. Given all the combinations of the C++ and Java programs in the training set, we have about 312K pairs of bilateral programs implementing the same algorithm in different languages, and more than 15 millions of pairs of programs that implement different algorithms. Because of the imbalance, we only randomly select 312K pairs that implement different algorithms to balance the training and testing data for binary classification.

3.3.2 Implementation and Research Questions

We have implemented multiple instantiations of the Bi-NN framework (cf. Section 3.2) for evaluation and comparison:

- BoW model,
- 3-gram model,
- 5-gram model,

[2] A detailed list of the algorithms can be found here: https://github.com/bdqnghi/bi-tbcnn/blob/master/algorithms.txt

- Tf-idf model,
- Siamese-LSTM model,
- Bi-TBCNN model,
- Bi-DTBCNN model, and
- Bi-GGNN model.

We adapt srcML to get Java and C++ programs into AST, and we annotate the AST using def-use relations extracted by `srcSlice` [8]. From these AST and the def-use relations we build dependency trees and derive GGNN-compatible graphs. We use `Tensorflow`[3] to build our Bi-NN. For the hidden layers, we add dropout with the probability of 0.7 to prevent the models from over-fitting. We use leaky `ReLU` as the activation function of the hidden layers. The GGNN encoding is implemented on the basis of the schema provided by Miltiadis et al.[5] and adapted by a preprocessing step to convert node tokens into node types, then the GGNN implementation is used to train/test the converted code graphs. Our techniques are implemented in a mix of Python and Bash scripts [4].

3.3.2.1 Research Questions

For our study on algorithm classification, we measure the effectiveness of each model by using the usual **accuracy** metric for the classification results. I.e., for a given set of test inputs, the accuracy of a model is the percentage of the tests for which the model produces a correct classification according to the ground-truth labels of the tests. Two kinds of classifications are considered: binary classification for determining whether or not two programs in two different languages implement the same algorithm (cf. Section 3.1.1), and multi-class classification for determining which algorithm class a given program implements (cf. Section 3.1.3).

We aim to compare the models in various settings against each other by answering the following research questions:

RQ1 Which instantiation of the Bi-NN framework achieves the best classification accuracy?

[3]https://github.com/tensorflow/tensorflow

[4]The code and evaluation results are available at `https://github.com/bdqnghi/bi-tbcnn`

RQ2 Does this best instantiation for cross-language classification achieve better classification accuracy than others in single-language settings too?

RQ3 How sensitive is this best instantiation when the number of classes varies?

RQ4 Does adding dependencies in the code representations achieve better classification accuracy?

We run our evaluations on a server machine with an Intel Xeon CPU E5-2640 v4 and an Nvidia P100 GPU with 12GB of memory and 4.7 TeraFLOPS double-precision performance. The server is shared among multiple users and its workload may affect the time measurements of the evaluations.

3.3.3 Summary of Classification Results

We provide summarized answers to the research questions here and present more details in later subsections.

RQ1 To classify programs across languages, we found that all the statistical language models, such as tf-idf, bag-of-words, n-gram, and LSTM can be employed, but with different effectiveness. Amongst various instances of Bi-NN models, our Bi-DTBCNN model achieves a better cross-language classification accuracy of 86% than others ranging from 46% to 77%, but at the price of being the slowest in training (see Table 4.2).

RQ2 All the models can be adjusted to recognize algorithm classes in a single language (SL) too (cf. Section 3.1.3). Our DTBCNN models achieves an accuracy of 91% and 93% for Java and C++ respectively, the highest among other models and much better than the cross-language (CL) settings (see Table 3.2).

RQ3 The number of algorithm classes has varying effects on different learning models. In both SL and CL settings, DTBCNN models maintain relatively good accuracy above 93% when the number of classes increases, whilst GGNN performance is much more sensitive and its accuracy decreases from 94% to 66% when the number of classes increases from 10 to 50 (see Table 3.3).

RQ4 DTBCNN with dependencies achieves significantly better accuracy than TBCNN for the binary cross-language classification task, improving it from 77% to 86% for the Github dataset (see Table 3.4). For the single language

Table 3.1: Results for cross-language binary classification by different code learning techniques.

Model	Accuracy	Training Time(hm)
Bag of words	0.46	5m
3-grams	0.51	5m
5-grams	0.52	5m
Tf-idf	0.49	7m
Siamese-LSTM	0.73	1h50m
Bi-TBCNN	0.77	3h10m
Bi-GGNN	0.76	5h50m
Bi-DTBCNN	0.86	8h25m

classification task, DTBCNN achieves comparable accuracy to TBCNN (see Table 3.2), which may imply that making implicit dependencies explicit in a single-language may not be necessary.

3.3.4 Details of Classification Results

RQ1: Results on Different Code Learning Techniques

Table 4.2 shows the results of various models for the binary cross-language algorithm classification task. The training time in Table 4.2 is measured as wall clock time by taking the average of 3 separate runs per configuration. The termination condition for each training depends on whether the loss function of a model reaches a certain threshold or the number of iterations/epochs in training exceeds a certain limit. The time needed to classify a test program is typically very short within a second.

The results show that NN-based models perform significantly better than the other token-counting based models (BoW, n-gram, and tf-idf). Our Bi-DTBCNN achieves the highest accuracy of 86%, but it takes the longest training time. GGNN training is faster than DTBCNN because both gated graphs and dependency trees are extensions of ASTs and our custom-built dependency trees in fact have more nodes than gated graphs although gated graphs may have more edges. Optimizing tree or graph representations of code and the training algorithms can be useful future research for better code learning.

RQ2: Results on Single-Language Classification

For each of TBCNN, DTBCNN and GGNN, we train a single-language model using the Java corpus and another single-language model using the C++

Table 3.2: Single-language algorithm classification results

Model	Dataset		
	Github		A
	C++	Java	C++
TBCNN, Mou et al. [81]	0.93	0.89	0.93
GGNN, Allamanis et al. [5]	0.66	0.56	0.49
DTBCNN	0.93	0.91	0.93

corpus, and compare their classification accuracies. Table 3.2 shows the results. DTBCNN models slightly improve the classification accuracies of TBCNN models [81], which are also higher than GGNN models.

RQ3: Results on Sensitivity Analysis wrt Numbers of Classes

We hypothesize that the number of algorithm classes would affect the performance of all the code learning techniques. In particular, it would be interesting to find out whether the numbers of classes affect GGNN more than tree-based models.

Thus we perform the sensitivity analysis by reducing the number of classes to see how it affects the performance of the models in both single- and cross-language settings. As shown by the results in both SL and CL settings in Table 3.3, DTBCNN models maintain relatively good performance when the number of classes increases, whilst GGNN performance is more sensitive to the number of classes. For SL settings, both GGNN and TBCNN perform well with 10 classes, reaching around 90% accuracy or better for both Github Dataset and Dataset A. However, when the number of classes increases, the GGNN cannot keep up the good performance but reduces the accuracy drastically, e.g 66% for Github C++ Dataset with 50 classes and 45% for Dataset A with 50 classes. On the contrary, DTBCNN can still maintain superior performance around 90%+ accuracy. The same situation occurs for cross-language (CL) settings: both Bi-GGNN and Bi-DTBCNN perform well when there are 10 classes in sub-components, but when the number of classes increases, the performance of Bi-GGNN drops and is more volatile than that of Bi-DTBCNN.

For SL settings, both GGNN and TBCNN perform well with 10 classes, reaching above 90% accuracy for both Github Dataset and Dataset A. When the number of classes increases to 50 for Github Dataset and 50 or 104 for Dataset A, the accuracy of GGNN reduces drastically, e.g., 66% for Github C++ Dataset with 50 classes and 45% for Dataset A with 104 classes. On the contrary, DTBCNN can maintain its performance above 93%. The same

Table 3.3: Sensitivity Analysis

Model	Dataset	Num Classes	Setting		
			SL		CL
			C++	Java	
GGNN	Github	50	0.66	0.56	0.76
		30	0.93	0.88	0.73
		10	0.94	0.94	0.88
	A	104	0.45	–	–
		50	0.48	–	–
		25	0.68	–	–
		10	0.89	–	–
DTBCNN	Github	50	0.93	0.91	0.86
		30	0.95	0.93	0.88
		10	0.96	0.95	0.88
	A	104	0.94	–	–
		50	0.95	–	–
		25	0.95	–	–
		10	0.98	–	–

Table 3.4: Results of cross-language algorithm classification with different dependency trees.

	Plain AST	AST+Def	AST+Def+Use
Accuracy	0.77	0.83	0.86

situation occurs for cross-language (CL) settings: both Bi-GGNN and Bi-DTBCNN perform well for 10 algorithm classes; when the number of classes increases, the performance of Bi-GGNN drops and is more volatile for different runs than that of Bi-DTBCNN.

RQ4: Results on Using Dependencies in the Models

A major intuition for adding dependencies into tree or graph based code representations is to expose more code semantics to help machine learning techniques to learn better. GGNN [5] is built on such an intuition to add many different kinds of edges into ASTs to form gated graphs, and it is shown to be useful for predicting variable names. For DTBCNN, we also want to find out whether adding dependencies into ASTs contaminates the code representations to make it harder to learn and how much effect adding dependencies has on the classification accuracy. As shown in Table 3.4, a neural network learning model trained on plain AST (i.e., TBCNN) produces worse cross-language classification accuracy than the same model trained on ASTs embedded with def or def-use relations (77% vs. 83% vs. 86%). On the other hand, for the single-language setting, TBCNN produces closely comparable accuracies to DTBCNN (see Table 3.2). This phenomena may indicate that in a cross-language setting, the code syntax can differ a lot between different languages

and require additional dependencies to help learn better representations for cross-language classification; while in a single-language setting, most code semantics are expressed in code syntax already, and the usefulness of extra dependencies may be reduced. It would be interesting future study to investigate further what dependencies may be useful for what kinds of tasks.

3.3.5 Threats to Validity and Discussions

We discuss several threats to the validity of our study, and discuss possible alternatives that can be done in future studies.

3.3.5.1 Threats to Validity

Data collections. Using Github Search API, we collected the code implementation of algorithms based on some specific keywords to identify the name of algorithms, such as “bfs”, “bubblesort”, “linkedlist”, etc. This approach may find some source code that are not actually related to the algorithm (i.e., false positives). To reduce the impact of such cases, we have to restrict the size of crawled code file to e.g. 500 bytes in order to exclude code files associated with auxiliary library code or details irrelevant to the algorithms. However, it is also possible that we have excluded many useful implementation of algorithms (i.e., false negatives). However, the quality of the Github search is an uncontrolled variable for the experiments, even though the authors have randomly inspected 200 returned results to find the false positive acceptable. Moreover, these samples from Github do not nessessarily compile, hence we choose srcML parser to generate AST and slicing information, instead of the production compilers from JDK (for Java) and clang (for C++).

Merging Layers. We used a subnetwork merging strategy and a softmax layer to classify programs, in either SL or CL settings. The merge can also be done using energy functions such as Manhattan euclidean distances in Siamese-LSTM [84] or using multi-layered perceptrons to fuse the vectors. We leave it for future work to explore the effectiveness of different alternatives for merging the subnetworks.

Node granularity: In our implementation of GGNN, TBCNN, and DT-BCNN, we model source code using the AST node types mostly (ignoring identifier names), which are less fine-grained than other models that consider concrete tokens. Despite being ‘coarser’, such node-type level encodings outperforms token-based LSTM. A possible explanation is that node types can

already keep structural and semantic features of source code relevant to an algorithm, without losing critical information. On the other hand, concrete token information can be useful for certain learning tasks, such as predicting wrong variable names [5]. A future improvement is to combine both the node type level and token level information and see how it will improve the performance of code learning models.

3.3.5.2 Justification for baseline results

As described by Hellendoorn and Devanbu [51], the n-gram model, if configured carefully, can achieve a comparable result to a neural network-based model. However, our results in Table 4.2 show that all of the token feature-based models, include the n-gram, underperform neural network-based models.

We would like to find the reason for the worse performance in feature-based models. As described in [51], the code model is built *per project*, that is, an n-gram sequence can be reused *across files in a project*, while in our work, each file in the corpus is an *isolated program* and all the files have no explicit connection to each other. This makes the token features extracted by n-gram, bag of words and tf-idf sparse, except for some common language keywords (if, else, include, etc) or common variable declarations (i, j, str, etc). In short, the feature-based models do not capture well the relations among the tokens well in our dataset.

In contrast, NN-based models (LSTM, CNN) take the input as the pretrained embedding of words. These pretrained embeddings capture the relations among tokens into a low dimensional continuous vector space. Thus, the NN-based models can produce better vector representations for the token features, which leads to better classification results.

3.3.5.3 Comparison between neural-network based models

Among the NN-based models, our custom-built DTBCNN achieves higher accuracy than the others. Here we want to provide some justifications for such results. The NN-based models considered can be categorized into two types: sequence-based (LSTM and CNN) and structure-based (GGNN, TBCNN and DTBCNN). The sequence-based models consider the source code at the token level, while the structure-based model considers the source code at the structure level (e.g., AST node types and dependencies). The advantage of treating source code as token sequences is that it is simpler to adapt well-known NLP techniques. One disadvantage of such techniques is that they cannot

make use of inherent features that hide inside the implicit structure of source code, such as data and control dependency, class inheritance, call relations, etc. Another disadvantage is that the number of tokens can be arbitrary as developers can introduce new tokens when writing code [117], making it harder to capture the relations among essentially the same tokens that appear differently.

On the other hand, AST or graph representation of source code are closer to the structural nature of algorithms. Since our goal is to classify algorithms *across different languages*, the inherent features of code structures can be more important than those specific tokens to distinguish the program.

For example, if we consider control dependency, the bubblesort makes multiple passes through a list of items. The code structure usually contains 2 nested `for` loop because we want to compare each item with every other item at least once, we also need an `if` inside the second `for` loop to check if the previous item is bigger than the current item for swapping. In addition, if we consider data dependency, the bubblesort involves comparison and swapping of items in a list and does not need to introduce many variable declarations since it is an in-place algorithm. All of these, unlike token sequences, can be features of bubblesort to distinguish itself from others, no matter in which language the code is written. In addition, we consider only the node types of AST instead of actual tokens, making the embedding vocabulary smaller to generate more compact vector representations.

In addition, as mentioned in Section 3.2.3, encoding a program as the graph intuitively adds richer semantic information of the program to the AST, thus is expected to yield a better result. However, our results shown in Table 4.2 and Table 3.4 are against this intuition. A possible reason is that graphs encode both control and data dependencies as edges between the nodes in the AST, thus complicating the structure of the AST, while our approach adds richer semantic information but remains to be tree-based. Graph-based similarity comparison boils down to graph isomorphism, which is a much harder problem than tree-based comparison.

In the future, we will conduct more experiments to observe the internal representations of the neural networks (NNs) with more datasets, in order to really understand what the NNs have learned to represent the programs.

The approach proposed is general to any pair of programming languages, although we only used Java and C++ to evaluate in this work. When the programming languages are cross paradigms, such as object-oriented versus

functional, it would be interesting to see whether the framework needs any further customizations, we leave this task for the future. More algorithms e.g., those listed in Rosatta Code can be added to the dataset in future, however, the 50 algorithms are already sufficient for us to assess multiple baselines and challenge the scalability of their training.

3.4 Conclusions and Future Work

In this work, we generalize a Bilateral Neural Network (Bi-NN) framework for cross-language algorithm classification problems. We instantiate this framework with different intermediate representations of 'Big Code' learning, including our own dependency tree-based convolutional neural networks (DTBCNN), and evaluate them on the tasks of classifying thousands of programs files as 50 algorithms, across different programming languages such as Java and C++.

We introduce DTBCNN to encode def-use relations (aka program dependencies) as part of abstract syntax trees, which can achieve the highest classification accuracy compared to other commonly used models (e.g., bags-of-words, n-gram, tf-idf, long short-term memory, gated graph neural networks).

Chapter 4

Learning Cross Language API Mapping with Little Knowledge

Migrating software projects from one language to another is a common and important task in software engineering. To support the process, various migration tools have been proposed. A fundamental challenge faced by such tools is to translate the library APIs of one language to functionally equivalent counterparts of another. Often, much manual effort is required to define the mappings between the respective APIs of two languages.

Several studies have addressed this API mapping problem, such as MAM [138], StaMiner [89], DeepAM [49], and Api2Api [94]. MAM [138] and StaMiner [89] require as input a large body of parallel program corpora, which contain functionally equivalent code that use APIs in both languages, in order to mine the mappings. Thus, they rely heavily on the availability of bilingual projects that implement the same functionality in two or more languages, which is not easy to find for any pair of languages. Although they rely on similar function names to reduce manual effort needed to identify parallel data, many functions with similar names may be actually functionally different, degrading the quality of training data and final mapping results. DeepAM [49] maps API sequences to sequences based on the text descriptions for the sequences. Its intuition is that two API sequences across languages may be mapped to each other if their text descriptions are similar. This approach does not need API mapping seeds, but requires many similar text descriptions across programs written in different programming languages whose availability can affect the mapping results. Api2Api [94] uses a vector space transformation method inspired by Mikolov et al. [79], but it still requires many API mapping seeds from an external source (Java2CSharp [29])) to map APIs across languages.

We propose an approach that can map APIs across languages while alleviating the shortcoming of existing approaches. We realize that the underlying goal of state-of-the-art techniques is essentially to find a transformation that can align two different domains (in our context, the two vector spaces for APIs in two different languages). Api2Api [94] is also an instance of this idea to learn an optimal transformation matrix between two vector spaces while requiring much parallel training data. However, empirical evidence of existing approaches suggest that collecting the training data is an expensive process that requires either availability of manual inspection or high-quality documentations. This has led to the following research question we aim to answer in chapter: "*Can a model be built to minimize the need of parallel data to map APIs across languages?*".

We realize that the API mapping problem may be addressed by techniques based on generative adversarial training [47] with the assistance of a pre-trained model. Given large code bases in two languages, it is likely that certain similarities between the code bases can be exploited to discover APIs of similar functionality across languages, without manually specifying parallel corpora. Such knowledge of similar functionalities may not be big enough for a complete mapping model, but it is small enough to afford human validation. Once validated, the knowledge can be *transferred* through *adversarial training* techniques to maximize the alignment between the two languages which results in better API mappings.

Our approach for API mapping works in the following way: (1) it takes in a large number of programs in two languages, and generates a vector space representing code and APIs in each language via a word embedding technique adapted from previous studies [22, 49, 89, 94, 138]; (2) it adapts domain adaption techniques [33, 43, 47] to transform and align the two vector spaces for the two languages, with mainly three technical components: Seeding, Adversarial training, and Refinement; and (3) it utilizes nearest-neighbors queries in the aligned vector spaces to identify the mapping result of each API. We name our approach **SAR**, after the three main technical components in the domain adaption step.

We have implemented the approach in a prototype tailored for Java and C#, and evaluated and compared it with the state-of-the-art techniques, such as StaMiner [89], DeepAM [49] and Api2Api [94]. We have evaluated the prototype on a dataset of more than 14,800 Java projects containing approximately 2.1 million files and 7,800 C# projects containing approximately 958,000 files.

Our evaluation results indicate that the approach can achieve 54% and 82% accuracy in its top-1 and top-10 API mapping results with only 174 automatically identified seeds, more accurate than other approaches using the same or much more mapping seeds. In addition, we also identify about 400 more API mappings between the Java and C# SDKs than other approaches.

The main contributions are as follows:

- We propose SAR, a new approach based on domain adaption techniques to transform and align different vector spaces across languages with the assistance of a seeding, adversarial learning, and refinement method. To the best of our knowledge, we are the first to apply the adversarial training techniques for the API mapping task.
- We adapt the adversarial training techniques in a number of ways to improve its alignment of the vector spaces: (1) we use nearest-neighbor queries to identify possible mapping candidates for better alignment; (2) we use a similarity-based model selection criteria and reduce the need of known API mappings during the training of our model; and (3) we use the Procrustes algorithm to find the exact solution of the mapping matrix.
- We have implemented the approach and evaluated it with a corpus containing millions of Java and C# source files; via an extensive empirical evaluation on different components of our approach, we demonstrate its advantages against other API mapping approaches in producing more accurate mappings with much fewer seeds that can be automatically identified.

4.1 Background

The goal of domain adaptation is to produce a mapping matrix as an approximation of the similarities between vectors in the two spaces. This section gives a brief overview of two methods for domain adaptation: seed-based (supervised) or unsupervised. Apart from the two input vector spaces, the seed-based method also requires a set of seeds as the parallel training data to learn the matrix, while unsupervised method does not: the mapping matrix can be obtained through adversarial learning assuming that similarity exists between the distributions of vectors in the two spaces.

4.1.1 Seed-based Domain Adaptation

Given two sets of embeddings have been trained independently on monolingual data, seed-based domain adaptation is to learn a mapping using the seeds *s.t.* their translations are close in a shared vector space. Such an idea has been explored for word translation in NLP [79], and Api2Api [94] adapts it to learn API mappings.

Formally, given two vector spaces, $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, containing n and m embeddings for two languages L_1 and L_2, and a set S of seeds of API embedding pairs $\{(x_{s_i}, y_{s_i})\}_{s_i \in \{1,|S|\}}$, we want to learn a linear mapping W between the source and the target space, such that Wx_{s_i} approximates y_{s_i}. In theory, W can be learned by solving the following objective function:

$$W^* \triangleq \underset{W \in M \subset \mathcal{R}^{d \times d}}{argmin} ||WX_S - Y_S|| \tag{4.1}$$

where d is the dimension of the embeddings; $M \subset \mathcal{R}^{d \times d}$ is the space of $d \times d$ matrices of real numbers; $X_S \triangleq \{x_{s_i}\} \subset X$ and $Y_S \triangleq \{y_{s_i}\} \subset Y$ contain the embeddings of the APIs in the seeds, which are matrices of size $d \times |S|$.

Instead of approximating a solution using traditional stochastic gradient descent method used in Api2Api [94], there exists an analytical Procrustes problem [112] solved by Xing et al. [122], which has a closed form solution of the mapping matrix derived from the singular value decomposition (SVD) of YX^T:

$$W^* \triangleq \underset{W}{argmin} ||WX_s - Y_s|| = UV^T, with\ U\Sigma V^T = SVD(Y_s X_s^T) \tag{4.2}$$

The advantage of a closed form solution is that one can get the exact solution which is better than the approximate solution of gradient descent, and is faster in computation.

With the mapping matrix W, one can use $y_x = Wx$ to map a query vector x. The vector y_x is the mapping, or adaptation, of x in the target space.

4.1.2 Unsupervised Domain Adaptation

Adversarial learning has been successfully used for domain adaptation in an unsupervised manner. In particular, the Generative Adversarial Network [47] achieves this goal by a model which comprises a generator and a discriminator as two inter-playing components. A generator network that aims to learn real

data distribution and produce fake data to fool the other component, so-called the discriminator; the discriminator network that acts as a classifier, which aims to distinguish the generated fake data from the real data. The two components are trained in a minimax fashion and would converge when the generator has maximized its ability to generate fake data so similar to the real data that the probability for the discriminator to make a mistake would be $\frac{1}{2}$.

Conneau et al. [33] use this idea as a variant for the machine translation task, which achieves significantly better results than other baselines of machine translation, which would require no parallel data to train the networks. The generator, in this case, is a mapping matrix W, which can simply be seen as a set of parameters that need to be learned, and the discriminator is a feed-forward neural network. We want to find a matrix W as an approximation of the mapping between the two vector spaces X and Y. In the adversarial learning setting, we aim to optimize two parameters: one is the discriminator's parameters, denoted as θ_D, the other is the mapping matrix W. Our goal is to find the optimal value of two sets of parameters, which results that we have two objective functions in the adversarial learning setting.

Discriminator objective Given the mapping W, the discriminator (parameterized as θ_D) is optimized by this objective function:

$$L_D(\theta_D|W) = -\sum_{i=1}^{n} logP_{\theta_D}(source = 1|Wx_i) - \sum_{i=1}^{m} logP_{\theta_D}(source = 0|y_i) \tag{4.3}$$

where $P_{\theta_D}\big(source = 1\big|v\big)$ is the probability that a vector v originates from the source embedding space (as opposed to an embedding from the target space).

Mapping objective Given the discriminator θ_D, the mapping W aims to fool the discriminator's ability of predicting the original domain of an embedding by minimizing this objective function:

$$L_W(W|\theta_D) = -\sum_{i=1}^{n} logP_{\theta_D}(source = 0|Wx_i) - \sum_{i=1}^{m} logP_{\theta_D}(source = 1|y_i) \tag{4.4}$$

Learning Algorithm The discriminator θ_D and the mapping W are optimized iteratively to minimize L_D and L_W, respectively by following the training procedure of adversarial networks proposed by Goodfellow et al. [47]

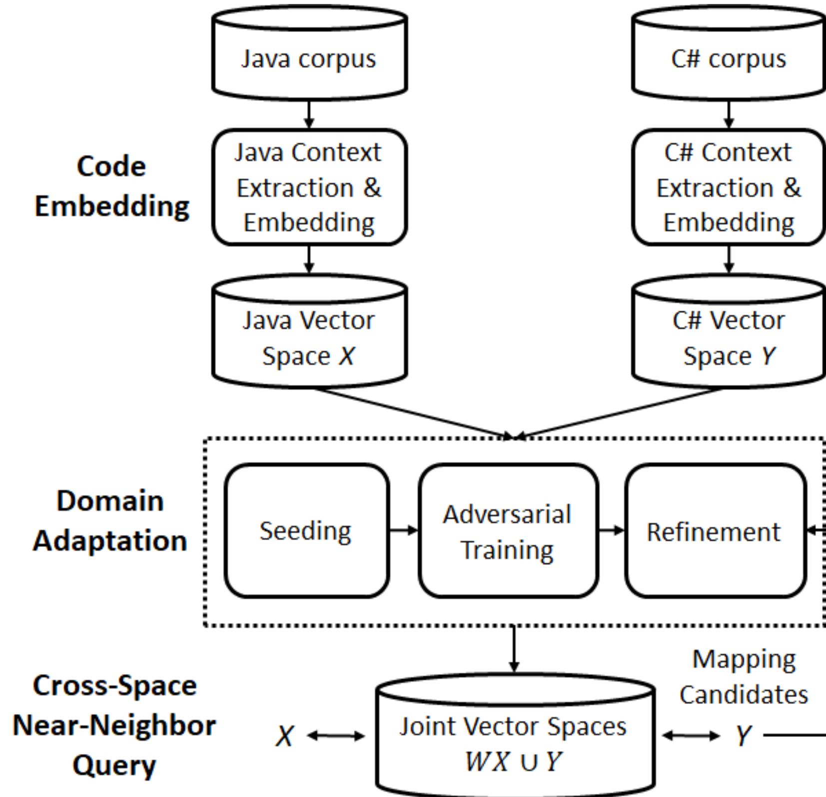


Figure 4.1: Overview of SAR: Domain Adaptation for API Mappings

4.2 Our Approach

Combining the virtues of seed-based and unsupervised adversarial methods described in the background, our domain adaptation approach can approximate two spaces of vectors with minimal parallel corpora. Although unsupervised adversarial learning method does not require any seed as parallel data, the distributions of vectors (i.e., embeddings) in the two spaces may not be similar. Therefore, it is our hypothesis that the performance could be improved by initializing the unsupervised adversarial learning method with a small set of seeds taken from the seed-based domain adaptation, and by generating the rest of API mappings in the two steps below:

- From large code corpora in two different languages, we create two vector spaces for APIs by adapting word embedding technique for code. From such corpora, we derive a small set of mappings based on a simple text similarity heuristic (see Code Embedding in Figure 6.1);
- The two vector spaces, along with the mapping seeds, are transformed by a mapping matrix to get aligned with each other. This step comprises three sub-steps: Seeding, Adversarial Learning, and Refinement (see Domain

Adaptation in Figure 6.1).

For any given API a in the source language and its continuous vector representation x, we can map it to the other domain space by computing $y_x = Wx$. Then, one can find the top-k nearest neighbors of y_x in the target vector space, using cosine similarity as the distance metric, and finally can retrieve the list of APIs in the target language that has the same embeddings as the top-k nearest neighbors. The list of APIs can then be used as the mapping results for a (see Cross-Space Near-Neighbor Query in Figure 6.1).

4.2.1 Code Embedding via Word Embedding

We first parse source code files into Abstract Syntax Trees (ASTs) using `fAST` [128] for both Java and C# projects. We convert each AST to a sequence of tokens by traversing the AST in its preorder. Through the traversal, we can identify the API token by checking the type of the node (e.g., function call nodes). We perform the normalization step to enrich the code sequence with structural information extracted from parsing, which constitutes two steps:

Filtering out unnecessary tokens: Once obtained the token sequence, we filter out tokens that are not necessary for our task, such as operators and primitive variable identifiers. Language keywords and AST node types are still kept for code embedding as they can enrich the structural information of the sequence.

Converting raw API tokens into signatures: This step reduces the variance of vocabulary existing in the source code. For example, one may extract the 'List.add' method from the 'java.util.List' class, or from the 'com.google.common.collect.List' in an external third-party library. Even though these two APIs have the same class and method names, their usages and semantics are different. To handle such cases, we propose this additional step to convert a raw API token to its signature in qualified name format 'Package.Class.Method'. The 'Package' is identified by using the 'import' statements (Java) or 'using' statements (C#).

Below shows an example of the normalization for the code token sequence:

```
float f List.add List.add if List.addAll else HashMap.put return
==> float java.util.List.add java.util.List.add if java.List.addAll
else java.util.HashMap.put return
```

From the corpora of code sequences, we use the skip-gram model [78] to train the embedding of tokens. Given a large corpus as the training data, the

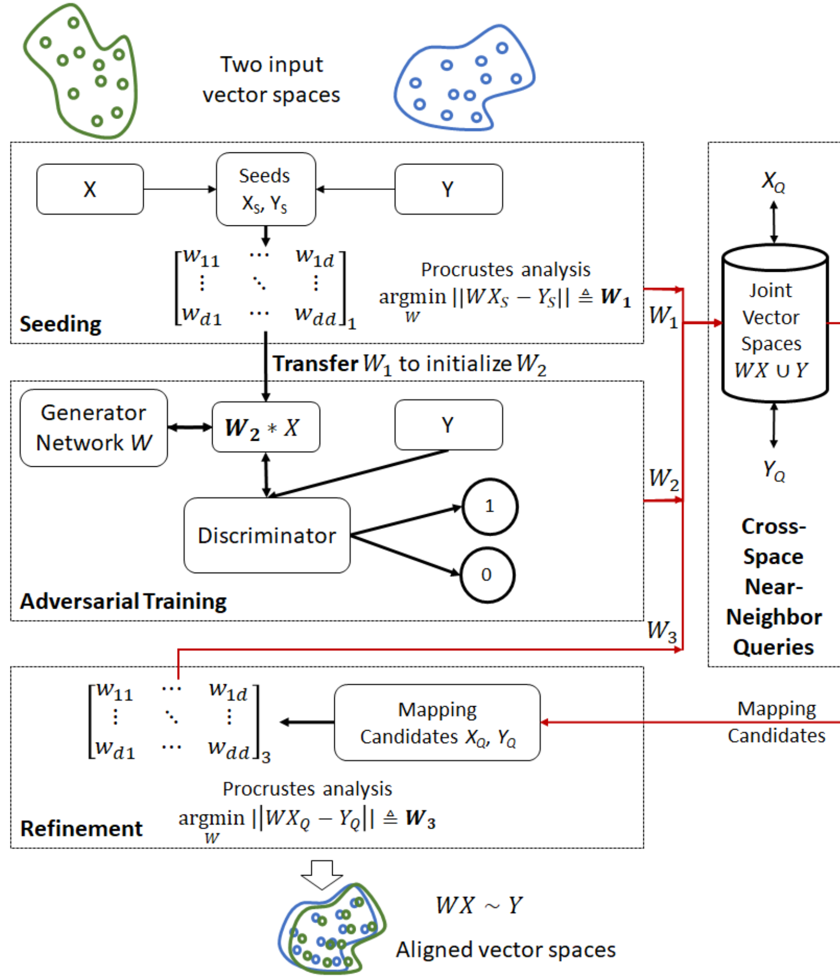


Figure 4.2: Domain adaptation steps to align two vector spaces

tokens appearing in the same context would usually have their embeddings close by distance in the vector space.

4.2.2 Domain Adaptation

Our domain adaptation comprises three steps: seeding, adversarial training, and refinement (hence the abbreviation SAR of our approach). Seeing SAR from outside as a black-box, it receives two vector spaces and a set of seeds as input and generates a mapping matrix W as output. Internally, each step of SAR is a different way to improve the mapping matrix, which receives the matrix output from the previous step as input and produces the improved version of it as output. We assign W_1, W_2 and W_3 as the output matrix for the three steps, respectively. Figure 4.2 summaries the domain adaptation procedure. The rationale for each step is described as follows: (1) **The Seeding** step to initialize a mapping matrix between the two vectors spaces based on some prior knowledge (i.e., seeds) (2) **The Adversarial Learning** step to re-use

the knowledge learned from the Seeding step as an initializer for adversarial training in order to maximize the similarity between the two vector spaces (or two distributions); and (3) **The Refinement** step to make the mapping matrix reach its optimal state.

4.2.2.1 Seeding

After Code Embedding, two vector spaces are obtained to produce a mapping matrix that approximates the two vector spaces by using the knowledge from mapping seeds in a dictionary. Notice that by simple signature-based comparison to identify APIs having the same signature name [1], one can identify many high-quality mapping candidates to be used as the seeds without any human effort to verify because developers often use the same name for the same functionality even when they are in different languages.

Having the set of seeds obtained, in addition to the two vector spaces X and Y, the initial mapping matrix produced is W_1 by solving the Equation 4.1 in Section 4.1 (also see Seeding in Figure 4.2). This seeding step can be seen as a function A, which will receive these three inputs and produces a transforming matrix W_1 such that $W_1 = A(X, Y, D)$. Internally, A solves the optimization problem described in Section 4.1 given the three inputs.

4.2.2.2 Adversarial Learning

The quality of the matrix W_1 learned in the previous step is limited by the number of seeds one can provide, which results in an approximation between the source and target domains. In this case, the knowledge learned for W_1 can be seen as a pre-trained model and can be reused for the other model.

Formally, given the two original vector spaces, $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, containing n and m API embeddings obtained from the Code Embedding step, we want to find the matrix W_2 to *maximize* the approximation of the mapping between the two vector spaces. We use adversarial learning to achieve this goal, which comprises of two steps: the mapping matrix W_2 and a discriminator network as described in Section 4.1.2. Our goal is to find the optimal value of W_2 and θ_D (discriminator parameters) We achieve this by training the adversarial network with the objective functions as described in Section 4.1.2 to find W_2 and θ_D.

[1]"Same signature" in our case means case-insensitive matches of the class and method names

The key difference with the general adversarial setting described Section 4.1.2 is that we do not initialize W_2 randomly as one usually does when training a neural network. Instead, we use W_1 as a pre-trained model to initialize for the W_2 so that W_2 is initialized with some good knowledge, even if it is small (see Adversarial Learning in Figure 4.2). This step is essential to improve the performance of the API mapping results.

Model Selection Criteria In short, this step is for choosing the optimal parameters for the Adversarial Learning step, although the heuristic used is similar to the Refinement step. To train the adversarial networks, like any other neural network architecture, we need a validation set to select the best model for the prediction step. The validation set is used to minimize over-fitting when training the neural network. Concretely, for each training epoch, one needs to evaluate against the validation dataset to pick the model that has the highest validation accuracy through training. Our goal is to use as little parallel data as possible to build the model. In practice, one only has a very small number of seeds inferred from the signature-based matching, or in the worst case, one cannot infer any seed to have data for validation. As such, it is impractical to use a parallel dataset as a validation set to train neural networks in the adversarial learning step, i.e., involving additional prior knowledge.

To address this issue, we perform a model selection using unsupervised criteria that quantify the closeness of the source and target embedding spaces. Specifically, we consider them as a set of K most frequent source APIs and multiply them with the mapping matrix W to generate a target mapping for each of them. After that, we get a set of mappings, then compute the average cosine similarity of these mappings and use the average as a validation metric.

4.2.2.3 Refinement for Better Alignment

The adversarial approach tries to align all words irrespective of their frequencies. However, rare tokens have embeddings that are less updated in the back-propagation step and are more likely to appear in different contexts in each corpus, which makes them harder to align [33]. To address this problem, we use the method proposed in [33] to infer a list of mapping candidates using only the most frequent tokens. Moreover, other heuristics are introduced to infer another candidate set of mapping based on the threshold of cosine similarity, which can be used as another synthetic dictionary that can combine with the top-K frequency mapping candidates.

Following the step shown in [33], it is possible to build a set of mapping candidates using W_2 just learned with adversarial training. Assume that one can induce a combined set of mapping candidates from different heuristics above, and the quality of the combined set is good, then this set of candidates should be used to learn a better mapping and, consequently, an even better set of candidates for the next iteration. The process can repeat iteratively to obtain a hopefully better mapping and candidates set each iteration until some convergence criteria are met. Formally, the refinement step receives W_2 from the previous adversarial learning step, along with the two original embeddings X and Y to produce the next W_3 iteratively (see Refinement in Figure 4.2).

Specifically, we produce the mapping candidates for refinement based on two heuristics:

Top-K Frequency: Conneau et al. [33] shows that by taking the top-k frequent words and their nearest neighbors in the transformed vector spaces, it can provide high-quality mapping candidates because the most frequently used words are likely to be the same across languages. Therefore, we can use the top-k frequent API names to induce the seeds for the refinement.

Cosine Similarity Threshold: Since finding API mappings in the aligned vector space is essential to finding APIs close enough in the vector space, all API pairs "similar enough" in the vector space aligned by Adversarial Learning can be good candidates for the refinement step. In this work, we use the cosine similarity as the metric to measure how similar two vectors are. We note that *not* all APIs in a language can have a mapping in another language. In the empirical case study, we show how a good threshold is found in Section 4.3.3.2.

Therefore, we can infer two sets of synthetic mapping candidates from the above heuristics. In fact, there are different ways to merge them into one single set as they can overlap as, e.g., (1) the union of the two sets, (2) the intersection of the two sets. In contrast to Conneau et al. [33], we use an additional Cosine Similarity Threshold heuristic to get a better set of mapping candidates.

The matrix W_3 in this step is the final output of the domain adaptation process. When it comes to the step to produce the mapping from the source query, the embeddings of the query will be multiplied with W_3 in order to obtain corresponding mappings in the target language.

Table 4.1: Example of Seeds from the Signature-based Matching Heuristic

Java	C#
java.lang.**String.equals**	System.**String.Equals**
java.util.**List.remove**	System.Collections.Generic.**List.Remove**
java.util.**Random.nextDouble**	System.**Random.NextDouble**
java.lang.**Math.round**	System.**Math.Round**
java.io.**File.Exists**	System.IO.**File.Exists**

4.3 Empirical Evaluation

We have conducted extensive empirical evaluations on our approach in various settings to answer the following research questions:

1. Compared to related methods, is our approach more effective in identifying API mappings?
2. How well do different combinations of refinement heuristics improve the performance?
3. What is the impact of each component in our approach on the performance?

4.3.1 Dataset

We use the Java Giga corpus data described by Allamanis et al. [7]. It involves approximately 14,807 Java projects from Github and contains approximately 2.1 millions of files. For C#, we clone the projects on Github that have at least 1 star and collect 7,841 C# projects with about 958,000 files. As the main advantage of our approach, there is no need to specify which code in Java is functionally equivalent to which code in C#. For each function in a file, we traverse the AST of the function to extract the API call sequences. For Java, we get a corpus containing 6.7 million code sequences; for C#, we get a corpus containing 5.1 million code sequences.

For evaluation, we take 860 method API mappings and 430 class API mappings defined in Java2CSharp [29] as the ground truth for evaluating our approach against the baselines.

4.3.2 Implementation

We adapt `Gensim` [110] in NLP to produce the embeddings of tokens for the Java and C# corpora. We use the same settings used by Mikolov et al. [80] during the training: stochastic gradient descent with a default learning rate of 0.025, negative sampling with 30 samples, skip-gram with a context window of size 10, and a sub-sampling rate of value $1e^{-4}$.

4.3.2.1 Evaluation Metrics

We use three metrics to measure the performance of our approach and the baselines.

Top-k Accuracy: The top-k accuracy is defined as follow: For a test JDK API j, SAR produces a resulting list. If the true mapping API in C# .NET for j is in the top-k resulting list, we count it a hit. If not, we count it a miss. Top-k accuracy is computed as the ratio between the number of hits and the total of hits and misses for a given ground-truth test set.

Mean Reciprocal Rank: For a test JDK API j as a query, SAR produces a resulting list, we calculate the Reciprocal Rank (RR) of that query. For all queries in our evaluation data, we calculate the Mean Reciprocal Rank (MRR) of the test set. MRR is the average of the reciprocal ranks of results for a sample of queries.

Formally, Reciprocal Rank can be defined as: $RR = \frac{1}{rank_i}$, where $rank_i$ refers to the rank position of the first relevant mapping for the i-th query. And the Mean Reciprocal Rank (MRR) can be defined as $MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} RR_i$, where RR_i refers to the Reciprocal Rank for the i-th query, $|Q|$ refers to the total number of queries.

F-score: The F-score is defined as $F = (2*P*R)/(P+R)$, where Precision $P = TP/(TP+FP)$ and Recall $R = TP/(TP+FN)$. TP refers to the number of true positives, which is the number of API mappings that are in both result dataset and the ground truth set; TN refers to the number of true negatives, which is the number of API mappings that are neither in the returned results nor in the ground truth set; FP refers to the number of false positives which represents the number of result mappings that are not in the ground truth set; FN refers to the number of false negatives, which represents the number of mappings in the ground truth set but not in the results.

4.3.2.2 Code Embedding

From the two code corpora, we scan through all pairs of APIs in the two corpora to produce a set of seeds using the signature-based matching heuristic. We got 257 seeds for this step. Table 4.1 shows examples of the seeds. Among these 257 seeds, we found that 83 seeds overlap in the 860 ground truth mappings. Note that for a fair comparison with the baselines (Api2Api, DeepAM, StaMiner), we remove these 83 overlapping seeds from the 257 seeds and we get a set of **174 seeds for the Seeding step**. Then, we apply word embedding on the corpora

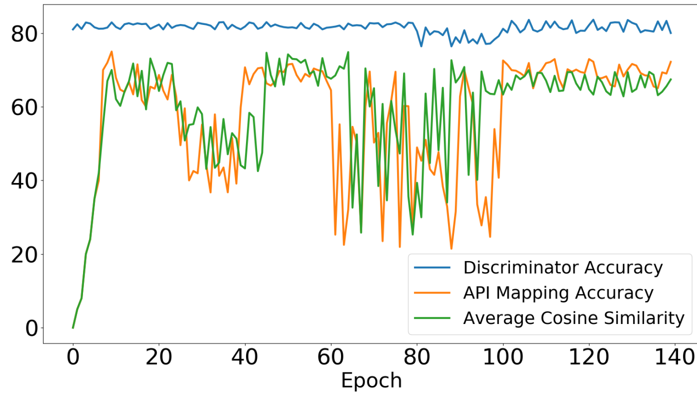


Figure 4.3: Unsupervised Model Selection Criteria

to get the source embedding and target embedding. We use the embeddings, along with the seeds as the input for the domain adaptation process.

4.3.2.3 Domain Adaptation

For the seeding step, we find W_1 by using the Procrustes solution in Equation 4.2 with three inputs: source embedding X (Java), target embedding Y (C#), and 174 mapping seeds. This step gives us the mapping matrix W_1. We implement the adversarial learning by using `PyTorch` [100]. We use Momentum Gradient Descent method [114] to search for the optimal transformation matrix[2].

We use the unsupervised model selection criteria proposed in Section 4.2.2.2 to select the best model by choosing the top 1000 frequent API token pairs, e.g., top-1 frequent token in the source is aligned with top-1 frequent token in the target as the validation set, then we extract the W_2 from the model. Figure 4.3 shows three different lines: (1) the discriminator accuracy, which is the accuracy in classifying the samples from the source and target embeddings, (2) the API mapping accuracy, which is the accuracy when using the model to evaluate against the 1000 pairs validation set, and (3) the average cosine similarity of all the pairs. As shown, the criteria correlate well to the mapping accuracy. The high instability is because of over-fitting. Thus, we only selected the model of the best validation accuracy.

From W_2 resulting from the adversarial training, we obtain the final W_3 by performing the refinement step on the basis of two heuristics in Section 4.2.2.3. For the top-N frequency heuristics, we choose top-500 frequent tokens for the synthetic dictionary, as suggested in [33]. For the second similarity threshold rule, we use 0.7 as the threshold as shown in Section 4.3.3.2, we found that this

[2]Our implementation (including source code and a docker image) can be accessed at the public repository: `https://github.com/bdqnghi/SAR_API_mapping`

Table 4.2: API Mapping Results

Index	Baselines	K-folds	Seeds	Top-1	Top-5	Top-10
1	Random seeds: Api2Api	-	0	0.03	0.05	0.1
2		-	10	0.09	0.12	0.14
3		-	50	0.14	0.19	0.22
4		-	100	0.19	0.24	0.32
5	Random seeds: SAR	-	0	0.25	0.30	0.35
6		-	10	0.28	0.35	0.40
7		-	50	0.26	0.43	0.47
8		-	100	0.44	0.50	0.69
9	K-Fold: Api2Api	1-fold	172	0.24	0.35	0.41
10		2-folds	344	0.34	0.45	0.55
11		3-folds	516	0.37	0.51	0.67
12		4-folds	688	0.43	0.64	0.72
13	K-Fold: SAR	1-fold	172	0.36	0.39	0.48
14		2-folds	344	0.45	0.50	0.61
15		3-folds	516	0.54	0.66	0.71
16		**4-folds**	**688**	**0.59**	**0.77**	**0.84**
17	Signature-based: Api2Api	-	25	0.12	0.16	0.18
18		-	50	0.20	0.23	0.29
19		-	100	0.27	0.32	0.38
20		-	174	0.31	0.41	0.60
21	Signature-based: SAR	-	25	0.30	0.32	0.39
22		-	50	0.35	0.39	0.45
23		-	100	0.41	0.50	0.63
24		**-**	**174**	**0.48**	**0.71**	**0.78**

Table 4.3: Accuracy of 1-1 Class-Level Mapping when compares with StaMiner and MAM

Package	Class Migration								
	Precision			Recall			F-Score		
	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR
java.io	70.0%	80.0%	80.0%	63.6%	75.0%	75.0%	66.6%	72.7%	77.5%
java.lang	82.5%	80.0%	82.5%	76.7%	81.3%	80.2%	79.5%	80.7%	82.6%
java.math	50.0%	66.7%	66.7%	50.0%	66.7%	66.7%	50.0%	66.7%	66.7%
java.net	100.0%	100.0%	100.0%	50.0%	100.0%	100.0%	66.7%	100.0%	100.0%
java.sql	100.0%	100.0%	100.0%	50.0%	100.0%	90.0%	66.7%	100.0%	95.0%
java.util	64.7%	69.6%	81.3%	71.0%	72.7%	71.0%	67.7%	71.1%	76.7%
All	77.9%	82.7%	**85.0%**	60.2%	**82.6%**	80.5%	66.2%	81.9%	**83.5%**

number balances coverage and precision of API mappings well.

4.3.3 Evaluation

4.3.3.1 RQ1. Effectiveness of SAR in Mining API Mapping

The first question we want to answer is how effective our approach in identifying API mappings from the two vector spaces. We compare SAR with Api2Api, StaMiner, and DeepAM.

Result Summary. Index 24 in Table 4.2 uses 174 API mappings automatically selected by the signature-based matching heuristic and test against the 860 ground truth mappings. Index 16 uses 688 mappings selected randomly from the 860 ground truth set and test against the rest. The performance of SAR in terms of top-k accuracy is shown. As one can see in both cases, the top-1 accuracies are above 50%, and the top-10 accuracies are above 80%.

Compare to Api2Api The method used in Api2Api is corresponding to the seeding step in our domain adaptation process, which finds a mapping

Table 4.4: Accuracy of 1-1 Method-Level Mapping when compares with StaMiner and MAM

Package	Method Migration								
	Precision			Recall			F-score		
	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR
java.io	70.0%	66.7%	66.7%	64.0%	87.5%	82.9%	66.9%	75.2%	74.8%
java.lang	86.7%	83.3%	81.5%	76.5%	87.2%	78.4%	81.3%	85.4%	84.4%
java.math	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%
java.net	100.0%	100.0%	80.0%	33.3%	100.0%	66.7%	50.0%	100.0%	81.7%
java.sql	100.0%	50.0%	70.0%	50.0%	66.7%	70.0%	66.7%	57.2%	70%
java.util	63.0%	64.3%	64.8%	54.8%	85.7%	85.0%	58.6%	73.5%	76.9%
All	**81.1%**	71.9%	71.7%	57.6%	**82.3%**	78.38%	65.0%	**76.3%**	75.5%

matrix by solving the Equation 4.1 given a large set of seeds. We use the top-k accuracy as the evaluation metric.

Table 4.2 shows the top-k accuracy of our approach when comparing to Api2Api in various settings. First, we compare Api2Api with SAR using the seeds coming from two different sources: the 860 mappings defined by Java2CSharp and the 174 mappings inferred from the signature-based matching. Here we described the variances as results shown in Table 4.2, indicating that our approach can use *much* fewer number of seeds compared to Api2Api but still achieve better results.

Select randomly: we select a subset of mappings r randomly from 860 mappings in the ground truth, and test against the rest $860 - r$ mappings. Concretely, $r = 0$, 10, 50, and 100;

k**-fold**: we divide the 860 mappings into $k = 1, 2, 3, 4$ folds and perform the variants of five-fold cross-validation: while k folds are used as training data, the other $5 - k$ folds are used as testing data;

Select by signature: we use 174 mappings inferred by method signature, and select randomly a varying number of them as the training data and test against the remaining mappings in the ground truth.

The process repeats for using different folds as the training data for both Api2Api and we take the average accuracy are some observations from the results:

- Using the same number of seeds, either using the seeds from Random, K-fold or Signature-based, we get significantly better results than Api2Api for every setting.

- When using all of the 174 signature-based seeds, our approach gets significantly better results than Api2Api: top-1 improves 17%, top-5 improves 30%, and top-10 improves 18%.

We also compare with Api2Api by using MRR as the evaluation metric. For a JDK API in the 860 ground truth mappings, we use it as a query for

SAR to produce a resulting list, then we calculate the RR for the first relevant mapping in the list. We do this for all of the JDK APIs in the 860 ground truth mappings and calculate the MRR of the test JDK APIs. We then do the same for Api2Api and get an MRR score. The MRRs for SAR and Api2Api are 0.67 and 0.43, respectively, which indicates that when given a query, SAR can retrieve the mapping results more accurately than Api2Api.

Compare to StaMiner and DeepAM We follow the details described in StaMiner and DeepAM to measure how well SAR performs in mining API mappings for Class API and Method API [3]. In Java, an API element, by definition, can be a class, a method or a field in the class; and it must belong to a package (or the namespace in case of C#). As such, the goal in this task is to measure the performance the Class and Method API mapping task one by one for each API of each package, i.e., to see which package has the best performance for API mappings, so-called 1-to-1 mappings. For the method API mapping, we use the 860 method ground truth mapping described in Section 4.3.1 for evaluation. For the class API mapping, we use the 430 class ground truth mapping described in Section 4.3.1 for evaluation. We follow the details described in DeepAM to choose only the APIs under the packages as shown in Table 4.3, column 'Package', so that the total number of method API mapping left is 289 (remaining from 860 ground truth method API mappings), and the total number of class API Mapping 283 (remaining from 430 ground truth class API mappings).

Adapting SAR for class-level API mapping is relatively easy: one can remove the method part of a qualified API signature token so that only the package and class parts of the token are retained in the code sequences. Then code embedding for the API sequence can be derived as the embedding of the class-level API, along with other keywords from the ASTs. We do this for both languages. To select mapping seeds by API signatures, we first infer the mappings from signatures at the class level, then follow a similar domain adaptation process from APIs at the method-level.

One could not run StaMiner and DeepAM directly because they require parallel data (aligned function body for StaMiner, and aligned code and text description for DeepAM) for training. Therefore, we had to compare to them by extracting the reported performance numbers from their papers. This is also how DeepAM compared itself to StaMiner. We use the F-score as the

[3]Note that we still use the model without the overlapping seeds

Table 4.5: Examples of newly found APIs in Java and C#

Java	C#
java.io.DataInputStream.readInt	System.Io.BinaryReader.ReadUInt16
java.lang.Byte.parseByte	System.sbyte.Parse
java.lang.Double.longBitsToDouble	System.BitConverter.Int64BitsToDouble
java.net.Datagramsocket.isConnected	System.Net.Sockets.Socket.Connect
java.awt.geom.AffineTransform.inverseTransform	System.Drawing.Drawing2d.Graphicspath.Transform
java.io.DataInputStream.readDouble	System.Io.BinaryReader.ReadDouble
java.net.Serversocket.accept	System.Net.Sockets.Socket.AcceptAsync

performance metric to measure accuracy in this evaluation

Table 4.3 and Table 4.4 shows the comparison results of our mined API mappings with StaMiner (Sta) and DeepAM (DeepA) at class level and method level, respectively. As one can see for the F-score, SAR produces better results than those of DeepAM and StaMiner at the class level. At method level, SAR produces better results than StaMiner, and is close to DeepAM in term of F-score. Note that while DeepAM needs to use millions of similar API sequence descriptions and StaMiner needs to use ten of thousands of pairs of parallel data, SAR only uses 174 pairs of mappings as a small set of parallel data for the Seeding step.

Newly found API mappings More interestingly, we found more new API mappings than other studies in our actual code corpora. For each of the API in Java, we query the top-10 nearest neighbors in C# and manually verify the mappings. We enforce the threshold = 0.7 as mentioned in Section 4.3.3.2 for this task. We found 420 new SDK API mappings that can complement the tool Java2CSharp. Comparing to MAM (25 new mappings), StaMiner (125 new mappings), Api2Api (52 new mappings), we found a sufficiently larger number of mappings and our newly found APIs also overlap with the APIs in these baselines. In Table 4.5, we show some interesting examples of such newly found API mappings whose name do not match exactly using traditional approaches. Our list of newly found Java/C# APIs mappings can be accessed at our Github repository.[4]

4.3.3.2 RQ2. Effect of Different Refinement Approaches

Effects of Cosine Similarity Threshold In this section, we measure the effect of different ways to combine the seeds for the refinement step. We want to measure the effects of cosine similarity threshold in order to choose a good one for the second heuristic in the refinement step. Since the threshold is a

[4] `https://github.com/bdqnghi/SAR_API_mapping/blob/master/new_found/new_found_apis.csv`

Table 4.6: Accuracy of the filtered mappings using various similarity thresholds

Threshold	Coverage		Accuracy	
	Top-1	Top-5	Top-1	Top-5
0.6	0.66	0.90	0.42	0.59
0.7	0.45	0.68	0.51	0.73
0.8	0.12	0.22	0.65	0.80
0.9	0.08	0.15	0.78	0.89

part of the refinement, the domain adaptation step only comprises of two steps: Seeding and Adversarial Learning. Once the threshold is found, we use it for the Refinement in the other experiments. Then we produce the mapping for each source query in the 860 ground truth mappings. For each mapping produce, we obtain the cosine similarity between the query and the result mapping. We choose a threshold to filter out the mapping that has the cosine similarity lower than the threshold, then we measure the accuracy of the filtered mappings.[5]

In Table 4.6, the column "Coverage" means the percentage of ground truth APIs that have mappings in the candidate selection results when choosing a specific cosine similarity threshold. The column "Accuracy" means the top-k accuracy in identifying the mapping given a cosine similarity threshold as a condition to identify. The results show that our approach in these experiments has higher mapping accuracy, but lower coverage with respect to the ground truth set when the similarity threshold increases. It is, therefore, a trade-off to have higher accuracy in the expense of coverage. For the other experiments that involve the cosine similarity threshold in the refinement, we choose 0.7 as the threshold as this number is balanced between the coverage and the accuracy.

Effects of Different Combinations of Refinement Heuristics Obtained 0.7 as a good threshold to identify correct mappings, we use this number for the "Cosine Similarity Threshold " heuristic in the Refinement step. What we measure is the impact of the two refinement heuristics on the performance, either using only one of them or combine them together. The domain adaptation also comprises of Seeding and Adversarial Learning. After Adversarial Learning, we use different combinations of refinement heuristics to measure the effect of each heuristic. We use the 860 ground truth mappings from Java2CSsharp as the test set.

The results in Table 4.7 show that taking the Intersection between the Top-K Frequency and the Cosine Threshold heuristic results in the best performance.

[5]Note that the number of filtered mappings can be different when using different cosine similarity threshold to filter out the mappings in the query results whose similarity is less than the threshold. For the whole SAR approach, we do not apply the filtering for more strict evaluation.

Table 4.7: Different ways to combine refinement heuristics

Refine Method	Top-1	Top-5	Top-10
Top-K	0.44	0.68	0.76
Cosine	0.25	0.31	0.36
Union Top-K + Cosine	0.36	0.42	0.50
Intersection Top-K + Cosine	0.48	0.71	0.78

This implies that the Cosine Threshold has an effect to filter out poor Top-K Frequency synthetic seeds, thus making the refinement better in overall.

4.3.3.3 RQ3: Effect of Each Component

We performed an ablation study of domain adaptation to measure the performance of individual components as well as their combinations (Table 4.8). Note that for the Refinement component, since Section 4.3.3.2 shows that using the intersection of Top-K and cosine threshold leads to better results than union, we refer Refinement to those of "Intersection of Top-K and Cosine" performance.

Here are some observations from the results:

- The seeding step is an important step for the domain adaptation to works well, e.g., even with a small set of seeds (25), which is a very small knowledge, it sets up a basis for the adversarial learning to improve the performance significantly.
- Adversarial Learning is essential in improving the performance, e.g., comparing Seeding+Adv against Seeding, the top-1 accuracy is improved by 14% on average.
- Refinement alone does not achieve any good result because the initial input matrix was completely random that cannot be refined to anything better;
- Using the Adversarial Learning alone achieves some reasonable results, e.g., top-1 = 25%, top-10 = 35%. Further with the Refinement step, top-1 improves to 29%, top-10 becomes 40%. These can be seen as the results of unsupervised domain adaptation without any initial seeds.

4.3.4 Explainability Analysis of the Results

We performed various explainability analyses of our model in varying configurations to obtain some insights about our method. From the results, we show that our approach performs significantly better than Api2Api in every

Table 4.8: Ablation Study – effects of each component

Baselines	Seeds	Top-1	Top-5	Top-10
Seeding+Adv	25	0.30	0.39	0.45
	50	0.32	0.40	0.54
	100	0.39	0.53	0.71
	174	**0.43**	**0.67**	**0.75**
Seeding+Refine	25	0.13	0.14	0.20
	50	0.18	0.23	0.29
	100	0.22	0.29	0.43
	174	0.30	0.40	0.49
Seeding	25	0.11	0.15	0.18
	50	0.18	0.22	0.28
	100	0.20	0.27	0.36
	174	0.29	0.36	0.42
Refine	-	0.01	0.01	0.01
Adv+Refine	-	0.29	0.34	0.40
Adv	-	0.25	0.30	0.35

Table 4.9: Effect of Refinement on Frequent vs. Rare Tokens

Baselines	% Ground truth	Eval size	Accuracy		
			Top-1	Top-5	Top-10
With Refine	Top 10%	86	0.65	0.78	0.85
	Bottom 10%	86	0.32	0.35	0.47
Without Refine	Top 10%	86	0.54	0.65	0.72
	Bottom 10%	86	0.30	0.34	0.45

perspective. An interesting question one may ask is "*why does this approach perform better than Api2Api?*". Although theoretically, Adversarial Learning maximizes the similarity between two distributions, it is still useful to explain this phenomenon using analysis of the results.

4.3.4.1 Effect of Refinement on Frequent vs Rare tokens

We note that the frequency of an API token could affect the quality of the mapping result, i.e more frequent tokens could affect performance more than the less frequent ones. With this assumption, the Refinement of the mapping matrix tries to improve the mapping by using frequent tokens as the anchor. To measure the effect of the refinement on the frequent tokens and rare tokens, we ranked the 860 ground truth mappings in Java2CSharp by the frequency of the source APIs, i.e., the Java JDK APIs. Then we use our model to produce the mapping results against the top 10%, which is a subset of frequent tokens; and bottom 10%, which is a subset of rare tokens. Note that to ensure a fair comparison, we use the 174 non-overlapping seeds to train the domain adaptation procedure.

The results in Table 4.9 show the following observations:

- Mapping accuracy decreases while increasing top-k frequent tokens in the evaluation set, in either setting. This implies that token frequency does affect on the mapping result;

- The refinement step can improve the result of both the frequent tokens and

Table 4.10: Retrieved API mapping results from sample queries produced by SAR and Api2Api.

SAR	Api2Api
(1) java.util.Collection.add	
System.Collections.Objectmodel.Collection.Add	**System.Collections.Generic.List.Add**
System.Collections.Generic.List.Add	System.Collections.Generic.List.Get
System.Collections.ObjectModel.Collection.Clear	System.Collections.Generic.List.Remove
System.Collections.Generic.List.Contains	**System.Collections.Objectmodel.Collection.Add**
System.Collections.Generic.Dictionary.Add	System.Collections.IDictionary.GetEnumerator
(2) java.io.File.exists	
System.Io.File.Exists	**System.Io.File.Exists**
System.Io.File.AppendText	System.Web.Errorformatter.ResolveHttpFileName
System.Io.File.Delete	System.Io.File.OpenRead
System.Io.Fileinfo.LastWriteTime	System.Io.Compression.Zipfile.OpenRead
System.Io.File.GetAttributes	System.Io.Compression.ZipFile.ExtractToDirectory
(3) javax.swing.Text.JtextComponent.setCaretPosition	
System.Windows.Controls.RichTextBox.Clip	System.Drawing.Image.GetframeCount
System.Web.Ui.Webcontrols.DataGrid.PageSize	System.Media.SoundPlayer.PlaySync
System.Windows.Controls.RichTextBox.CaretPosition	System.Web.Ui.Webcontrols.Calendar.WeekendDayStyle
System.Windows.Forms.ContextMenuStrip.SuspendLayout	System.Configuration.Xmlutil.StrictSkipToNextElement
System.Windows.Controls.RichTextBox.CaretBrush	System.Media.SoundPlayer.PlayLooping
(4) java.util.concurrent.atomic.AtomicInteger.getAndDecrement	
System.Threading.Interlocked.Decrement	System.Directoryservices.SearchResultCollection.GetEnumerator
System.Threading.ReaderWriterLockSlim.EnterWriteLock	System.Directoryservices.SearchResultCollection.Dispose
System.Threading.Interlocked.Increment	System.Runtime.Serialization.ObjectIdGenerator.HasId
System.Threading.EventWaitHandle.OpenExisting	System.Collections.Generic.Queue.CopyTo

rare tokens, although the impact is bigger on frequent tokens, e.g., improved by 10% for top-10% , and only 2% for bottom-10%.

4.3.4.2 Retrieved Results Comparison

To evaluate our approach qualitatively, we retrieved C# API methods from sample queries in Java SDK. Table 4.10 shows the resulting top-5 C# APIs for four queries: *java.util.Collection.add*, *java.io.File.exists* *javax.swing.Text.JTextComponent.setCaretPosition*, and *java.util.concurrent.atomic.AtomicInteger.getAndDecrement*. They are ordered by increasing difficulty in finding a mapping.

For the first query, we can see that both Api2Api and our approach can successfully select the correct top-1 mapping, the other results are also related. This case can be considered as easy for both approaches to performing well. For the second query, both approaches can achieve a good exact mapping, but for the other results, our approach can generalize all of the results under the '*System.IO.File*' class, while there are some less related results in the top-5 produced by Api2Api, e.g., '*System.Web.ErrorFormatter.ResolveHttpFileName*'. The third query token ranks the 11,204th in the embedding table[6]. As discussed earlier, embedding quality of rare tokens is not as good as those of frequent tokens.

[6]The order of the token embedding provided by word2vec is proportional to the frequency of the token [80]

Therefore, it is more difficult to find an exact mapping for such a query. Even so, our approach can still rank a correct mapping at the third place (‘*System.Windows.Controls.RichTextBox.CaretPosition*’), while Api2Api produce totally unrelated results. For the last query, even though there has no mapping in C# by the ground truth, the retrieved results are still reasonably close. The query, in this case, is an API for an atomic operation, which is related to thread handling. Our approach can generalize the result mappings to the ‘*System.Threading*’ APIs in C#, while the results from Api2Api are totally unrelated.

This experiment shows that Adversarial Learning can maximize the similarity between the two distributions so that similar APIs are clustered together.

4.4 Threats to Validity and Limitations

The goal of domain adaptation is to use as little knowledge as possible for any pair of languages. However, we only perform the experiments on Java and C# because it is not easy to find a good and large enough evaluation dataset for other pairs of languages.

While unsupervised adversarial learning method does not require any seed as parallel data, there is a risk that the distributions of vectors (embeddings) in the two spaces are not so similar. Through our experiments, it is confirmed that the performance could be improved further by initializing the unsupervised adversarial learning method with a small set of seeds taken from the seed-based domain adaptation, and by generating the rest of API mappings.

4.5 Conclusion

In this chapter, we have proposed a domain adaptation approach, named SAR, to automatically transform and align the vector spaces of two different languages and APIs used therein. Before and after the adversarial learning step, we adapted the code embedding technique with a seeding and a refinement method respectively. SAR can identify API mappings across different programming languages. Our evaluation shows that the mappings between Java and C# APIs identified by SAR can be more accurate than other approaches with just 174 mapping seeds that can be easily identified by an automatic, simple signature-based heuristic, and that SAR helps to identify hundreds of more API mappings between Java and C# SDKs.

Domain adaptation methods are useful for other software engineering tasks that involve two different domains targeted by transferred learning [85, 86, 124], such as cross-language program classification, cross-language/project bug prediction. These tasks may benefit from the proposed approach when little curated data is available. Other SE tasks that are challenging due to lack of data, such as the out-of-vocabulary (OOV) problem [3, 35, 52] for learning and modeling fast-evolving software code, may also benefit from our domain adaptation approach, because the embeddings of OOV words may be approximated on-the-fly by adapting the known embeddings of their contextual or similar words in different languages. In the future, we will explore these variants of applications.

Chapter 5

Interpretability for Program Representation Learning

Learning from large corpora of code, or Big Code, deep learning based techniques have been adapted for various software engineering tasks, such as code completion, bug prediction, and program classification [4, 44, 82, 87, 133]. Despite high prediction accuracies achieved, deep neural networks are mostly treated as black boxes without explanation on why certain outputs are generated for certain inputs [11, 45, 64], so that users lack of confidence in the results. *Attention mechanisms* have been proposed [14, 76] for neural networks to focus on certain input elements or features when making predictions, and such elements or features are *assumed* to reflect certain interpretability of the networks. However, in many cases the features getting higher attentions by the networks may be implicit, and the prediction outputs produced by the attention networks according to the features may disagree with human users' understanding.

In this proposal, we aim to justify and improve the interpretability of attention-based neural networks with the AutoFocus approach. The approach is designed to reveal correlations between inputs and outputs of attention networks by perturbing inputs and observing the effects of perturbed inputs on the outputs. When applied to the attention networks trained for classifying programs (e.g., [82, 87, 133]), the approach helps to correlate attention scores for certain code elements (e.g., statements) of a program with the importance of the elements in determining the program's algorithm class. Such a correlation provides us a strong basis for using attention scores of individual statements as a metric to visualize a program, and thus helps users' in interpreting the neural networks' prediction outputs and understanding the program with increased

focus, saving the need to read through all code for comprehension.

To realize AutoFocus, we combine two intuitions:

1. **Syntax-Directed Attention**: We adapt attention mechanisms for algorithm classification neural networks, and generate attention scores for syntactically meaningful elements in the input programs (e.g., statements), instead of arbitrary elements, to facilitate code understanding;
2. **Code Perturbation**: We systematically perturb input programs according to syntactical structures too (e.g., deleting statements one by one) to observe how the perturbations affect neural networks' classification outputs and relate to the attention scores.

With the purpose to interpret tree-based and graph-based algorithm classification neural networks (TBCNN and GGNN [6, 82, 133]), our key research question here is:

Can the syntax-directed attention scores be used as a proxy to interpret the decisions made by the neural networks?

With code perturbation of hundreds of test programs evaluated on TBCNN and GGNN, we show that the attention scores of individual statements are strongly correlated with the importance of the statements on the classification results, and thus can be used to interpret the input/output behaviour of the networks. Furthermore, the statements in the test programs can be visualized according to the attention scores to facilitate more focused and faster code comprehension.

The interpretability produced by the AutoFocus approach technically only depends on the availability of attention scores and the interpretability of the input code elements that follow certain syntax. Thus, it is likely to be generally applicable to interpret many neural networks for various code learning tasks.

5.1 Overview

Figure 6.1 gives an overview of the six major steps in AutoFocus. Next section explains the steps in more details.

1. **Training of attention-based neural networks**: We add additional aggregation layers in the conventional neural networks (e.g., TBCNN [82] and GGNN [6]) for generating the attention scores for input elements using a *global* attention mechanism [14, 76]. Given training programs, we obtain trained attention networks.

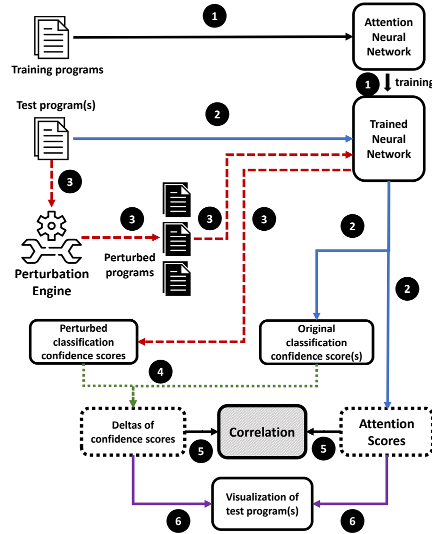


Figure 5.1: Overview of AutoFocus approach

2. **Generation of classification confidence score $c(p)$ for a test program p and attention scores $a(s)$ for each suitable code element s in p**: Given a test program p, the classification confidence score $c(p)$ is derived from the softmax layer of the attention networks, indicating the likelihood for p to belong to a certain algorithm class. For multi-class classification tasks (e.g., [82, 87, 133]), there is a confidence score for each class, while the correct class for p often but not necessarily has the highest confidence score. In this work, we always take the confidence score produced by the trained networks for the correct class of p as the $c(p)$. Meanwhile, the attention networks produce an attention score for each tree or graph node from the inputs, and we aggregate the scores according to p's syntactical structure and produce an attention score for each statement s in p, denote as $a(s)$.
3. **Perturbation of test program(s)**: each test program p is modified into a set of perturbed programs $P' = \{p'_s\}$, where p'_s indicates a perturbed program by deleting a certain statement s from p. For each perturbed program p'_s, we apply the attention networks to predict its class and obtain a new confidence score $c(p'_s)$.
4. **Impact measurement of perturbing statements**: Given a set of perturbed programs $\{p'_s\}$, we have a set of classification confidence scores $\{c(p'_s)\}$ which may be the same as or different from the classification confidence score $c(p)$ of the program p. The differences between $c(p)$ and $\{c(p'_s)\}$ are denoted as $\Delta(p) = \{\delta(s) = c(p'_s) - c(p) | s \in p\}$. Intuitively, a higher $\delta(s)$ may indicate the statement s has more impact on the attention networks' classification accuracies and thus more important for understanding p.

5. **Correlating statement-level attention scores $\{a(s)\}$ and perturbed confidence scores $\{\delta(s)\}$**: We analyze the Pearson Correlation Coefficients between the two kinds of scores for various test programs so that we may use the perturbed classification confidence scores to justify the uses of attention scores to interpret the classification decisions made by the attention networks.
6. **Visualization of statements**: Given the attention scores $\{a(s)\}$ and perturbed confidence scores $\{\delta(s)\}$ as a proxy for the importance of individual statements in a program p, we visualize p with a spectrum of derived colours to facilitate focused view on more important statements for program comprehension.

5.2 Approach Details

5.2.1 Building Attention Neural Networks

We choose state-of-the-arts tree-based and graph-based neural network graphs [6, 82, 133], for they yield accurate outputs for code classification tasks.

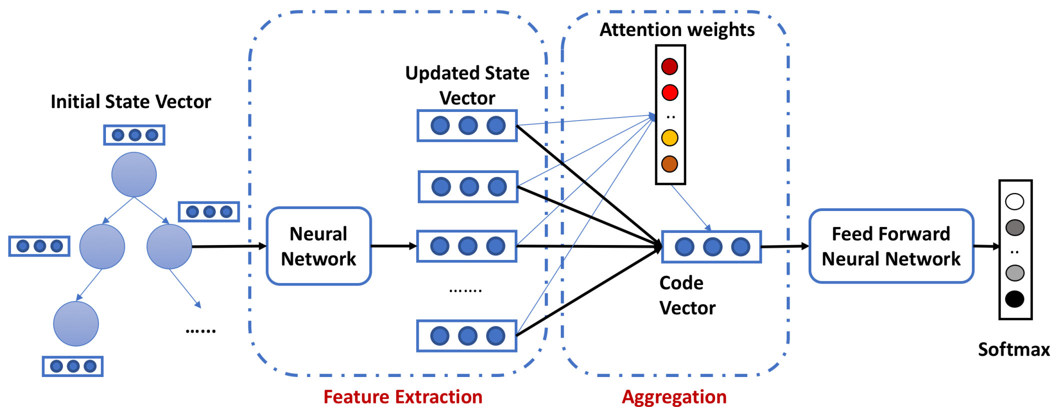


Figure 5.2: Attention mechanism as the aggregation layer for the neural network

Figure 5.2 illustrates the process of adding attention layers for algorithm classification neural networks. First, source code is parsed as an AST and a graph by connecting tree nodes to dependent ones. Then the neural networks are used as a feature extractor to update the information of each node following the edges. An aggregation layer is used to combine the information about all of the nodes into one single vector as the representation for the code (see Section 5.2.2).

Since a graph is a more general form of a tree, we summarize the design principle of both TBCNN and GGNN with graph notations. A graph $G = (V, \mathcal{E}, X)$ is composed of a set of nodes V, a set of node features X, and a list

of directed edges set $\mathcal{E} = \{(\mathcal{E}_1, ..., \mathcal{E}_K)\}$ where K is the number of edge types. Initially, we annotate each node $v \in V$ with a real-valued vector $x_v \in \mathcal{R}^d$ representing the features of the node. The node features X come from a pretrained embedding [82]. We associate every node v with a hidden state vector h_v, initialised from pretrained features embedding x_v.

The process of attention networks can be split into the feature extraction and the aggregation phases.

The **feature extraction** phase aims to propagate information from a node v to its neighbor. Specifically,

- The input to TBCNN is AST which is an undirected graph. A function f_{convo} aggregates the information of the direct children of a node v to update its state vector $h_v = f_{convo}(h_{children_of_v})$. This process runs through a few time steps to update the state vector x_v of node v.
- The input to GGNN is a graph representation of the AST plus additional edge types. GGNN can be described as a message passing network [6], where the "messages of type k are sent from each node v to its neighbor u, here k is corresponding to a particular kind of edge in the edge set E. The new state of the node v is computed from its current state vector, its neighbor, and the edge as: $h_v = f_k(h_v, h_u, e_K)$, where e_K is the edge of type K. We choose a linear function by following the suggestion of Allamanis et al. in [6]. This process also runs through a few time steps to update the state vector h_v.

Once the feature extraction process finishes, we have a matrix of dimension $m \times n$, where m is the number of nodes and n is the length of node feature embeddings. Then the **aggregation** phase aims to combine the hidden state vectors of all nodes in the graph into one single vector, which computes a feature vector for the whole graph (or tree) using an *aggegation function* R, such that: $y = R(\{h_v \mid v \in G\})$, and y is a vector of dimension $1 \times n$. R can be a max pooling function [82] which takes the max value of the features. However, it lacks the interpretability through which one does not know which node contributes more to the classification result. As such, we propose to use an attention mechanism as an aggregation function instead. The attention layer, in this case, will assign a score for each node in the input graph to represent its importance, which may lead to better interpretability from the human points-of-view.

5.2.1.1 Aggregation Using Attention Mechanism

Formally, a global attention vector $\boldsymbol{a} \in \mathcal{R}^d$ is initialised randomly and learned simultaneously with updates of the networks. Given n node state vectors: $\{\boldsymbol{h_1, ..., h_n}\}$, the *attention weight* α_i of each $\boldsymbol{h_i}$ is computed as the normalised inner product between the node state vector and $\boldsymbol{a}$: $\alpha_i = \frac{\exp(\boldsymbol{h_i}^T \cdot \boldsymbol{a})}{\sum_{j=1}^{n} \exp(\boldsymbol{h_j}^T \cdot \boldsymbol{a})}$. The exponents are used to make the attention weights positive, and they are divided by their sum to have max value of 1, as done by a standard softmax function.

The aggregated code vector $\boldsymbol{v} \in \mathbb{R}^d$ represents the whole code snippet. Its *attention score* is derived from the linear combination of the node's state vectors $\{\boldsymbol{h_1, ..., h_n}\}$ weighted by their attention weights: code vector $y = \sum_{e=1}^{n} \alpha_e \cdot \boldsymbol{h_e}$.

5.2.1.2 Objective Function

Since we aim for the code classification task, we use the cross-entropy as the objective function to train our network, which is defined as $J(\theta, \hat{x}, c) = \sum(-log\frac{\exp(\hat{x}_c)}{\sum_j \exp(\hat{x}_j)})$, where θ denotes parameters of all the weight matrices in our model, $\hat{x}$ is the predicted classification vector for all the class labels and c is the true label.

5.2.2 Deriving Statement-Level Attention Scores

The purpose of this step is to derive attention scores for code elements at specific levels of granularity to tell the importance of the code elements. In this work, we consider attention scores at the statement level. The attention score for a statement node in an AST is obtained by a simple summation of the attention scores of all descendant nodes of the statement node. For later visualization of the program source code (Section 5.2), we also need to map the attention scores of statement nodes to the actual tokens in the statements. When a token belongs to multiple nested statements, the attention score of the closest enclosing statement is used as the score for the token.

5.2.3 Finding Decision-Changing Subset of Code Component

5.2.4 Definition of Code Component

Formally, let a denote the AST representation of an input program, which has a set of nodes $N = \{n_1, n_2, n_3, ..., n_K\}$, where K is the number of nodes in the

AST. Each node in the AST has a unique id for it.

Our goal is to find components in the AST such that without the availability of such components, the model's decision will change. We define a code component c as a subset of nodes of N.

5.2.5 Different Types of Code Components

In this work, we focus on these two types of code components:

- **Single Code Component**: can be seen as a single subtree of the original AST, which can be a statement, an expression or even a token. In this work, we focus on the single code component at the statement level, although the same principle can be applied to other types of single components.
- **Dependency Code Component**: is the code component that may contain multiple code components that are dependent on each other, i.e., program slice.

5.2.5.1 Single Code Component

A single code component can be a statement, an expression or even a token. In this work, we focus on the statement level, although the same principle can be applied to another level. The statement, in this case, is stand-alone, means that we assume any of the statement has no relation with the other statements. Since we represent a program as an AST, each of the statement can be seen as a small sub-trees. Figure 5.3a is an example of a single code component, all of the nodes in the subtree $a = 2$ forms a code component.

There are a few cases that need special treatment.

- We treat the Function Declaration as a special statement.
- For a nested statement, such as *while*, *for* that includes the header and sub-statements inside the body, we split the head of such statement and all included statements.

Figure 5.4 shows an example of the above two special treatments . The Function Declaration for the function *appendFile* is considered as a special statement node. The headers of the *while* (line 4) or the *if* (line 5 and 8) are now considered as a separate statement. By this way, we can break a large AST representation of a program into smaller sub-trees, each of the subtrees is independent of each other. We would like to measure how different

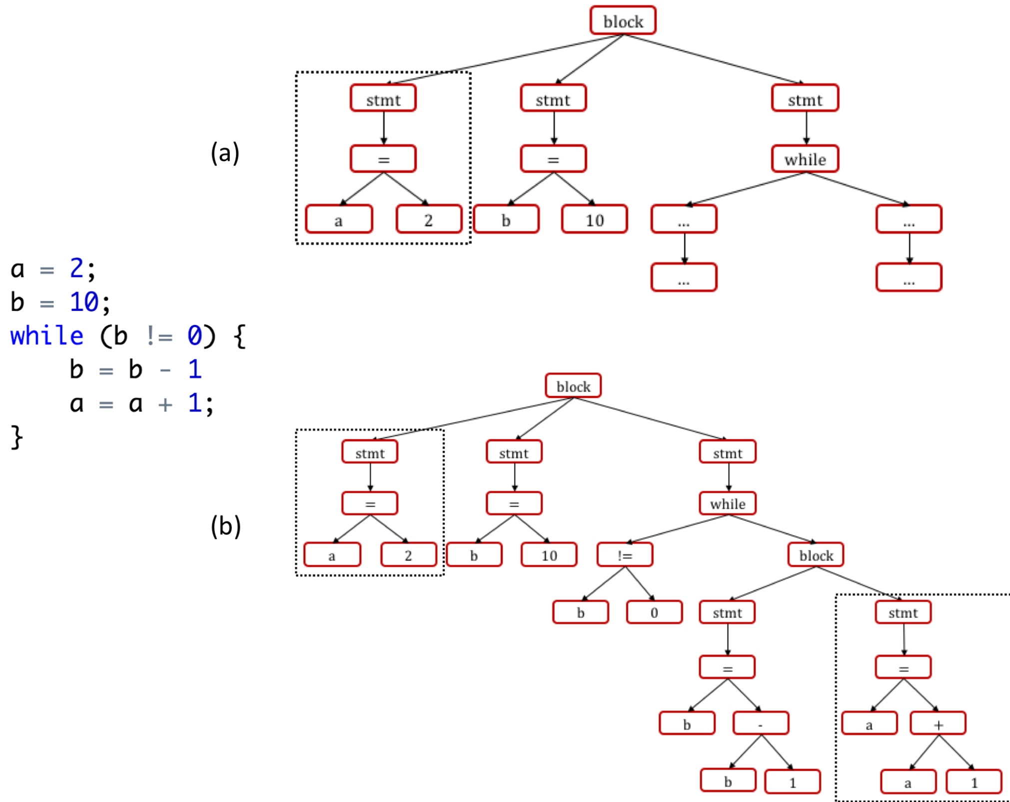


Figure 5.3: Example of different component types

combination of these statements may affect the decision of the neural network on the program.

5.2.5.2 Dependency Code Component

Formally, a static program slice s consists of all statements in a program P that may affect the value of variable v in a statement x. As such, each slide contains a set of statements $\{x_1, x_2, s_3, ...\}$. Note that difference slices can contain similar statements. A program P contains a set of program slices $S = \{s_1, s_2, s_3, ..., s_M\}$ where M is the number of variable in program P. The union of all the nodes in all of the statements subtrees in a slice represents for a code component.

Figure 5.3b is an example of dependency code component. The slice for variable a comprises of two statements, which are $a = 2$ and $a = a+1$. As such, the union of the nodes in the two subtrees represent for $a = 2$ and $a = a + 1$ forms a code component in this case.

```
1  private static void appendFile(String fileName, boolean keep) {
2          try {
3              BufferedReader br = new BufferedReader(new FileReader
                   (fileName));
4              while (true) {
5                  String line = br.readLine();
6                  if (line == null)
7                      break;
8                  if (keep || !map.containsKey(line)) {
9                      System.out.println(line);
10                     map.put(line,line);
11                 }
12             }
13             br.close();
14         } catch (Exception e) {
15             e.printStackTrace();
16             System.exit(1);
17         }
18     }
```

Figure 5.4: Example of a program after broken down into smaller statement sub-trees

5.2.5.3 Finding a Subset of Decision-Change Code Components

We want to answer the question: "*How can we identify a set of important code components that that significantly contribute to a model's decision for the program classification task?*". We propose a technique, so-called **Minimality of Deletion**: deleting the minimum number of code components to change the model's prediction. The rationale behind this is that once we found the set of code components that changes the model's decision, it implies that we found the most important parts of the source code. Note that the deletion step here is the way to find important parts of the source code, not the other way around.

5.2.5.4 Problem Formulation

Let a denote the AST representation of an input program. We derive a set of components $C = \{c_1, c_2, ..., c_N\}$ from a, where N denotes the number of code components in a. Let L_a be the index of the label that M gives to a. Our task is to explore all of the subset D' of a through a function f that receives D' so that f can identify which subset are the most important in a. This can be formulated as the task to discover the minimal subset of a, denoted by $D \subset a$, such that the deletion of all code components in D from a will change the label L_a, i.e., set D is the most important parts of the program, where the deletion step can be abstracted as a function f to measure the importance of a subset. The deletion step is the process to *delete all of the nodes in a component* c out of the AST.

By this way, we can identify the important code components to present to the user, i.e., we highlight all the components in D when present the program

to the user. The remaining code components are denoted as $a - D$, which indicates that the code components in $a - D$ are not important and can be deemphasized from the program when visualizing the components to the user. Let $|D|$ denote the number of components in D. The problem is formalized as follows:

$$min_D|D| \; s.t \; L_{a-D} \neq L_a \tag{5.1}$$

One can enumerate through all different component combinations to find the optimal solution, but it will be computationally intractable when the number of components in a gets large. To address this issue, we formalize this problem as the search space optimization problem. We define the state of the program after deleting an arbitrary set of code components D' as $S_{D'}$, i.e., a without the set of sub-trees D'. We denote the original state of a as S_0, i.e., a as the original AST. The goal is to find the state that make the label changes. This task can be simulated as a graph search problem, in which each node in the graph represents for a state of a after deleting one or multiple code components.

Concretely, the label of each node are presented as $(S_{D'}, L_{a-D'})$, where D' is a subset of code components to be deleted, e.g., $(S_{\{1,2\}}, 2)$ means after deleting two components c_1 and c_2 from a, the index of the label that M gives to a is 2. The root node is a special node, which is presented as (S_0, L_a) means the label that M gives to a is L_a.

Our task is to search through the graph to find the state that change the label of the programs, i.e, finding a node $(S_{D'}, L_{a-D'})$ where $L_{a-D'} \neq L_a$.

In Figure 5.4, the components in this example are the statements. Each statement is assigned an index and will used as a part of the label for nodes in the search graph. The same thing can be applicable is the component is a slice of the program.

We use the ***attention score*** to guide the searching step to find the optimal state. Assume that we have already gotten the attention score for each of the components, we incorporate the attention score into the edges between different states as guidance of where to go next when searching on the graph, i.e., the score of the incoming edge on a state represents for a attention score the model assigns for the code components. Next we present the algorithms to search for the optimal state in the graph by using the attention score as a guidance.

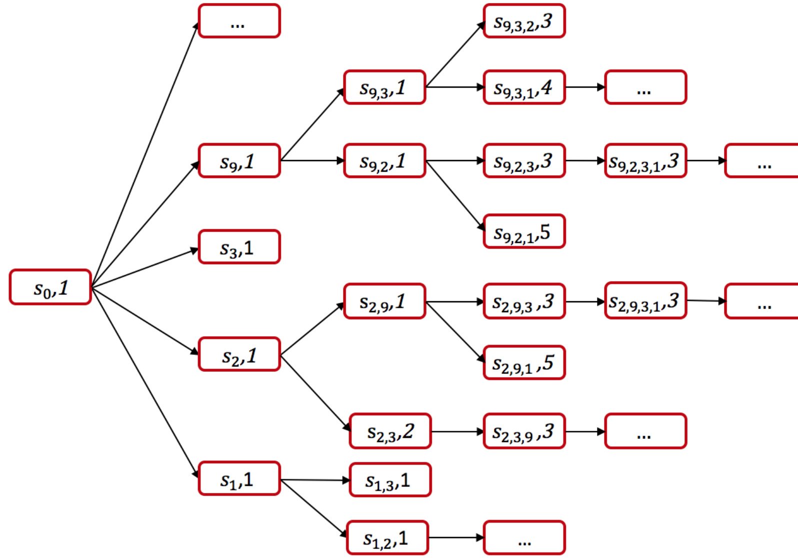


Figure 5.5: Search Graph Example

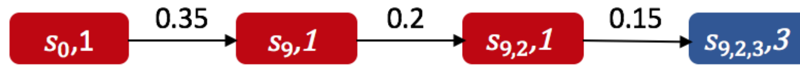


Figure 5.6: Example of Greedy 1-best Search

5.2.5.5 Greedy 1-best Search

The simplest way of searching for the state is greedy search, in which we simply expand to the next state based by selecting the state that gives us the highest score (score on the arrow), and use it as the next state in our searching process. Algorithm 1 shows the process of the Greedy Search algorithm. We are given: (1) M as a trained neural network, (2) S_0 as the original AST, i.e., original state of the program, (3) L as the label of S_0 that is assigned by M, (4) C as the set of all component ids, (5) E as a set of component ids, which is the found solution. At each time step, the greedy search algorithm selects the code components which has the highest attention score, delete that one from S_0, then use M to check if the label of the modified S_0 is change or not. If the label does not change, we move to the next time step to delete the code components that has the highest attention score, we do this until we find the series of deletion that make the label changes.

Figure 5.6 shows an example of the algorithm on the search graph. From the root node, we traverse to the next level of the tree, we identify that the the edge that has the highest attention score (0.35) leads to the node $(S_9, 1)$, we move the pointer to node $(S_9, 1)$. Next, the edge that has the highest attention score (0.2) leads to the node $(S_{9,2}, 1)$, them move the pointer to node $(S_{9,2}, 1)$.

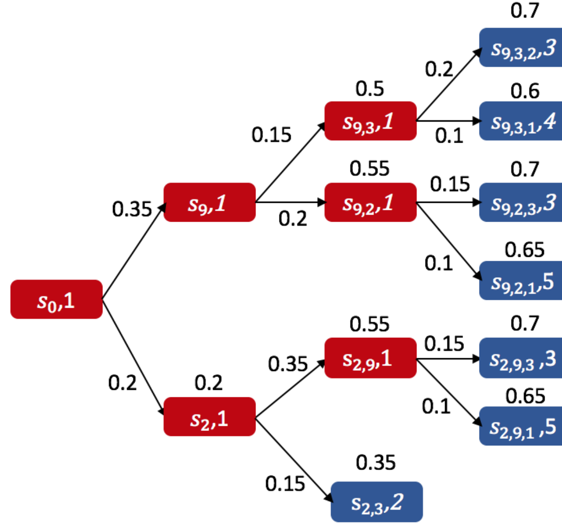


Figure 5.7: Example of Beam Search, beam size $b = 2$

Finally, we can identify that the edge that leads to node $(S_{9,2,3}, 3)$ makes the label change from 1 to 3. We stop the search process at node $(S_{9,2,3}, 3)$ and take the set c_9, c_2, c_3 as the set D to be found.

Algorithm 1 Greedy 1-best Search

1: $M \leftarrow$ Neural Network
2: $S_0 \leftarrow$ Original AST
3: $L \leftarrow$ Label Index of S_0
4: $C \leftarrow$ Index of All Components
5: $E \leftarrow$ Solution
6: **procedure** GREEDYSEARCH(S_0, L, C, E)
7: $\quad maxC \leftarrow FindComponentWithHighestScore(C)$
8: $\quad S \leftarrow DeleteComponent(maxC,\ S_0)$
9: $\quad E \leftarrow E + \{maxC\}$
10: $\quad C \leftarrow C - \{maxC\}$
11: $\quad L' \leftarrow M(S)$
12: $\quad$ **if** $L' == L$ **then** $GreedySearch(S, L', C, E)$
13: $\quad$ **else**
14: $\qquad return\ E$
15:

5.2.5.6 Attention-based Beam Search

Greedy Search is not guaranteed to find the state with the minimal subset of components D, because it always selects the state with the highest the attention score on the incoming edge, thus it loses the opportunity to explore the other branches of the graph, which may lead to a better solution. To address this issue, we propose to adapt the idea of Beam Search, into our context, so-called

Algorithm 2 Beam Search

```
1: M ← Neural Network
2: S_0 ← Original AST
3: L ← Label Index of S_0
4: C ← Index of All Components
5: E ← Solution
6: P ← All Potential Solutions
7: b ← Beam Size
8: procedure BeamSearch(S_0, L, C, E, b)
9:     candidateC ← FindTop_b_Components(C, b)
10:    for c in candidateC do
11:        S ← DeleteComponent(c, S_0)
12:        E ← E + {c}
13:        C ← C − {c}
14:        L′ ← M(S)
15:        if L′ == L then BeamSearch(S, L′, C, E, b)
16:        else
17:            P ← P + E
18:
19: procedure SelectBestSolutions(P)
20:    P ← P − {SelectSolutionsWithMinC(P)}
21:    P ← P − {SelectSolutionsWithMinSumAttention(P)}
22:
23: O ← SelectBestSolutions(P)
```

Attention-based Beam Search. Beam Search is similar to greedy search, but instead of considering only the one best states, we consider b best states at each time step, where b is the width of the beam. Algorithm 2 shows the pseudo-code of our proposed Attention-based Beam Search.

We are given: (1) M as a trained neural network, (2) S_0 as the original AST, i.e., original state of the program, (3) L as the label of S_0 that is assigned by M, (4) C as the set of all component ids, (5) E as a set of component ids, which is the potential solution, and (6) P as a set of all potential solutions, which we will use some conditions to select the best one. At each time step, the algorithm selects the best top_b components based on the attention scores. Then the process to delete the components is similar to the one in Greedy Search. Except in Line 17, instead of returning the solution if we can find one, we store it into a set of all potential solutions, because by exploring different paths of the search space, we can find many potential solutions and we use two conditions to select the best one. The first condition is that if we can find many solutions, we select the ones with the minimum number of components. The second condition is that if there are more than 1 solutions with the same number of components, we select the one that has the lower summation of attention scores. The rationale for the second condition is that if the set of components receive lesser attention but still change the label, which means that we are closer to the minimal.

An example of beam search with $b = 2$ is shown in Figure 5.7. Note that in this case, there is the number on top of each state, which represents for the summation of attention score for the set of components represented at that state, e.g., the score 0.55 on top of the node $(S_{9,2}, 1)$ is the sum of attention scores for statement 9 and statement 2 ($0.2 + 0.35 = 0.55$). From the original state, we expand to the two nodes $(S_{2,1}, 1)$ and $(S_{9,1}, 1)$ since they are the two states with highest attention scores. We keep expanding until we find the set of all possible solutions, which are $(S_{9,3,2}, 3)$, $(S_{9,3,1}, 4)$, $(S_{9,2,3}, 3)$, $(S_{9,2,1}, 5)$, $(S_{9,3,2}, 3)$, $(S_{2,9,3}, 3)$, $(S_{2,9,1}, 5)$, $(S_{2,3}, 2)$. Among these solutions, $(S_{2,3}, 2)$ has the minimal number of components, thus it is the final solution. This example shows that for the same program, the Greedy Search fails to find the minimal set of components since its solution is $(S_{9,2,3}, 3)$.

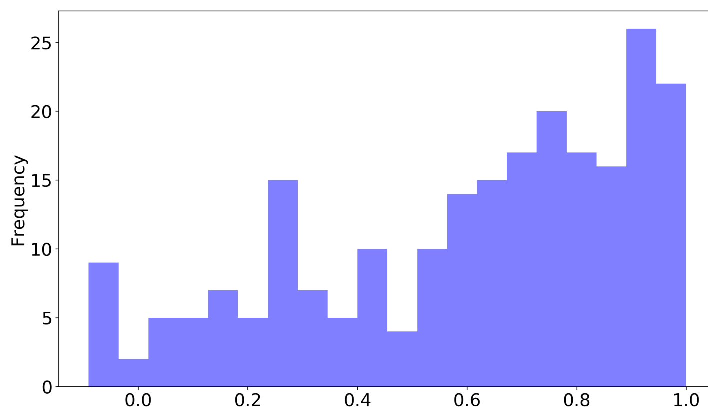


Figure 5.8: Histogram of Pearson Correlation Coefficients of all test data

5.3 Evaluation

We evaluate the interpretability of attention networks by checking whether statement-level attention scores correlate to the deltas in classification confidence after code perturbation. The data used for the evaluation consists of 1023 unique Java programs crawled from GitHub for 10 distinct sorting algorithms [87], where 70% of the data was used for training, 10% for validation, and 20% for testing.

In the experiments, the attention settings described in GGNN [6] are used to train a model of 85% accuracy on the test data of 200 programs, which is fairly good as the ground truth for interpretation. For each test program, we follow the steps described in Section 5.1 to derive attention scores and deltas for deleting statements. We conducted a statistical analysis on the correlation between the deltas and the attention scores over the statements deleted by code perturbation. Following Step 5 in Section 5.1, we obtained the Pearson correlation ratio. For all the test programs, a list ot Pearson correlation ratios can be seen as a discrete variable P. Figure 5.8 shows the histogram of P, whose mean value is 0.65 and standard deviation is 0.26. This indicates a strong correlation between the attention scores and the deltas.

Based on the intuition that statements are a reasonable granularity for developers to understand a program, the strong correlation gives us a basis to use attention scores to interpret neural networks and build code visualizations for more focused views to facilitate program comprehension. Figure 5.9 exemplifies the visualization of attention scores of statements inside Visual Studio Code IDE. The left pane visualizes the statements according to their attention scores. The higher the attention score, the darker color the statement gets. The

right pane visualizes the statements according to the confidence deltas of each statement, which is similar but slightly different.

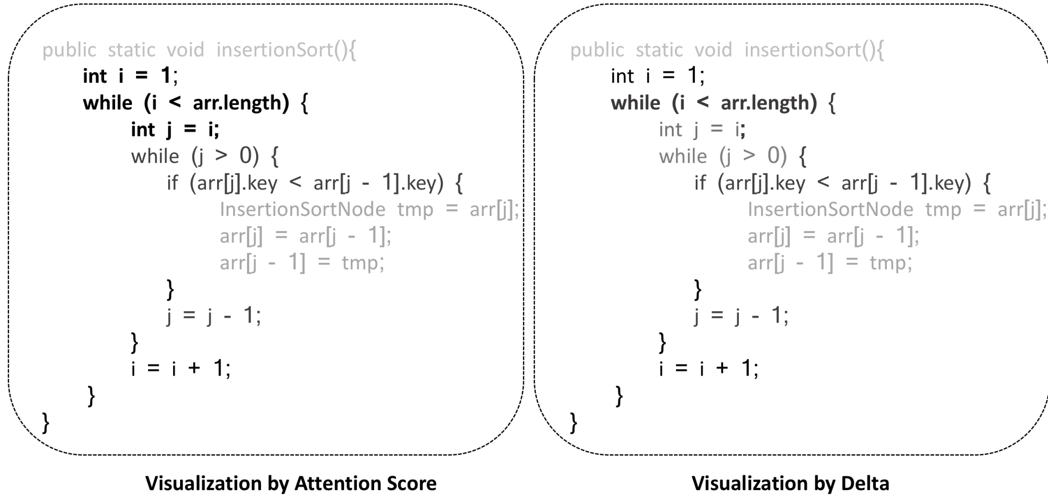


Figure 5.9: AutoFocus visualization of attention scores in Visual Studio Code

5.4 Future Work

To summarize, what we have done so far is:

- We have implemented the algorithm to find the single important code components by deleting the code components one by one.
- We have implemented the tool to highlight the importance of code components by using a range of colors to assign to the components.
- We found the evidence to show that the attention score has a strong correlation with the important parts of the source code. This will be leverage to find a set of decision-change code components, as described in 5.2.5.3.
- We will implement the algorithm to find the set of decision-change code components according to the algorithm in Section 5.2.5.3
- Once we find the set of important code components, we will visualize them and show it to the user. Concretely, all of the important code components will be assigned the black color, while the remaining will be assigned the gray color to deemphasized them.
- We need a method to evaluate if the algorithm can find the important code components. We use human study to validate this. First, we show the source

code to the user and ask him to label which part is important to form a specific algorithm, e.g. which part is the most important to decide if a piece of code is an implementation of bubble sort. This can be considered as the step to label the ground truth. Then we compare the manual labeling code with the code that has been highlighted by our algorithm to measure the difference between them.

- Besides the code classification task, our algorithm is application into tasks such as bug prediction and code summarization. This will provide stronger evidence to prove that our interpretability algorithm is applicable into a wide range of software engineering tasks.

Chapter 6

Tree-based Capsule Network for Code Processing

6.1 Introduction

Software developers often spend the majority of their time in navigating existing program code bases to understand the functionality of existing source code before implementing new features or fixing bugs [20, 40, 121]. Learning a model of programs has been found useful for their tasks such as classifying the functionality of programs [36, 96, 99, 109], predicting bugs [67, 71, 125, 139], translating programs [26, 49], etc.

It is a common belief that adding semantic descriptions (e.g., via code comments, visualizing code control flow graphs, etc.) enhances human understanding of programs and is also helpful for machine learning. With the help of static code dependency analysis techniques [95], for example, Gated Graph Neural Networks (GGNN) [3, 41, 70] learn code semantics via graphs where edges are added between the code syntax tree nodes to indicate various kinds of dependencies between the nodes. However, such edges may be noise due to inaccurate code analyses that are inherently unsound or incomplete, and too many such edges contribute to long training time and may hinder the learning techniques from achieving higher accuracies [23]. Another issue with the GGNN method is that it relies on a synchronous message-passing mechanism to accumulate information from a node's neighbors to learn their embeddings and it usually uses as few as eight message-passing iterations due to computational concerns [3]. This mechanism hinders GGNN from capturing information from distant parts in large graphs.

There also exist deep learning techniques that process code syntax trees or

abstract syntax trees (ASTs) [10, 83, 134]. However, they are limited in how they represent and learn ASTs although ASTs entail code semantics precisely. The TBCNN [83] method shares the same computational principle with the GGNN method, i.e., information is accumulated from children to parent nodes only, which limits the number of iterations for a node to accumulate information from its distant descendants. Code2vec [10] decomposes trees into a bag of path-contexts for learning, while ASTNN [134] splits big trees for programs and functions into smaller subtrees for individual statements. Then, they adapt different recurrent neural network models to learn the path-contexts or flattened subtrees but are likely to miss code dependency information that is not represented in the decomposed paths and subtrees.

It is desirable to learn code semantics via ASTs directly because trees can be efficiently and precisely constructed from code without any inaccurate semantic analysis and the model should not be limited by a few numbers of message-passing steps when accumulating information. In this chapter, we proposes a novel architecture called **TreeCaps** by fusing capsule networks [111] with TBCNN [83] to process tree inputs directly.

TreeCaps first adapts TBCNN to take in trees and extract (local) node features with its convolution capability and converts the node features into capsules in its *Primary Variable Capsule* (PVC) layer where the number of capsules can change for different tree inputs. It then adapts CapsNet by introducing two methods to route the dynamic number of capsules in PVC to a static number of capsules in its *Secondary Capsule* (SC) layer. Our first method inherits the dynamic routing algorithm [111] for static numbers of capsules; it shares a global transformation matrix across every pair of capsules between the layers [135, 136]. Our second method is a novel *Variable-to-Static* (VTS) routing algorithm that selects the capsules with the most prominent outputs in the PVC layer and squeezes them into a fixed set of capsules. The method utilizes the common intuition that code semantics can often be determined by considering only a portion of code elements. Further, we apply a dynamic routing algorithm from the capsules in the SC layer to the final *Code Capsule* (CC) layer whose number of capsules is fixed according to a specific learning task, to get the vector representations of the trees for the task. Compared to the max-pooling method to combine node features in TBCNN, the pipeline of our routing methods (PVC→SC →CC) can learn more sophisticated combinations of features in the ASTs.

Across codebases in C/C++ and Java with respect to commonly compared

program comprehension tasks such as code functionality classification and function name prediction, our empirical evaluation shows that TreeCaps achieves better classification accuracy and better F1 score in prediction compared to other code learning techniques such as GGNN, Code2vec, ASTNN, and TBCNN. We have also applied three types of semantic-preserving transformations [105, 132] that transform programs into syntactically different but semantically equivalent code to attack the models. Evaluations also show that our TreeCaps models are the most *robust*, able to preserve its predictions for transformed programs more than other learning techniques.

6.2 Tree-based Capsule Networks

An overview of the TreeCaps architecture is shown in Fig. 6.1. The code snippet

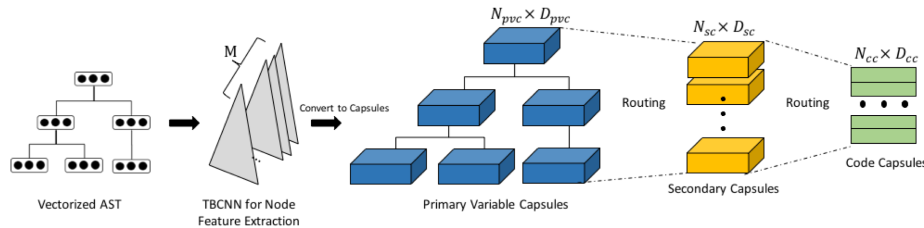


Figure 6.1: Overview of TreeCaps: Source codes are parsed, vectorized and fed into the TBCNN to extract node features, then the node features are combined through the TreeCaps network.

in the training data is parsed into an AST and vectorized. The node vectors are fed into the TBCNN to extract node features to be used as the input for the Primary Variable Capsule (PVC) layer where the number of capsules can change dynamically according to the input tree size. The capsules in the PVC layer are then routed and reduced to a fixed number of capsules in the Secondary Capsule (SC) layer. The outputs of the SC layer are routed to the final Code Capsule (CC) layer. The capsules in the CC layer can be seen as the vector representations for the input code, and can be trained with respect to different code comprehension tasks, such as code functionality classification and function name prediction, which are evaluated in the next sections.

6.2.1 Tree-based Convolutional Neural Networks

We briefly introduce the Tree-based Convolutional Neural Networks (TBCNN, [83]) for processing tree-structured inputs used in TreeCaps.

A tree $T = (V, E, X)$ consists of a set of nodes V, a set of node features X, and a set of edges E. An edge in a tree connects a node and its children. Each node in an AST also contains its corresponding texts (or tokens) and its type (e.g., operator types, statement types, function types, etc.) from the underlying code. Initially, we annotate each node $v \in V$ with a D-dimensional real-valued vector $x_v \in \mathbb{R}^D$ representing the features of the node. We associate every node v with a hidden state vector h_v, initialized from the feature embedding x_v, which can be computed from a simple concatenation of the embeddings of its texts and type [3]. The embedding matrices for the texts and types are learn-able in the whole model training pipeline.

In TBCNN, a convolution window over an AST is emulated via a binary tree, where the weight matrix for each node is a weighted sum of three fixed matrices $\mathbf{W}^t$, $\mathbf{W}^l$, $\mathbf{W}^r \in \mathbb{R}^{D\times D}$ (each of which is the weight for the "top", "left", and "right" node respectively) and a bias term $\mathbf{b} \in \mathbb{R}^D$ Hence, for a convolutional window of depth d in the original AST with $K = 2^d - 1$ nodes (including the parent nodes) belong to that window with vectors $[\mathbf{x}_1, ..., \mathbf{x}_K]$, where $\mathbf{x_i} \in \mathbb{R}^D$, the convolutional output $\mathbf{y}$ of that window can be defined as follows: $\mathbf{y} = tanh(\sum_{i=1}^{K}[\eta_i^t \mathbf{W}^t + \eta_i^l \mathbf{W}^l + \eta_i^r \mathbf{W}^r]\mathbf{x}_i + \mathbf{b})$, where $\eta_i^t, \eta_i^l, \eta_i^r$ are weights calculated corresponding to the depth and the position of the nodes. A TBCNN model usually stacks M such convolutional layers to generate the final node embeddings, where output at layer m will be used as the input for the next layer $m + 1$. Each layer has its own $\mathbf{W}^t$, $\mathbf{W}^l$, $\mathbf{W}^r \in \mathbb{R}^{D\times D}$ and the bias term $\mathbf{b} \in \mathbb{R}^D$ with different initialization.

6.2.2 The Primary Variable Capsule Layer (PVC)

In the PVC layer, we use M tree-based CNN layers with different random initializations for $\mathbf{W}^t$, $\mathbf{W}^l$, $\mathbf{W}^r$ and $\mathbf{b}$. We group outputs of the convolutional layers together to form $N_{pvc} = |V| \times D$ sets of capsules with outputs $\mathbf{c}_i \in \mathbb{R}^{D_{pvc}}$, $i \in [1, N_{pvc}]$, where $D_{pvc} = M$ is the dimension of the capsules in the PVC layer. We apply a non-linear squash function [111] to a capsule to produce $\mathbf{u}_i$, which represents the probability of existence of an entity by the vector length:

$$\mathbf{u}_i = \frac{||\mathbf{c}_i||^2}{||\mathbf{c}_i||^2 + 1} \cdot \frac{\mathbf{c}_i}{||\mathbf{c}_i||} \tag{6.1}$$

Hence, the output of the primary variable capsule layer is $\mathbf{X_{pvc}} \in \mathbb{R}^{N_{pvc}\times D_{pvc}}$.

6.2.3 The Secondary Capsule Layer (SC)

As argued by [111], the capsule network tries to address the representational limitation and exponential inefficiencies of convolutions with transformation matrices. To this, the child capsules in the PVC layer will be routed to the parent capsules in the next capsule layer through a transformation matrix.

6.2.3.1 Sharing Weight across Child Capsules with Dynamic Routing (DRSW)

Since the number of capsules in the PVC is dynamic, a global transformation matrix cannot be defined in practice with variable dimensions. The solution for this problem is to defined a shared transformation matrix $\mathbf{W_s} \in \mathbb{R}^{N_{pvc} \times D_{pvc} \times D_{sc}}$ across the child capsules, where N_{pvc} is the number of capsules in the PVC layer [136], D_{sc} is the dimension of the capsules in the SC layer, and use the dynamic routing algorithm to route the capsules (as summarized in Algo.3).

For each capsule i in the PVC layer (layer l in Algo.3), and for each capsule j in the SC layer (layer $l+1$ in Algo.3), we multiply the output of the PVC layer $\mathbf{u}_i$ by the *shared* transformation matrix $\mathbf{W}_s$ to produce the prediction vectors $\hat{\mathbf{u}}_{j|i} = \mathbf{W}_s \mathbf{u}_i$. The "prediction vectors" are responsible to predict the strength of each capsule in the PVC layer, then a weighted sum over all "prediction vectors" $\hat{\mathbf{u}}_{j|i}$ will produce the capsule j in the SC layer. The trainable shared transformation matrix learns the part-whole relationships between the primary capsules and secondary capsules, while effectively transforms $\mathbf{u}_i$'s into the same dimensionality as $\mathbf{v}_j$ where each $\mathbf{v}_j$ denotes the capsule output of the SC layer. The coupling coefficients β_{ij} between capsule i and all the capsules in the SC layer sum to 1 and are determined by a "routing softmax" whose initial logits α_{ij} are the log prior probabilities that capsule i in PVC layer should be coupled to capsule j in the SC layer. Then we use r iterations to refine β_{ij} based on the agreements between the prediction vectors $\hat{\mathbf{u}}_{j|i}$ and the secondary capsule outputs $\mathbf{v}_j$ where $\mathbf{v}_j = squash(\sum_i \beta_{ij} \hat{\mathbf{u}}_{j|i})$.

Algorithm 3 Dynamic Routing

1: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
2: Initialize $\forall i \in [1, l], \forall j \in [1, l+1], \alpha_{ij} \leftarrow 0$
3: **for** r iterations **do**
4: $\forall i \in [1, l], \beta_i \leftarrow softmax(\alpha_i)$
5: $\forall j \in [1, l+1], \mathbf{v}_j \leftarrow squash(\sum_i \beta_{ij} \hat{\mathbf{u}}_{j|i})$
6: $\forall i \in [1, l], \forall j \in [1, l+1], \alpha_{ij} \leftarrow \alpha_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$
7: **return** $\mathbf{v}_j$

Algorithm 4 Variable-to-Static Capsule Routing

1: **procedure** ROUTING($\mathbf{u}_i, r, a, b$)
2: $\quad \mathbf{U_{sorted}} \leftarrow sort([\mathbf{u}_1, ..., \mathbf{u}_b])$
3: $\quad$ Initialize $\mathbf{v}_j : \forall i, j \leq a, \mathbf{v_j} \leftarrow \mathbf{U_{sorted}}[i]$
4: $\quad$ Initialize $\alpha_{ij} : \forall j \in [1, a], \forall i \in [1, b], \alpha_{ij} \leftarrow 0$
5: $\quad$ **for** r iterations **do**
6: $\quad\quad \forall j \in [1, a], \forall i \in [1, b], f_{ij} \leftarrow \mathbf{u}_i \cdot \mathbf{v}_j$
7: $\quad\quad \forall j \in [1, a], \forall i \in [1, b], \alpha_{ij} \leftarrow \alpha_{ij} + f_{ij}$
8: $\quad\quad \forall i \in [1, b], \boldsymbol{\beta}_i \leftarrow Softmax(\boldsymbol{\alpha}_i)$
9: $\quad\quad \forall j \in [1, a], \mathbf{v}_j \leftarrow Squash(\sum_i \beta_{ij} \mathbf{u}_i)$
10: $\quad$ **return** $\mathbf{v}_j$

6.2.3.2 Variable-to-Static Routing (VTS)

Sharing the transformation matrix reduces the ability to learn different features as each pair of capsules is supposed to have its transformation matrix. Because of this limitation, we offer the second solution to route the variable number of capsules in the PVC layer. It is based on an observation of source code that, in practice, not every node of the AST contributes towards a source code learning task. Often, source code consists of non-essential entities, and only a portion of all entities determine the code class. Therefore, we propose a novel variable-to-static capsule routing algorithm, summarized in Algo. 4. The intuition of this algorithm is that we squeeze the variable number of capsules in the PVC layer to a static number of capsules by choosing only the most important capsules in the PVC layer. The major difference between the VTS algorithm and the DRSW algorithm is that the DRSW needs to produce prediction vectors by multiplying the capsule outputs in PVC layer with the shared transformation matrix, and then the prediction vectors will be combined to produce the capsules for SC layer, while in the VTS, the capsule outputs in the PVC layer are selected and the prominent ones are used to initialize for the capsules in SC layer directly.

We initialize the outputs of the SC layer with the outputs of the a capsules with the highest L_2 norms in the PVC layer. Hence, the outputs of the PVC layer, $[\mathbf{u}_1, ..., \mathbf{u}_{N_{pvc}}]$, are first sorted by their L_2 norms, to obtain $\mathbf{U_{sorted}}$, and then the first a vectors of $\mathbf{U_{sorted}}$ are assigned as $\mathbf{v}_j, j \leq a$.

Since the probability of the existence of an entity is denoted by the length of the capsule output vector (L_2 norm), we only consider the entities with the highest existence probabilities for initialization (in other words, highest activation) following the aforementioned intuition. It should be noted that the capsules with the a-highest norms are used *only for the initialization*; the actual

outputs of the static capsules in the SC layer are determined by iterative runs of the variable-to-static routing algorithm. It is the capsules with the most prominent outputs along with the capsules of the highest vector similarities to them that get routed to the next layer. In this way, rare capsules, when they have prominent outputs, are still preserved and routed to the next layer.

Next, we route all b capsules in the PVC layer based on the similarity between them and the static capsule layer outputs. We initialize the routing coefficients as $\alpha_{ij} = 0$, equally to the b capsules in the primary variable capsule layer. Subsequently, they are iteratively refined based on the *agreement* between the current SC layer outputs $\mathbf{v}_j$ and the PVC layer outputs $\mathbf{u}_i$. The agreement, in this case, is measured by the dot product, $f_{ij} \leftarrow \mathbf{u}_i \cdot \mathbf{v}_j$, and the routing coefficients are adjusted with f_{ij} accordingly. If a capsule u in the PVC layer has a strong agreement with a capsule j in the SC layer, then f_{ij} will be positively large, whereas if there is strong disagreement, then f_{ij} will be negatively large. Subsequently, the sum of vectors $\mathbf{u}_i$ is weighted by the updated β_{ij} to calculate $\mathbf{s}_j$, which is then squashed to update $\mathbf{v}_j$.

6.2.4 The Code Capsules layer

The Code Capsule (CC) layer in TreeCaps outputs the vector representations for the code $\mathbf{X_{cc}} \in \mathbb{R}^{N_{cc} \times D_{cc}}$, where D_{cc} is the dimensionality of each code capsule and N_{cc} is fixed with respect to a specific code learning task. Since the outputs of the Secondary Capsule layer $\mathbf{X_{sc}} \in \mathbb{R}^{N_{sc} \times D_{sc}}$, where N_{sc} is also fixed, It then produces the needed final capsule outputs $\mathbf{X_{cc}}$.

In the following subsections, we explain how we set N_{cc} and train the TreeCaps models for different code learning tasks.

6.2.4.1 Code (Functionality) Classification

This task is to, given a piece of code, classify the functionality class it belongs to. We want N_{cc} capsules in the CC layer, each of which corresponds to a functionality class of code that appeared in the training data. As such, we let $Ncc = \kappa$, where κ is the number of functionality classes. We calculate the probability of the existence of each class by obtaining L_2 norm of each capsule output vector. We use the margin loss [111] as the loss function during training.

6.2.4.2 Function (Method) Name Prediction

This task is to, given a piece of code (without its function header), predict a meaningful name that reflects the functionality of the code. For this task, following [10]'s prediction approach, we let N_{cc} of the CC layer be 1, and the output of the only capsule represents the vector for the given piece of code. In this case, the output capsules of the CC layer has the shape of $\mathbf{X_{cc}} \in \mathbb{R}^{1\times D_{cc}}$, which is also the code vector that represents for the code snippet $\boldsymbol{C}$, denoted as $\boldsymbol{v_C}$. The vector embeddings of the function are learn-able parameters, formally defined as $functions_vocab \in \mathbb{R}^{|L|\times D_{cc}}$, where L is the set of function names found in the training corpus. The embedding of $function_i$ is row i of $functions_vocab$. The predicted distribution of the model $q\left(l\right)$ is computed as the (softmax-normalized) dot product between the context vector $\boldsymbol{v_C}$ and each of the function embeddings:

$$for\, l_i \in L:\; q\left(l_i\right) = \frac{\exp(\boldsymbol{v_C}^T \cdot functions_vocab_i)}{\sum_{l_j \in L} \exp(\boldsymbol{v_C}^T \cdot functions_vocab_j)} \tag{6.2}$$

where $q\left(l_i\right)$ is the normalized dot product between the vector of l_i and the code vector $\boldsymbol{v_C}$, i.e., the probability that a function name l_i should be assigned to the given code snippet $\boldsymbol{C}$. We choose l that gives the maximum probability for the snippet $\boldsymbol{v_C}$. For training the network, we use the cross-entropy as the loss function.

6.3 Empirical Evaluation

General Settings. We use SrcML [31] to parse source code into ASTs;[1] we also use another parser PycParser[2] used by TBCNN and ASTNN to ensure a fair comparison and evaluate the effects of different parsers. For the parameters in our TBCNN layer, we follow [83] to set the the size of type embeddings = 30, and size of text embeddings = 50 and the number of convolutional steps $M = 8$. For the capsule layers, we set $N_{sc} = 100$, $D_{sc} = 16$ and $D_{cc} = 16$. Table 6.1 summarizes the major hyper-parameters in TreeCaps (both DRSW and VTS) that are used with various datasets. We have used Tensorflow libraries to implement TreeCaps. To train the models, we have used the Rectified Adam

[1] `https://www.srcml.org/`, 400+ node types for supporting multiple programming languages. We chose SrcML because (1) it provides unified AST representations for various languages such as C/C++/Java in our datasets, and (2) it provides an extension SrcSlice (`https://github.com/srcML/srcSlice`) to help identify code dependencies and construct the graphs needed for GGNN, which is a baseline in our evaluation.

[2] `https://github.com/eliben/pycparser/`, about 50 node types for C.

(RAdam) optimizer [74] with an initial learning rate of 0.001 subjected to decay, on an Nvidia Tesla P100 GPU.

Table 6.1: Experimental Settings

Hyper-parameters	**Notation**	**Value**
Dimension of the embedding for each node type in TBCNN	D_{type}	30
Dimension of the embedding for each node token in TBCNN	D_{token}	50
Number of the convolutional layers in TBCNN	M	8
Number of capsules in the Primary Variable Capsule (PVC) layer	N_{pvc}	Depend on the input tree size
Dimension of each capsule in the PVC layer	D_{pvc}	8
Number of capsules in the Secondary Capsule (SC) layer	N_{sc}	100
Dimension of each capsule in the SC layer	D_{sc}	16
Number of capsules in the Code Capsule (CC) layer	N_{cc}	Depend on the target learning task
Dimension of each capsule in the CC layer	D_{cc}	16
Number of iterations in routing	r	3
Number of top-a output capsules chosen from PVC to initialize for SC	a	N_{sc}
Number of capsules to route from PVC to SC	b	N_{pvc}

6.3.1 Set up for Code Classification

Datasets, Metrics, and Models. We use datasets in two different programming languages. The first Sorting Algorithms (SA) dataset is from [23], which contains 10 algorithm classes of 1000 sorting programs written in `Java`. The second OJ dataset is from [83], which contains 52000 `C` programs of 104 classes. We split each dataset into training, testing, and validation sets by the ratios of 70/20/10. We use the same classification accuracy metric as [83] for comparing classification results.

We compare TreeCaps with other techniques applied for the code classification task, such as Code2vec, TBCNN, ASTNN, GGNN, etc. Since TBCNN [83] and ASTNN [134] use PycParser to parse code into AST, we also compare TreeCaps with all the baselines by using both PycParser and SrcML. We also include a token-based baseline by treating source code as a sequence of tokens and using the 2-layer Bi-LSTM to process the sequence of tokens. We follow [3] to use the concatenation of embeddings of node types and texts (tokens) for node initialization and representation. We also include an ***ablation study*** to measure the impact of different combinations of node initialization and representation.

Code Classification Results. As shown in Table 6.2, TreeCaps models, especially TreeCaps-VTS, have the highest classification accuracy when combining node type and node token information, for both of the SA and OJ datasets. When only node token information is used, the simpler 2-layer Bi-LSTM models may achieve higher accuracy. The OJ dataset also shows that the choice of

Table 6.2: Performance in Code Functionality Classification compared. A '-' means that the model is not suited to use the relevant node representation or the parser and thus not evaluated.

Model	SA Dataset			OJ Dataset					
Parser	SrcML			PycParser			SrcML		
Initial Info	Type	Token	Combine	Type	Token	Combine	Type	Token	Combine
2-layer Bi-LSTM	-	81.83	-	-	83.51	-	-	83.51	-
Cod2vec	-	-	80.44	-	-	86.21	-	-	80.15
TBCNN	78.09	71.23	82.02	94.0	78.34	95.06	81.15	71.15	83.90
GGNN	82.12	74.25	83.81	-	-	-	85.23	72.23	85.89
ASTNN	-	-	84.32	-	-	98.2	-	-	85.32
Treecaps-DRSW	83.15	74.56	84.57	96.22	79.21	96.74	83.59	77.59	87.77
Treecaps-VTS	84.60	78.15	**85.43**	96.48	79.85	**98.43**	83.40	78.45	**88.40**

a parser affects the performance significantly. The models using PycParser all achieve higher accuracy than the models using SrcML. This is due to the reason that ASTs generated by PycPaser have only around 50 node types, while SrcML has more than 400 node types, which makes it harder for the networks to learn. Across the datasets, The TreeCaps-VTS performs consistently the best in terms of the F1 measure among the baselines under different settings.

6.3.2 Set up for Function Name Prediction

Datasets, Metrics, and Models. We have used the datasets from [9] that contain three sets of Java programs: Java-Small (700k samples), Java-Med (4M samples), and Java-Large(16M samples). We measure prediction performance using precision (P), recall (R), and F1 scores over the sub-words in generated names, following the metrics used by [10, 41]. For example, a predicted name `result_compute` is considered to be an exact match of the ground-truth name called `computeResult`; predicted `compute` has full precision but only 50% recall; and predicted `compute_model_result` has full recall but only 67% precision.

We compare TreeCaps to the following baselines applied to the function name prediction task: Code2vec, TBCNN, GGNN, and a neural machine translation baseline that reads the input source code as a stream of tokens by using a 2-layered bidirectional encoder-decoder LSTMs (split tokens) with global attentions. The ASTNN is not designed for this task, so we exclude it in this evaluation.

Function Name Prediction Results. As seen in Table 6.3, TreeCaps-DRSW and TreeCaps-VTS are comparable in term of F1 score, although the TreeCaps-VTS is slightly better. The TreeCaps models also achieve comparable or better results than all other models for most of the settings. In particular, TreeCaps are comparable or better than GGNN but without the need of

additional code dependency analysis for constructing graphs.

Table 6.3: Performance of TreeCaps and the baselines for Function (Method) Name Prediction

Model	java-small			java-med			java-large		
Metric	P	R	F1	P	R	F1	P	R	F1
2-layer BiLSTM	40.02	31.84	35.46	43.05	41.69	42.31	48.34	40.27	44.63
TBCNN	28.89	21.67	22.56	35.98	33.41	35.23	41.15	37.29	38.33
Code2vec	23.35	22.01	21.36	36.43	27.93	31.89	44.24	38.25	41.56
GGNN	42.25	35.25	35.25	53.14	**44.59**	47.31	50.18	44.2	46.23
TreeCaps-DRSW	42.15	37.49	37.04	47.11	41.15	43.28	49.37	46.58	47.85
TreeCaps-VTS	**43.52**	**38.56**	**38.51**	**55.35**	42.98	**47.89**	**50.88**	**47.01**	**48.34**

6.3.3 Model Analysis

To better understand the importance of the different components of our approach, we evaluate the effect of various aspects of the TreeCaps models. We use the code classification task on the OJ Dataset for the experiments in this subsection.

6.3.3.1 Robustness of Models

We measure the robustness of each model by applying the semantically-preserving program transformations to the OJ Datasets test set. We follow [105, 118] to transform programs in three ways that change code syntax but preserve code functionality: (1) Variable Renaming (VN), a refactoring transformation that renames a variable in code, where the new name of the variable is taken randomly from a set of variable vocabulary in the training set; (2) Unused Statement (US), inserting an unused string declaration to a randomly selected basic block in the code; and (3) Permute Statement (PS), swapping two independent statements (i.e., with no dependence) in a basic block in the code.

The OJ test set is thus transformed into a new test set. We then examine if the models make the same predictions for the programs after transformation as the prior predictions for the original programs. We use ***percentage of predictions changed*** (PPC) as the metric previously used by [105, 118, 132] to measure the robustness of the models. Formally, suppose P denotes a set of test programs, a semantic-preserving program transformation T that transforms P into a set of transformed programs $P' = \{p' = T(p)|p \in P\}$, and a source code learning model M that can make predictions for any program p: $M(p) = l$, where $l \in L$ denotes a predicted label for p according to a set of labels L learned

by M, we compute the percentage of predictions changed as follows.

$$PPC = \frac{|\{p' \in P' | M(p) \neq M(p')\}}{|\{p' \in P'\}|} * 100. \tag{6.3}$$

The lower PPC values for M suggest higher robustness since they can maintain more of correct predictions with respect to the transformation. As shown in Table 6.4, TreeCaps-VTS is the most robust model against the program transformations. Even though more kinds of program transformations could be applied to evaluate model robustness in our future work, the current analysis gives us the confidence that TreeCaps can be more robust against attacks via such adversarial examples [16, 108].

Table 6.4: Model robustness, measured as percentage of predictions changed wrt. semantic-preserving program transformations. The lower the more robust.

	Variable Renaming	Unused Statement	Permute Statement	Average
Code2vec	13.45%	19.42%	18.56%	17.04%
TBCNN	10.16%	16.33%	15.43%	13.97%
ASTNN	10.43%	13.82%	12.14%	12.23%
GGNN	9.34%	11.89%	10.48%	10.57%
TreeCaps-DRSW	8.46%	11.72%	10.41%	10.19%
TreeCaps-VTS	**8.15%**	**11.08%**	**8.87%**	**9.37%**

6.3.3.2 Comparison between the Two Routing Algorithms

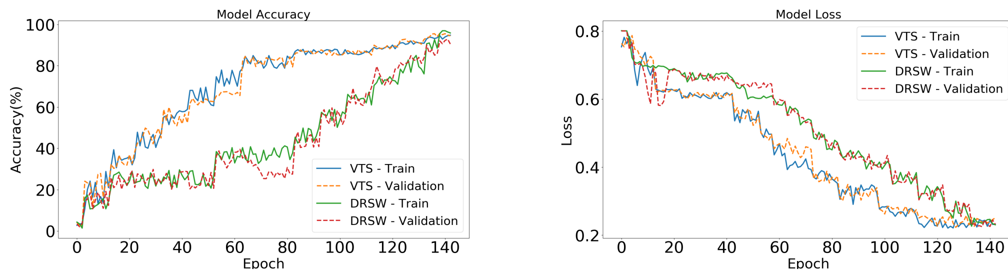


Figure 6.2: Comparisons between the Two Routing Algorithms

Figure 6.2 shows the comparisons between the Dynamic Routing algorithm with Shared Weights (DRSW) and Variable-to-Static Routing algorithm (VTS) (cf. Section 6.2.3). There are two main observations: (1) when DRSW is used, the loss decreases much slower than when VTS is used (in the right plot); and (2) VTS improves validation accuracy much faster than DRSW (in the left chart). An explanation for this is that DRSW has to learn an additional shared transformation matrix $\mathbf{W_s}$, resulting in slower convergence due to a larger number of parameters to be learned.

6.3.3.3 Effect of the Secondary Capsule (SC) Layer

In our design, we choose to route the dynamic number of capsules in the PVC layer into a fixed number of capsules in the SC layer. An issue is that we can directly route the capsules in the PVC layer to the CC layer using either the DRSW or the VTS algorithm (without the intermediate SC layer). In this section, we conduct an experiment to demonstrate the need for the SC layer to combine the information of the capsules in the PVC layer into a fixed number of capsules representing different aspects of the input trees. We remove the SC Layer and do the routing from the PVC layer directly to the CC layer. We apply the dynamic routing algorithm over the PVC layer with a shared transformation matrix for the DRSW algorithm and get a fixed number of N_{cc} capsules that represent each class. For the VTS algorithm, we pick the number of top a capsules with the maximum $L2$ norms equal to the number of classes, so $a = \kappa = N_{cc}$, and conduct the training for code functionality classification task on both the SA and OJ datasets.

The accuracy of VTS for both the dataset is significantly lower than the DRSW algorithm in Table 6.5, since there is no child-parent relationship learned from any transformation matrix in the VTS. On the other hand, the DRSW can still retain the accuracy of more than 60%, because there is still a transformation matrix that can learn the spatial relationship between capsules. This means that the SC layer is a necessary component to maintain important spatial correlations in the AST.

Table 6.5: Performance of TreeCaps after removing the Secondary Capsule Layer

Model	SA Dataset	OJ Dataset
DRSW w/o SC	62.3%	66.2%
VTS w/o SC	43.5%	51.6%

6.3.3.4 Effect of the Number of Capsules (N_{sc}) in the Secondary Capsule (SC) Layer

Table 6.6 shows the results of this analysis. First, we choose $N_{sc} = 1$, which means that there is only one capsule in the SC layer. The accuracy is low, i.e., 32.57%. This means that only one capsule is not enough to capture features in the PVC layer. Next, we keep increasing the number of capsules in the SC layer by about 20 and retrained the models for each of the choices. The results show that $N_{sc} = 100$ is a reasonable choice and provide the best results. For

$N_{sc} = 120$ and $N_{sc} = 140$, the performance gap does not differ significantly, so that it is not necessary to choose a larger N_{sc}.

Table 6.6: Effect on N_{sc} on the classification accuracy for the OJ Dataset

N_{sc}	1	20	40	60	80	100	120	140
Classification Accuracy	32.57	81.49	86.82	84.17	86.79	88.40	87.93	88.32

6.3.3.5 Effect of the Variable-To-Static (VTS) Routing Algorithm

We investigate the effect of the variable-to-static routing algorithm by replacing it with Dynamic Max Pooling (DMP). Since there is no alternative approach existing in the literature for routing a variable set of capsules to a static set of capsules, we compare the proposed routing algorithm with dynamic pooling. The output of the PVC layer, $\mathbf{X_{pvc}} \in \mathbb{R}^{N_{pvc} \times D_{pvc}}$ consists of a variable component, N_{pvc}. Using dynamic max-pooling (DMP) across all the N_{pvc} capsules will result in one output capsule, $\mathbf{X_{dmp}} \in \mathbb{R}^{1 \times D_{pvc}}$. Since $\mathbf{X_{dmp}}$ has no variable components across the training samples, it can now be routed to the code capsules using the dynamic routing algorithm. However, it should be noted that DMP destroys the spatial and dependency relationships between the capsules; we use DMP here only for comparison purposes.

As summarized in Table 6.7, DMP yields a considerably lower accuracy of 78.58% than our routing algorithm by a significant margin of 9.82%, establishing the effectiveness of our proposed algorithm.

6.3.3.6 Effect of the Dimension of Capsules (D_{cc}) in the Code Capsule (CC) Layer

The instantiation parameters D_{cc} of the Code Capsule layer acts as the dimensionality of the vector representations of source code. If the dimensionality of the representation is higher than required, it can introduce sparsity and/or correlations between the instantiation parameters, reducing the classification accuracy. On the contrary, if the dimensionality of the latent representation is too low, it may not be sufficient to capture the variations in source code, leading to under-representation, reducing the classification accuracy. Hence, in an attempt to identify a suitable value for D_{cc} for source code classification, we investigate the effect of D_{cc} in the accuracy. As summarized in Table 6.7, we observed that the most suitable value was $D_{cc} = 16$ for the OJ Dataset.

Table 6.7: Effect of different model variants for the OJ Dataset

Model Variant	**Classification Accuracy**
Variable-to-Static Routing Algorithm → Dynamic Pooling	78.58%
Dimension of capsules in SC layer → $D_{sc} = 4$	79.90%
$D_{cc} = 8$	83.15%
$D_{cc} = 12$	82.31%
$D_{cc} = 16$	88.40%
$D_{cc} = 20$	86.20%
$D_{cc} = 24$	85.98%

6.3.3.7 Sensitivity to Input Code Sizes

We examined how the size of a function body (i.e., lines of code (LOC)) affects the performance of the function name prediction task. We used the results for the Java-Large dataset in this examination. As shown in Figure 6.3, the TreeCaps-VTS, TreeCaps-DRSW, and GGNN are comparable to each other. We can observe that the TreeCaps-VTS is slightly better than the other two. All models give their best results for short snippets of code, i.e., less than 3 lines. As the size of a function increases, the performance of all examined models starts to decrease slowly for the size of 10 and above. When reaching the size of 37 and above, we can see the significant drop-down for all of the models.

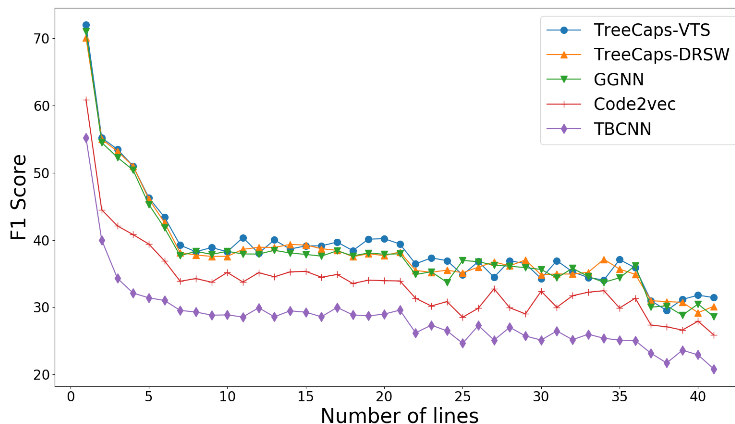


Figure 6.3: Sensitivity of Input Code Sizes

6.4 Conclusion

We proposed TreeCaps, a novel neural network that incorporates the capsules theory into Tree-based CNN for a better representation learning of code. To handle dynamic numbers of capsules produced from TBCNN, we propose two methods to route the capsules in the Primary Variable Capsule layer to fixed number of capsules in the Secondary Capsule layer. We are the first to re-purpose capsule networks over syntax trees to learn code without the need for explicit semantics analysis. We also showed that TreeCaps can perform well in different use-case scenarios of code processing, such as code classification and method name prediction. It is our belief that the new method can be applied to other valuable software engineering tasks, such as bug localization, code clone detection, etc.

Since TreeCaps is based on the capsule networks, it inherits a few limitations of capsule networks. The first limitations is the high computational complexity in comparison to CNNs, our novel VTS routing method is designed to compensate such limitation and produce comparable results and faster training time comparing with the DR-SW routing method. The second limitation of capsule networks is that it lacks an explainability method, i.e. understand how a capsule represents for which feature of source code and how the capsules are correlated. We intend to extend our work to a method to identify the piece of code that is represented by the capsules and compare them with program dependencies identified by program analysis techniques, to evaluate the effectiveness of TreeCaps as an embedding generating technique.

Our empirical evaluations have shown that TreeCaps can outperform existing code learning models (e.g., Code2vec, TBCNN, ASTNN, GGNN, etc.) for two different program comprehension tasks (e.g., code functionality classification and function name prediction) on C/C++/Java programs. It is our belief that the new method can be applied to other valuable

Chapter 7

Related work

In this chapter, we review some existing work in using machine learning on source code and the applications in solving a wide range of software engineering tasks. There have been a number of methods proposed to address learning on source code, but it is not clear how these methods are related to each other. This review thus tries to organize the existing work and layout an overall picture of the source code learning with its possible solutions.

7.1 Learning Code as Natural Languages Tokens

Mining a large corpus of open-source Java projects on the repositories, Hindle et al. [54] showed that programming languages are largely in common to natural languages in terms of probabilistic predictability (i.e., low entropy) of next tokens, hence statistical methods apply well to model repetitiveness in source code, and those language models can be used for the code suggestion task in IDEs. Furthermore, it has been established that programmers tend to focus on code local to their context, hence the locality can be exploited by 'caches' while keeping both the long and short-term memory of the sequences processed. For example, Hellendoorn et al. [51] employed both n-grams (with and without cache) and LSTM NN to train statistical models and demonstrated that such straight application of NN to sequences do not necessarily enhance the accuracy greatly compared to the cached n-gram models. In a survey, Allamanis et al. [2] further categorized the related research directions in this area. Our hierarchical learning technique (chapter 2) treat the source code as a sequence of tokens and learn the combination of different code levels in a hierarchy style, which

has been proved to perform well in many SE tasks.

7.2 Learning Programs with Structures

It is known that code is structured (e.g., nested in syntax) and programmers do not read the code from the beginning to the end. Static program analysis tools rely on AST and program dependence graphs to capture such information. However, exist tools for translating code among specific languages (e.g., Java2CSharp) are mostly rule-based, rather than statistics-based [62]. Boccardo et al. [17] proposed to use neural networks for learning program equivalence based on callgraphs, the accuracy was not too good because it does not take into account code syntax structures. Alexandru et al. [1] proposed to analyze only the revised source files from a Git repository, instead of parsing or analyzing those unchanged revisions, reducing redundancies in analyzing the evolution of source code of different programming languages. On learning from code syntax structures, TBCNN was first proposed by Mou et al. [81], which designed a set of fixed-depth subtree filters sliding over an entire AST to extract structural information of the tree. They also proposed "continuous binary trees" and applied dynamic pooling [113] to deal with varying numbers of children of AST nodes. Our DTBCNN (chapter 3) extends the TBCNN by adding the dependency information into the AST so that the source code semantic can be captured in a more explicit way.

Capsule networks [55, 111] use dynamic routing to model spatial and hierarchical relations among objects in an image. The techniques have been successfully applied to different tasks, such as computer vision, character recognition, and text classification [60, 106, 136]. None of the studies has considered complex tree data as input. Capsule Graph Neural Network [135] has been recently proposed to classify biological and social network graphs, yet, has not been applied to trees for programming languages processing yet. Our TreeCaps (chapter 6) is also a structure-based learning approach, where the learning algorithm from the capsule theory has been proved to capture the structural information better than TBCNN.

7.3 Combining syntactical and semantic information

For semantic information such as program dependence graphs, Allamanis et al. [5] used GGNN to unify the information with AST. Using open-source C# projects they have demonstrated significant improvement in predicting the correct or misused names. The evaluation shows our Bi-DTBCNN approach performs better than GGNN for our algorithm classification tasks. A possible reason is we embed *all* contextual information on the AST whilst GGNN relies on how certain semantic relations that can be derived from AST are explicitly encoded.

7.4 Bi-lateral representations for cross-language learning

Cross-language learning representation structures are typically bilateral. Studies on sentence comparisons and translations in NLP involve variants of bilateral structures as shown by Wang et al. [119]. Among them, Bromley et al. [21] pioneered "Siamese" structures to join two subnetworks for written signature comparison. He et al. [50] also use such structures to compute sentence features at multiple levels of granularity. Yin and Neubig [127] and Oda et al. [98] used Seq2Seq NN to perform code generation from one programming language to another. Much work has utilized various statistical language models for tokens [91], phrases [62, 92, 93], or APIs [25, 39, 89, 90, 104, 137, 138]. A few studies also used word embedding for API mapping and migration (e.g., [48, 49, 94, 104, 123]), but our work does not need large number of manually specified parallel corpora or mapping seeds. Tools for translating code among specific languages in practice (e.g., Java2CSharp [29]) also often dependent on manually defined rules specific to the grammars of individual languages, while our approach alleviates the need of language-specific rules.

MAM [138] and StaMiner [89] rely on the availability of bilingual projects that implement the same functionality in two or more languages. DeepAM [49] requires many similar text descriptions across programs written in different programming languages whose availability can affect the mapping results. Api2Api [94] requires many API mapping seeds from Java2CSharp [29]) to map APIs. The idea of our SAR approach (chapter 4) is most similar to Api2Api,

while we combine seed-based and unsupervised domain adaptation techniques to reduce the need of mapping seeds.

7.5 Intepretability for Code Processing

Interpretability is important for software mining and analysis in general [37]. In domains other than software engineering, various techniques have been proposed to interpret machine learning results in different ways, such as by inverting trained CNN models to project outputs through hidden neurons to input image pixels and visualizing the hidden neurons using inputs [130], by quantifying the effects of composing different representations of meanings in English sentences to visualize compositionality of RNN and LSTM models [68], and by perturbing input images to evaluate its impact on black box neural networks [42]. Autofocus (chapter 5) is a technique to visualize important parts of the source code to understand how the code processing techniques work from the human understanding perspective.

Chapter 8

Conclusion

In this chapter, we summarize the findings of this thesis and point out some future research directions.

8.1 Summary

We have shown that the combination of deep learning and static code analysis can greatly increase the performance of source code processing. In chapter 2, we present a hierarchical representation of source code and apply a simple learning technique borrowed from Natural Language Processing. We show that with such hierarchical representation and only a simple learning technique, the performance was quite well on tasks such as code clone detection, code classification. In Chapter 3, we present a method to enrich the representation of code with the information from static code analysis, we call it Dependency Tree. The tree-based deep learning technique can apply directly to the Dependency Tree. We show that our novel Dependency Tree outperforms the other alternative methods for cross-language code classification. In Chapter 4, we tackle a challenging task to map code fragments (i.e., APIs) across languages without any labeled data. In this work, we use static code analysis to extract the API calling sequences and apply an advanced deep learning method (Generative Adversarial Network) to map the APIs. This is an important stepping stone towards the goal to automatically translate a program from one language to another, which is essential for the process to migrate (or upgrade) legacy codebases to a newer version. In Chapter 5, we present a method to interpret how deep learning understand source code, we show that the attention mechanism can shred some light towards the goal to understand how machine learning understand source code

Although the goal of this thesis is to present how the combination of static code analysis and deep learning can greatly improve the performance of software analytics, we realize for certain tasks, expensive static code analysis may not be necessary, and the accuracy of such analysis can be low, which might add noise to the code representation. As such, in Chapter 6, we propose TreeCaps, a novel method that fuses the idea of capsules to combine information of source code without any static code analysis.

8.2 Future Directions

In the future, we plan to further optimize the current solutions for source code processing.

Unsupervised Learning for Source Code Processing Unsupervised learning allows us to learn useful representations from large unlabeled corpora. Unsupervised learning has been received a huge interest in computer vision [38, 66, 97, 101], and natural language processing [13, 19, 32, 53]. The idea of self-supervision has recently become popular where representations are learned by designing learning objectives that exploit labels that are freely available with the data (without any human effort to label data). Given the huge amount of available source code on hosting platforms, such as Github, Bitbucket, Gitlab, etc., unsupervised learning is likely to be applicable in the source code model too. However, the effort for this research direction is little. If we can design the model that leverages the huge amount of available data from source code, it can be very beneficial.

Using Deep Learning for Source Code Translation We have shown that deep learning is able to map code fragments across languages. However, program translation is a more challenging task than the API mapping. The most challenging part of program translation is to collect the parallel data, which is essential to build a translation model.By leveraging the recent advances in unsupervised learning, the program translation can be build with little effort to collect the parallel data.

Interpretability of Source Code Model on More Challenging Tasks We proposed a method to interpret how the model understands the code classification task in Chapter 5. The interpretability on source code processing

need to be tackled on more challenging tasks, such as variable name prediction, type prediction, program synthesis, etc.

Bibliography

[1] Carol V. Alexandru, Sebastiano Panichella, and Harald C. Gall. Reducing redundancies in multi-revision code analysis. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 148–159, 2017.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81:1–81:37, July 2018.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, 2018.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

[7] 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*, pages 207–216. IEEE Press, 2013.

[8] Hakam W Alomari, Michael L Collard, Jonathan I Maletic, Nouh Alhindawi, and Omar Meqdadi. srcslice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process*, 26(11):931–961, 2014.

[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Programming Languages*, 3(POPL):40:1–40:29, January 2019.

[11] David Alvarez-Melis and Tommi S. Jaakkola. Towards robust interpretability with self-explaining neural networks. In *NeurIPS*, 2018.

[12] Anonymous Authors. Anonymous paper. In *Anonymous Publication Venue*.

[13] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[15] Alexandre Bérard, Christophe Servan, Olivier Pietquin, and Laurent Besacier. MultiVec: a Multilingual and Multilevel Representation Learning Toolkit for NLP. In *10th Language Resources and Evaluation Conference (LREC)*, May 2016.

[16] Pavol Bielik and Martin Vechev. Adversarial robustness for code. *arXiv preprint arXiv:2002.04694*, 2020.

[17] Davidson Boccardo, Tiago Monteiro Nascimento, Charles Prado, Luiz Fernando Rust da Costa Carmo, and Raphael Machado. Program equivalence using neural networks. In *5th International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems, At Boston, United States*, 10 2010.

[18] Jürgen Börstler. Feature-oriented classification for software reuse. In *SEKE'95, The 7th International Conference on Software Engineering and Knowledge Engineering, June 22-24, 1995, Rockville, Maryland, USA, Proceedings*, pages 204–211. Knowledge Systems Institute, 1995.

[19] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Józefowicz, and Samy Bengio. Generating sentences from a continuous space. In Yoav Goldberg and Stefan Riezler, editors, *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016*, pages 10–21. ACL, 2016.

[20] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Quantify the time and cost saved using reversible debuggers. Technical report, Cambridge Judge Business School, 2012.

[21] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS)*, pages 737–744, 1993.

[22] Nghi D. Q. Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 33–36, 2018.

[23] Nghi D. Q. BUI, Yijun YU, and Lingxiao JIANG. Bilateral dependency neural networks for cross-language algorithm classification. In *IEEE/ACM International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[24] Silvio Cesare and Yang Xiang. A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*, pages 721–728. IEEE Computer Society, 2010.

[25] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019.

[26] 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, 2018.

[27] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Mining revision histories to detect cross-language clones without intermediates. In *ASE*, pages 696–701, 2016.

[28] K. L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

[29] codejuicer. Java2csharp: a maven plugin to convert java classes to c#, 2017. Last commit in Sep 19, 2017.

[30] Michael L Collard, Michael J Decker, and Jonathan I Maletic. Lightweight transformation and fact extraction with the srcML toolkit. In *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 173–184, 2011.

[31] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance (ICSM)*, pages 516–519, 2013.

[32] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 670–680. Association for Computational Linguistics, 2017.

[33] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *CoRR*, abs/1710.04087, 2017.

[34] James R. Cordy and Chanchal K. Roy. Tuning research tools for scalability and performance: The nicad experience. *Sci. Comput. Program.*, 79:158–171, 2014.

[35] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. Deep learning on code with an unbounded vocabulary. In *Machine Learning for Programming (ML4P) Workshop at Federated Logic Conference (FLoC)*, 2018.

[36] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[37] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Explainable software analytics. In *ICSE: New Ideas and Emerging Results*, pages 53–56. ACM, 2018.

[38] Carl Doersch, Abhinav Gupta, and Alexei A. Efros. Unsupervised visual representation learning by context prediction. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1422–1430. IEEE Computer Society, 2015.

[39] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. Unsupervised learning of api aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 745–759. ACM, 2019.

[40] Evans Data Corporation. Global developer population and demographic study. `http://evansdata.com/reports/viewRelease.php?reportID=9`, 2019.

[41] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

[42] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *ICCV*, pages 3429–3437, 2017.

[43] Yaroslav Ganin and Victor S. Lempitsky. Unsupervised domain adaptation by backpropagation. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 1180–1189, 2015.

[44] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4):56:1–56:36, 2017.

[45] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *DSAA*, pages 80–89, 2018.

[46] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring likely mappings between apis. In *ICSE*, pages 82–91. IEEE, 2013.

[47] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *CoRR*, abs/1406.2661, 2014.

[48] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *FSE*, pages 631–642, November 13-18 2016.

[49] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3675–3681, August 19-25 2017.

[50] Hua He, Kevin Gimpel, and Jimmy J. Lin. Multi-perspective sentence similarity modeling with convolutional neural networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[51] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *ESEC/FSE*, pages 763–773, 2017.

[52] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA, 2017. ACM.

[53] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. In Kevin Knight, Ani Nenkova, and Owen Rambow, editors, *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 1367–1377. The Association for Computational Linguistics, 2016.

[54] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, pages 837–847. IEEE, 2012.

[55] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *International Conference on Learning Representations*, May 2018.

[56] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, Aug 1995.

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[58] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, Mar 2002.

[59] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *2012 19th Working Conference on Reverse Engineering*, pages 387–391, Oct 2012.

[60] Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Jathushan Rajasegaran, Suranga Seneviratne, and Ranga Rodrigo. TextCaps: Handwritten character recognition with very small datasets. In *IEEE Winter Conference on Applications of Computer Vision*, pages 254–262, 2019.

[61] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 81–92, 2009.

[62] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 173–184, New York, NY, USA, 2014. ACM.

[63] Tom Kenter and Maarten De Rijke. Short text similarity with word embeddings. In *24th ACM International on Conference on Information and Knowledge Management (CIKM)*, pages 1411–1420, 2015.

[64] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Jun Cai, James Wexler, Fernanda Viegas, and Rory Abbott Sayres. Interpretability

beyond feature attribution: Quantitative testing with concept activation vectors (TCAV). In *ICML*, 2018.

[65] Youngjoong Ko. A study of term weighting schemes using class information for text classification. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 1029–1030, New York, NY, USA, 2012. ACM.

[66] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1558–1566. JMLR.org, 2016.

[67] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *IEEE International Conference on Software Quality, Reliability and Security*, pages 318–328. IEEE, 2017.

[68] Jiwei Li, Xinlei Chen, Eduard H. Hovy, and Dan Jurafsky. Visualizing and understanding neural models in NLP. In *NAACL HLT*, 2016.

[69] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, November 2016. arXiv: 1511.05493.

[70] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, November 2016.

[71] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[72] Percy Liang, Ben Taskar, and Dan Klein. Alignment by agreement. In *the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (HLT-NAACL)*, pages 104–111. Association for Computational Linguistics, 2006.

[73] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *CoRR*, abs/1712.03201, 2017.

[74] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

[75] Thang Luong, Hieu Pham, and Christopher D Manning. Bilingual word representations with monolingual quality in mind. In *1st Workshop on Vector Space Modeling for Natural Language Processing (VS@HLT-NAACL)*, pages 151–159, 2015.

[76] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1412–1421, 2015.

[77] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[78] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[79] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NIPS)*, pages 3111–3119, 2013.

[81] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293, February 12-17 2016.

[82] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, pages 1287–1293, February 12-17 2016.

[83] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence*, 2016.

[84] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2786–2792, 2016.

[85] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, Sep. 2018.

[86] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391, 2013.

[87] Bui D. Q. Nghi, Yijun Yu, and Lingxiao Jiang. Bilateral dependency neural networks for cross-language algorithm classification. In *SANER*, pages 422–433, 2019.

[88] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *ASE*, pages 457–468, 2014.

[89] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 457–468, 2014.

[90] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning of API mappings for language migration. In *ICSE*, pages 618–619, May 31 - June 07 2014.

[91] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *ESEC/FSE*, pages 651–654, August 18-26 2013.

[92] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (T). In *ASE*, pages 585–596, November 9-13 2015.

[93] Anh Tuan Nguyen, Zhaopeng Tu, and Tien N. Nguyen. Do contexts help in phrase-based, statistical source code migration? In *ICSME*, pages 155–165, October 2-7 2016.

[94] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring API embedding for API usages and applications. In *39th International Conference on Software Engineering (ICSE)*, pages 438–449, 2017.

[95] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.

[96] R. Nix and J. Zhang. Classification of android apps and malware using deep neural networks. In *International Joint Conference on Neural Networks*, pages 1871–1878, May 2017.

[97] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI*, volume 9910 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2016.

[98] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 574–584. IEEE Computer Society, 2015.

[99] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1916–1920. IEEE, 2015.

[100] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[101] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2536–2544. IEEE Computer Society, 2016.

[102] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM)*, pages 547–553, October 28-30 2015.

[103] F. Peters, T. Tun, Y. Yu, and B. Nuseibeh. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering*, 1(1):1–1, 2018.

[104] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. Statistical migration of API usages. In *ICSE*, pages 47–50, May 20-28 2017.

[105] Md Rabin, Rafiqul Islam, and Mohammad Amin Alipour. Evaluation of generalizability of neural program analyzers under semantic-preserving transformations. *arXiv preprint arXiv:2004.07313*, 2020.

[106] Jathushan Rajasegaran, Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Suranga Seneviratne, and Ranga Rodrigo. DeepCaps: Going deeper with capsule networks. In *Computer Vision and Pattern Recognition*, 2019.

[107] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society.

[108] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

[109] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2013.

[110] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[111] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Conference on Neural Information Processing Systems*, pages 3856–3866, Long Beach, CA, 2017.

[112] Peter H. Schönemann. A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10, Mar 1966.

[113] Richard Socher, Eric H. Huang, Jeffrey Pennington, Andrew Y. Ng, and Christopher D. Manning. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *24th International Conference on Neural Information Processing Systems (NIPS)*, pages 801–809, December 12-14 2011.

[114] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1139–1147, 2013.

[115] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

[116] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *Comput. J.*, 54(7):1049–1066, 2011.

[117] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. On the localness of software. In *FSE*, pages 269–280, 2014.

[118] K Wang and Zhendong Su. Learning blended, precise semantic program embeddings. *ArXiv, vol. abs/1907.02136*, 2019.

[119] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4144–4150, August 19-25 2017.

[120] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3034–3040, 2017.

[121] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, Oct 2018.

[122] Chao Xing, Dong Wang, Chao Liu, and Yiye Lin. Normalized word embedding and orthogonal transform for bilingual word translation. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 1006–1011, 2015.

[123] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of api migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 335–346. IEEE Press, 2019.

[124] S. Yan, B. Shen, W. Mo, and N. Li. Transfer learning for cross-platform software crowdsourcing recommendation. In *24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 269–278, Dec 2017.

[125] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 17–26, 2015.

[126] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 121–128, March 2013.

[127] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In Regina Barzilay and Min-Yen Kan, editors,

Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pages 440–450. Association for Computational Linguistics, 2017.

[128] Yijun Yu. fAST: Flattening abstract syntax trees for efficiency. In *Proceedings of the 41th International Conference on Software Engineering, ICSE*, pages 278–279, 2019.

[129] C. Yuan, S. Wei, Y. Wang, Y. You, and S. ZiLiang. Android applications categorization using bayesian classification. In *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 173–176, Oct 2016.

[130] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV*, pages 818–833, 2014.

[131] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 111–121, New York, NY, USA, 2012. ACM.

[132] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *34th AAAI Conference on Artificial Intelligence*, 2020.

[133] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *ICSE*, 2019.

[134] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *International Conference on Software Engineering*, pages 783–794, 2019.

[135] Xinyi Zhang and Lihui Chen. Capsule graph neural network. In *International Conference on Learning Representations*, 2019.

[136] Wei Zhao, Jianbo Ye, Min Yang, Zeyang Lei, Suofei Zhang, and Zhou Zhao. Investigating capsule networks with dynamic routing for text classification. *arXiv preprint arXiv:1804.00538*, 2018.

[137] Hao Zhong, Suresh Thummalapenta, and Tao Xie. Exposing behavioral differences in cross-language API mapping relations. In *FASE*, pages 130–145, March 16-24 2013.

[138] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *ICSE*, pages 195–204, May 1-8 2010.

[139] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207, 2019.