



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Learning Natural Coding Conventions

Miltiadis Allamanis



Doctor of Philosophy

Institute for Adaptive and Neural Computation

School of Informatics

University of Edinburgh

2016

Abstract

Coding conventions are ubiquitous in software engineering practice. Maintaining a uniform coding style allows software development teams to communicate through code by making the code clear and, thus, readable and maintainable — two important properties of good code since developers spend the majority of their time maintaining software systems. This dissertation introduces a set of probabilistic machine learning models of source code that learn coding conventions directly from source code written in a mostly conventional style. This alleviates the coding convention enforcement problem, where conventions need to first be formulated clearly into unambiguous rules and then be coded in order to be enforced; a tedious and costly process.

First, we introduce the problem of inferring a variable’s name given its usage context and address this problem by creating Naturalize — a machine learning framework that learns to suggest conventional variable names. Two machine learning models, a simple n -gram language model and a specialized neural log-bilinear context model are trained to understand the role and function of each variable and suggest new stylistically consistent variable names. The neural log-bilinear model can even suggest previously unseen names by composing them from subtokens (*i.e.* sub-components of code identifiers). The suggestions of the models achieve 90% accuracy when suggesting variable names at the top 20% most confident locations, rendering the suggestion system usable in practice.

We then turn our attention to the significantly harder *method naming* problem. Learning to name methods, by looking only at the code tokens within their body, requires a good understating of the semantics of the code contained in a single method. To achieve this, we introduce a novel neural convolutional attention network that learns to generate the name of a method by sequentially predicting its subtokens. This is achieved by focusing on different parts of the code and potentially directly using body (sub)tokens even when they have never been seen before. This model achieves an F1 score of 51% on the top five suggestions when naming methods of real-world open-source projects.

Learning about naming code conventions uses the syntactic structure of the code to infer names that implicitly relate to code semantics. However, syntactic similarities and differences obscure code semantics. Therefore, to capture features of semantic operations with machine learning, we need methods that learn *semantic* continuous logical representations. To achieve this ambitious goal, we focus our investigation on

logic and algebraic symbolic expressions and design a *neural equivalence network* architecture that learns semantic vector representations of expressions in a syntax-driven way, while solely retaining semantics. We show that equivalence networks learn significantly better semantic vector representations compared to other, existing, neural network architectures.

Finally, we present an unsupervised machine learning model for mining syntactic and semantic code idioms. Code idioms are conventional “mental chunks” of code that serve a single semantic purpose and are commonly used by practitioners. To achieve this, we employ Bayesian nonparametric inference on tree substitution grammars. We present a wide range of evidence that the resulting syntactic idioms are meaningful, demonstrating that they do indeed recur across software projects and that they occur more frequently in illustrative code examples collected from a Q&A site. These syntactic idioms can be used as a form of automatic documentation of coding practices of a programming language or an API. We also mine semantic loop idioms, *i.e.* highly abstracted but semantic-preserving idioms of loop operations. We show that semantic idioms provide data-driven guidance during the creation of software engineering tools by mining common semantic patterns, such as candidate refactoring locations. This gives data-based evidence to tool, API and language designers about general, domain and project-specific coding patterns, who instead of relying solely on their intuition, can use semantic idioms to achieve greater coverage of their tool or new API or language feature. We demonstrate this by creating a tool that suggests loop refactorings into functional constructs in LINQ. Semantic loop idioms also provide data-driven evidence for introducing new APIs or programming language features.

Lay Summary

Software systems are made out of source code that defines in a formal and unambiguous way the instructions that a computer needs to execute. Source code is a core artifact of the software engineering process. However, since software systems need to be maintained and extended, source code needs to be frequently revisited by software engineers who need to read, understand and maintain the code. To this effect, source code acts as a *means of communication* between software developers and therefore source code needs to be easily understandable (and therefore easily modifiable).

To achieve this, software teams enforce — implicitly and explicitly — a set of coding conventions, *i.e.* a set of self-imposed restrictions on *how* source code is written. These conventions are not a product of any technical constraints or limitations but are imposed for efficient developer communication through source code. One important coding convention is related to naming software artifacts. The names need to clearly reveal the role and the function of each code artifact. Other conventions include the *idiomatic* use of source code constructs. These idioms convey easily understandable semantics and therefore aid humans when reasoning about code functionality.

This thesis presents an automated way for inferring and enforcing coding conventions to help software engineers write conventional and thus more maintainable code. To achieve this, we use machine learning — a set of statistical and mathematical modeling methods whose parameters are learned from data and can be used to make “smart” predictions about previously unseen observations. Specifically, this thesis presents machine learning models that learn to suggest conventional names for software engineering artifacts. This task requires novel machine learning models that “understand” the role and the function of the source code artifacts and how they compose to provide a distinct functionality.

In addition, this dissertation presents a machine learning-based method that automatically finds widely used *source code idioms* from a large set of source code. Code idioms are “mental chunks” of code that serve a single, easily identifiable semantic purpose. The mined idioms serve as a form of documentation of how code libraries and programming language constructs are used. Finally, we mine *semantic* idioms, mental chunks of code that are not syntactic but represent common types of operations. We show how these idioms can be used within software engineering tools and to support the evolution of programming languages.

Acknowledgements

When writing an acknowledgments section, one has to decide between being brief but vague or exhaustive and specific. I will pick the latter since I feel it is the only way to fully express my gratitude to all the people that have helped in many different ways during the last few years.

This PhD thesis would not have been possible without the constant, help from my PhD advisor, Charles Sutton. We have spent hundreds of hours in discussions and emails about research projects, while he patiently taught me how to tackle hard problems and acquire a “taste” for research problems. Without his visionary understanding of the field and his belief that great research impact is possible, this dissertation would not have been at its present state.

I would also like to thank Earl T. Barr, who although not officially related to my PhD acted as a remote PhD advisor, frequently chatting about new ideas, while he patiently explained to me programming language and software engineering concepts. Although being at UCL, his support was vital throughout this PhD.

This PhD has been kindly and generously supported by Microsoft Research through its scholarship program, thanks to the *Edinburgh Microsoft Research Joint Initiative in Informatics*. The scholarship has funded my PhD studies for the first three years. It also funded my travel expenses to conferences at amazing places all over the world. I am also grateful to Microsoft Research for the great experiences, during my two internships in Cambridge, UK and Redmond, WA, USA. I would like to specially thank Danny Tarlow, Andrew D. Gordon, Christian Bird and Mark Marron for their guidance throughout the internships and thereafter that significantly helped me. My interactions with them led to important adjustments to the course of this dissertation.

I would like to also thank Premkumar Devanbu and Pushmeet Kohli for their valuable help, advice and feedback. I am also grateful to Mirella Lapata, Shay Cohen, Jaroslav Fowkes, Krzysztof Geras, Akash Srivastava, Pankajan Chanthirasegaran and the members of CUP, IANC and ILCC for the numerous discussions and feedback that I have received the last few years.

This dissertation was only possible thanks to all the people and friends that have made me who I am; unfortunately I cannot list them all here. However, I want to specially thank Stella for making life fun and interesting for the last three years. Finally, and most importantly, I am grateful to my parents — Aleka and Nikos — who have patiently taught me so many things and have been a constant help, support and inspiration. This thesis is dedicated to them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Miltiadis Allamanis)

To my parents, Aleka and Nikos.

Table of Contents

1	Introduction	1
1.1	Main Contributions	4
1.2	Thesis Structure	5
1.3	Declaration of Previous Work	5
2	Background: Probabilistic Models of Source Code	7
2.1	Structure and Scope	9
2.2	Source Code Representations and Derivative Artifacts	9
2.3	Probabilistic Models of Source Code	12
2.3.1	Code Generating Probabilistic Models of Source Code	13
2.3.2	Representational Models of Source Code	18
2.3.3	Pattern Mining Models of Source Code	24
2.4	Applications	29
2.4.1	Recommender Systems	29
2.4.2	Inferring Coding Conventions	33
2.4.3	Code Defects and Debugging	34
2.4.4	Code Migration	35
2.4.5	Source Code and Natural Language	35
2.4.6	Program Synthesis	36
2.4.7	Documentation and Summarization	37
2.4.8	Program Analysis	37
2.5	Conclusions	38
3	Background: Coding Conventions	39
3.1	Naming Coding Conventions	40
3.2	Formatting Coding Conventions	42
3.3	Coding Patterns	43

3.4	Enforcing Coding Conventions in Practice	44
3.5	Conclusions	45
4	Learning Variable Naming Conventions	47
4.1	Motivating Example	49
4.1.1	Use Cases and Tools	50
4.2	The NATURALIZE Framework	54
4.2.1	The Core of NATURALIZE	55
4.3	Choices of Scoring Function	59
4.3.1	Using the n -gram Language Model	59
4.3.2	Log-bilinear Context Models of Code	61
4.3.3	Subtoken Context Models of Code	67
4.3.4	Source Code Features for Context Models	69
4.4	Evaluation	70
4.4.1	Quantitative Evaluation	73
4.4.2	Suggestions Accepted by Projects	79
4.5	Learned Representations	79
4.6	Conclusions	84
5	Learning Method Naming Conventions	85
5.1	A Convolutional Attention Model	89
5.1.1	Learning Attention Features	91
5.1.2	Simple Convolutional Attention Model	93
5.1.3	Copy Convolutional Attention Model	93
5.1.4	Predicting Names	94
5.2	Evaluation	95
5.2.1	Quantitative Evaluation	98
5.2.2	Qualitative Evaluation	103
5.2.3	Comparison with Log-Bilinear Model	104
5.3	Learned Representations	106
5.4	Conclusions	108
6	Learning Continuous Semantic Representations of Symbolic Expressions	111
6.1	Neural Equivalence Networks	114
6.1.1	Neural Equivalence Networks	117
6.1.2	Training	119

6.2	Evaluation	120
6.2.1	Quantitative Evaluation	121
6.2.2	Qualitative Evaluation	130
6.3	Related Work in Machine Learning	131
6.4	Conclusions	132
7	Mining Idiomatic Source Code	133
7.1	Problem Definition	137
7.2	Mining Idioms	141
7.2.1	Probabilistic Grammars	142
7.2.2	Learning TSGs	144
7.3	Mining Syntactic Idioms	152
7.4	Syntactic Idioms Evaluation	154
7.4.1	Top Idioms	158
7.4.2	Code Cloning vs. Code Idioms	161
7.4.3	Extrinsic Evaluation of Mined Syntactic Idioms	162
7.4.4	Syntactic Idioms and Code Libraries	163
7.5	Mining Semantic Idioms	165
7.5.1	Purity Analysis	168
7.5.2	Coiling Loops	170
7.5.3	Mining Semantic Idioms	172
7.5.4	Semantic Idiom Ranking	173
7.6	Semantic Idioms Evaluation	173
7.6.1	Using Semantic Loop Idioms	177
7.7	Conclusions	183
8	Conclusions	185
8.1	Future Work	187
A	List of Published Work	189
B	GitHub Pull Request Discussions	191
B.1	JUnit Pull Request #834	192
B.2	Elasticsearch Pull Request #5075	194
B.3	K9 Pull Request #454	195
B.4	libgdx Pull Request #1400	196

Acronyms

API application programming interface.

AST abstract syntax tree.

CFG context free grammar.

CoFG control flow graph.

CRF conditional random field.

DSL domain specific language.

FPR false positive rate.

GPU graphical processing unit.

IDE integrated development environment.

IR information retrieval.

LBL log-bilinear.

LM language model.

ML machine learning.

NLP natural language processing.

PCFG probabilistic context free grammar.

PDG program dependence graph.

SMT statistical machine translation.

TPR true positive rate.

TSG tree substitution grammar.

Chapter 1

Introduction

*“No matter what future you might imagine, it will
rely on software-intensive systems not yet built.”*

– Grady Booch (ICSE 2015 Keynote)

Software plays an increasingly important role in our lives, permeating our everyday activities. We use software for our daily communications, work and entertainment. Thanks to software, we are able to communicate with people that are thousands of miles away, watch movies and read articles. Software makes transportation safer and more efficient and powers essential devices in our homes. “*Software is eating the world*” (Andereessen, 2011). However, developing software is a costly process: software engineers need to tackle the inherent complexity of software to avoid bugs, reduce development and maintenance costs, and deliver software products on time. Our reliance on software makes it imperative to research new tools that help us make software more reliable and maintainable. New methods are needed to reduce the complexity of software and help engineers construct better software. Additionally, novel methods that transfer knowledge across projects and people are required to make software development a less time consuming process. Finally, new techniques for detecting software defects are necessary for creating even more robust software systems.

The interdisciplinary area of *probabilistic machine learning models of source code* is one new and promising approach towards solving these problems. It is located at the intersection of machine learning, software engineering and programming languages research. The goal is to create probabilistic source code models that *learn* how developers use code artifacts within existing code while taking into account the highly structured information within code. These models are then used to augment existing tools with statistical information and enable new machine learning-based software engineering

tools, such as recommender systems and statistical code analyses.

This new direction is now possible thanks to the increased availability of open-source projects and centralized hosting services such as GitHub and BitBucket. This raises the exciting possibility that probabilistic models of source code learn from the code that has already been written. Then these models can be used to create machine learning-based tools that help software engineers develop, maintain, reuse and test their code.

Within this field, this thesis presents the first — to our knowledge — research on probabilistic models of source code that learn how to name source code identifiers such as variable and method names, and detect common and conventional code patterns. This dissertation examines the topic of inferring coding conventions and creating recommender systems that suggest or present the inferred conventions to developers. Coding conventions, a topic of common concern within software engineering teams, refer to implicit or explicit conventions that software developers use when developing software. These conventions are *not* requirements of the programming language but self-imposed constraints that aim to maintain a uniform coding style within a codebase and render source code easier to read, understand and maintain. Software engineers care deeply about conventions, such as naming variables and methods. Good names make the code more understandable (McConnell, 2004; Brooks, 1975) and reveal the role and function of each code unit (*e.g.* method or variable) while appealing to the well-developed verbal abilities of humans (Siegmund et al., 2014). This also applies to non-naming conventions, such as idiomatic usage of programming constructs to more clearly reveal and communicate the intention of the code. For example, in C-style languages although a developer can use either a `for` or a `while` loop to express the same semantic operation, `for` loop constructs are preferred by convention when iterating over an array.

This thesis introduces the *coding convention inference problem*, *i.e.* the problem of automatically inferring coding conventions within a codebase. We are aware of no previous work on *learning* conventions from existing codebases. Once the knowledge about conventions is inferred, it can be presented or embedded within software engineering tools that help engineers write, understand and maintain code. Machine learning systems that “understand” and reason about code have the potential to change the software engineering process by providing smart software assistant-like tools to software engineers that can help them throughout their daily work.

In Chapter 4, we present two approaches to the variable naming inference prob-

lem and in Chapter 5 we present two models for the method naming inference. We are not aware of any previous work tackling the variable name inference problem¹ or the method naming problem. These two chapters present the first — to our knowledge — approach to solving the variable and method naming problems. In fact, the performance on the variable naming problem is adequate to be embedded in a usable software engineering tool. The model presented in those chapters make an extensive use of the syntactic structure of the code and common patterns that appear within them. In Chapter 6, we look into machine learning methods that abstract the syntax of the symbolic language used to express procedural knowledge but retain its semantics. This is a significantly harder problem since we now need to directly represent code semantics rather than its syntactic structure, while allowing syntax to drive semantics. Learning about semantic code operations features can help machine learning systems reason about semantics when inferring coding conventions. Finally, in Chapter 7 we define the notion of *coding idiom* as that of a widely reusable pattern of code and present a method for inferring *syntactic and semantic code idioms*. We are not aware of any other methods for mining coding patterns that include syntactic or semantic structure. These idioms represent conventional code patterns used within projects that can be used for documentation, software tool design and language design.

Although the coding conventions inference problem is of direct interest to software engineering research, it should not be seen solely as a research problem in this domain. This problem is deeply connected to core machine learning and programming language research questions. Learning to name source code identifiers implies a deep understanding about the *role* and *function* of the named elements. A machine learning model that is able to correctly name a variable implicitly has learned something about the semantic role and function of that variable. For example, if a model successfully predicts the name of a variable `elementCounter`, it has learned features that characterize the semantic role and function of variables acting as counters, *e.g.* the fact that they are non-negative integers counting some quantity. Such information can be useful for automatic verification of computer programs. Similarly, if a probabilistic model correctly predicts the name of a method name `sortList`, this suggests that this model has linked the semantics of the code within the method body to the concept of sorting. Such an understanding could be useful for program synthesis or probabilistic reuse of programs within machine learning algorithms. Therefore, convention inference should *not* just

¹The variable naming inference problem was independently and simultaneously researched by Raychev et al. (2015).

be seen just as a problem useful solely to software engineers, but also as a first step towards creating probabilistic models that are able to probabilistically “understand” code semantics.

In a “*software-powered world*”² it is imperative that we research, develop and deploy new tools and methods that enable engineers to efficiently create robust and bug-free software. In this thesis, we argue that probabilistic models of source code can be part of the solution towards this goal and present novel machine learning methods and tools that move towards this direction.

1.1 Main Contributions

The main contributions of this thesis are:

- A novel comprehensive approach to the problems of inferring variable and method names at a token and subtoken level from existing code. We present the first general framework for suggesting and evaluating the quality of the suggested names.
- Three machine-learning models that learn to name variables and methods at a token or subtoken level, outperforming existing methods on the task. Our best variable naming model achieves an F1 score of 94% on the top suggestions, while our method naming model achieves an F1 of 45%.
- A novel neural equivalence network architecture that learns to abstract syntactic similarities of expressions but retain its semantics, learning semantic vector representations that map expression semantics to a real-valued D-dimensional space. This method is important for learning semantic features about source code or symbolic expressions. We show that this network significantly outperforms other existing architectures.
- A novel method for mining conventional syntactic and semantic idioms from existing codebases based on unsupervised Bayesian nonparametrics. We show that this model learns idioms that are useful as documentation to software engineers and that idioms can provide valuable information when designing software tools and programming languages.

²Quote from Satya Nadella’s email to employees on first day as CEO at Microsoft <https://news.microsoft.com/2014/02/04/satya-nadella-email-to-employees-on-first-day-as-ceo/>.

1.2 Thesis Structure

The structure of the thesis follows. First, Chapter 2 presents an extensive literature review of the area of probabilistic modeling of source code artifacts. Chapter 3 presents a literature review of coding conventions within software engineering research. The two aforementioned chapters present the necessary common background for the next chapters that present the research undertaken during this PhD.

Chapter 4 discusses and evaluates two probabilistic models of source code for learning variable naming conventions. Next, Chapter 5 presents and evaluates two models for learning and inferring method naming conventions. Chapter 6 presents a method for learning continuous *semantic* vector representations of symbolic expression abstracting over their syntactic form. Finally, Chapter 7 discusses a method for mining syntactic and semantic conventional idioms of source code and their applications.

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 for all these publications. Specifically, Chapter 4 contains work published in “Learning Natural Coding Conventions” ([Allamanis et al., 2014](#)) and “Suggesting Accurate Method and Class Names” ([Allamanis et al., 2015a](#)). Chapter 5 contains work published in “Suggesting Accurate Method and Class Names” ([Allamanis et al., 2015a](#)) and “A Convolutional Attention Network for Extreme Summarization of Source Code” ([Allamanis et al., 2016d](#)). Chapter 6 contains the work published in [Allamanis et al. \(2016c\)](#). Finally, Chapter 7 contains work found in “Mining Idioms from Source Code” ([Allamanis & Sutton, 2014](#)) and “Mining Semantic Loop Idioms from Big Code” ([Allamanis et al., 2016a](#)). Finally, Chapter 2 contains material that the author presented to the NL+SE workshop in Seattle, WA in November 2016.

Chapter 2

Background: Probabilistic Models of Source Code

“All models are wrong, but some are useful.”

– George Box

This section reviews the area of probabilistic, machine learning-based source code models aiming to give a broad overview of the area, explain the core methods and techniques and present the available tooling and applications. Source code is a highly structured object, but traditionally it is analyzed deductively. This review chooses to ignore deterministic, deductive and other logic-based methods for source code analysis and discusses work that uses a *statistical learning component* to acquire knowledge and generalize accurately about some aspect of the code to provide probabilistic estimations for some prediction or other target quantity.

This type of source code models has recently created a new interdisciplinary subfield called *naturalness of code* or *big code*. This subfield is found in the intersection of software engineering — and its mining software repositories community — programming language, machine learning and natural language processing research. The goal of this research area is to develop tractable machine learning models of source code elements with the intention to provide to developers and their managers useful, “smart” software engineering tools that learn from existing codebases. The recent advances in this field are attributed to the rapid improvements in machine learning (ML) and natural language processing (NLP) and the proliferation of open-source software and collaborative code sites such as GitHub and BitBucket that make large amounts of source code repositories available at a central location.

The core intuition of probabilistic models of source code is that developers im-

plicitly embed knowledge in source code when they are constructing new code or maintaining existing software systems. By using statistical, probabilistic models of various source code elements, valuable information can be mined, handling ambiguities and partial information in a principled way. Ambiguities and partial information arise in multiple contexts, such as when inferring the latent user intent (*e.g.* during code completion), when synthesizing code from incomplete context (*e.g.* from examples) or when performing an inherently ambiguous task such as code summarization. The work in this field aims to learn from code to create new tools that may allow new code analyses or speed up or improve existing ones. This field can, therefore, be thought as a form of *statistical source code analysis*.

The names of this nascent area, “big code” and “naturalness”, find their origins in some of the first papers within this subfield. The term “big code” — which was created as an analogous to the (marketing) term *big data* by DARPA’s MUSE project¹ and Raychev et al. (2015) — suggests that with large amounts of source code data we can learn valuable information about code and use this information within tools. However, this term may miss the fact that probabilistic models of source code can still be useful without requiring large amounts of data to learn. The notion of “naturalness” of source code suggests that source code is repeatable and thus has some predictability — observed by many researchers (Hindle et al., 2012; Gabel & Su, 2010; Barr et al., 2014; Velez et al., 2015; Nguyen et al., 2016a). Although this might not seem surprising, one should appreciate the root cause of “naturalness”. Naturalness of code seems to have a strong connection with the fact that developers write conventional and idiomatic code (Allamanis et al., 2014) because it helps with understanding and maintaining software systems. However, the term “naturalness” may suggest that this area is just as a way of learning surface coding conventions, whereas “code predictability” notion is significantly wider suggesting that most code artifacts — from simple token sequences to formal verification statements — contain useful and somewhat predictable patterns that can be exploited. Both the “naturalness” and “big code” concepts should be viewed in a more general setting. They suggest that there is exploitable regularity across code that can be “absorbed” and generalized by a learning component that can in turn probabilistically reason about new code.

¹<http://science.dodlive.mil/2014/03/21/dar-pas-muse-mining-big-code/>

2.1 Structure and Scope

This review is structured as follows. Since source code is the data of our models of interest, we first discuss the format of source code data, its representations and its derivative formats (Section 2.2). Then, a taxonomy of probabilistic models of source code is discussed (Section 2.3). Finally, we describe the software engineering and programming language applications of probabilistic source code models (Section 2.4).

In this review, we choose to include work that models source code in a probabilistic way, contains a *learning component* and uses complex representations that transcend the simple bag-of-words (of source code tokens) representation of the underlying code. Non-probabilistic models of source code (*e.g.* formal specifications) are widely used in software engineering and programming language research but are out of scope. Probabilistic models of source code that do *not* include a learning component (*e.g.* Lau (2001)) are also out of scope. We also exclude machine learning models that use other software engineering data that can be derived from the software engineering process (*e.g.* process metrics, requirement traceability) but do not directly model source code.

Other Reviews To our knowledge there have been some reviews that summarize the progress and the vision of the research area. Devanbu (2015) discusses the area of code naturalness and the applications that it has in software engineering, targeting the software engineering community. Devanbu briefly discusses some areas where research activity is happening in the subfield: (a) Statistical Models of Code (b) Porting and Translation (c) Linguistics of Code (d) Suggestion and Completion (e) Analysis and Tools (f) Assistive Technologies. Bielik et al. (2015) discuss the ideas of probabilistic models of source code and their applications to a programming language audience and describe a framework stemming from their previous work (Raychev et al., 2015). Finally, Neubig (2016) provides an informal survey of methods for generating natural language from source code. Some resources, datasets and code can also be found at <http://learnbigcode.github.io/>.

2.2 Source Code Representations and Derivative Artifacts

Source code is the means of explaining to a computer *what* instructions to execute in order to achieve a specific task. However, for the purposes of probabilistic models of

source code and this review, source code acts as the input or output *data* of the described models. It is therefore important to discuss the *form* of the data and the various code representations that are used.

Source code is usually a set of text files. At its simplest form each source code file is a sequence of characters that form tokens (also known as lexemes). There are two types of tokens. The instruction tokens that have a syntactic or semantic meaning and are essential for the program to run. Non-instruction tokens (*e.g.* comments and whitespace characters) are meant to help the developers understand the code and are ignored by the computer. Note that in some cases whitespace characters have syntactic meaning tokens (*e.g.* indentations in Python) and are essential for the code semantics. For languages that have preprocessors (*e.g.* C, C++, C#) the tokenized code usually refers to the code *after* the preprocessor is executed. To extract the source code tokens a lexer (also referred to as a tokenizer or scanner) is used.

Given the tokens of the code, a parse tree (also known as “syntax tree”) and an abstract syntax tree (AST) can be computed in an unambiguous way. The AST contains all the necessary information about the code, abstracting some tokens (*e.g.* commas, parentheses, braces *etc.*). In some tools, although the AST contains all the necessary information that are contained in the code, the transformation between token sequences and ASTs is not one-to-one. For example, parenthesizing a single variable or removing the parentheses may result into identical ASTs. Some libraries support round-trip conversion of ASTs to source code (possibly also including whitespace) by internally storing additional information.

Although ASTs are tree structures, their structure is more complex than trees studied in introductory computer science courses or in natural language processing. Each node in an AST has one or more *properties* that are fixed for a given type of node by the (formal) grammar of the programming language specification. For example, in C# `ForStatementSyntax` nodes always have the properties `Declaration`, `Initializers`, `Condition`, `Incrementors`, `Statement`. Each property — depending on its predefined semantics — may have exactly one, zero or one, or zero or more nodes as children. In the probabilistic source code modeling literature, this subtlety is handled in different possible ways: (a) considering the properties as (dummy) children nodes (Allamanis et al., 2016a) (b) ignoring the properties and using a “flat” structure for the children of the node (Maddison & Tarlow, 2014; Bielik et al., 2016) (c) handle the property as substructure within each node (*e.g.* as labels) (Allamanis & Sutton, 2014).

Both token-level and AST-level models of source code can be further augmented

by resolving information about some of the tokens. Compilers and some analysis tools bind each token/node to symbols, given the scope and other available compile-time information. For example, all the tokens that bind to the same local variable in a scope are linked to the same symbol.

From an AST we can create graphs that provide partial or complete views of the code. Control flow graphs (CoFGs)² are directed graphs that describe how control flows within a given procedure. In the case that a CoFG is computed across functions/methods it is called an *interprocedural* CoFG, although in most real-life cases retrieving a static interprocedural CoFG is impossible (*e.g.* because of polymorphism).

Another view of the code is the data flow graph, that presents how data flows within a piece of source code. The program dependence graph (PDG) combines control and data flow within a single graph. The literature contains other (partial) views of the code, such as Groums (Nguyen et al., 2009) that contain only actions (*e.g.* application programming interface (API) calls) and control flow.

Finally, we have derivative source code artifacts, that are retrieved from run-time traces acquired during code execution. In comparison with the aforementioned representations, traces provide partial information about the behavior of programs. Commonly used traces include memory traces (*e.g.* the heap, represented as a directed graph) and code execution traces (represented as trees or sequences).

Natural Language Text vs. Programming Languages Text Although source code tokens and ASTs have significant similarities to natural language sentences and parse trees there are a few important differences. Programming languages are by definition non-ambiguous and can be deterministically parsed. Another important aspect of this domain is that software engineers actively avoid repeating the same functionality twice, pursuing *code reusability* (Bourque et al., 2014). Most often, code that is reused in multiple locations is encapsulated within libraries and is referenced when needed. This creates a scarcity of data and multiple implementations of identical functionality are rare in real-life source code, with the exception of coding competitions, student assignments, and implementation of similar features across different programming languages.

Natural language sentences tend to be relatively small — usually less than 15 tokens long — and their syntax tends to be relatively shallow. In contrast, even when limiting code to single functions (methods) we can easily get source code “sentences” that are

²Literature in programming language research uses the abbreviation CFG for control flow graphs, but we avoid this here due to the possible confusion with the Context Free Grammars that are widely used in natural language processing.

Table 2.1: Commonly used Tools for Extracting Code Data.

Tool	Type	Language	Notes
ANTLR	Tokenizer, Parser	Any	A general framework for lexing and parsing
Eclipse JDT	Tokenizer, Parser, Compiler	Java	Provides support for tokenization and AST extraction. Symbols may not be full resolved if not tied with an Eclipse Project
esprima.js	Tokenizer, Parser	JavaScript	JavaScript library for tokenization and parsing. Multiple plugins.
ast	Parser	Python	Support for AST extraction. Limited round-trip information
Roslyn	Tokenizer, Parser, Compiler	C#, Visual Basic	.NET library for source code analysis.
astroid	Parser	Python	AST extraction and symbol resolution. Limited documentation.

100 tokens long and the depth of the AST often exceeds 50. A further complication arises from the identifiers in source code (Allamanis & Sutton, 2013b). Developers frequently create new names for variables by agglutinating different words/subtokens (*e.g.* NUM_NODES and numRedNodes). This creates significant sparsity within the vocabulary. Splitting the identifiers into subtokens is frequently necessary.

Tools Extracting the source code representations mentioned above can be done using a preexisting tool. In Table 2.1 we mention some of the tools that researchers have commonly used. This is *not* intended to be a complete list, rather than a set of pointers that may allow someone interested in this domain to get started.

2.3 Probabilistic Models of Source Code

We now turn our attention to probabilistic machine-learning models of source code. Probabilistic machine-learning models, are statistical models that make some (simplifying) assumptions about the domain being modeled. These assumptions are necessary to make the models tractable to learn and use, but also induce errors. Since each model makes a different set of assumptions, each model has its own strengths and weaknesses and therefore it may be more suitable for different applications. In the next section (Sec-

tion 2.4) we will discuss the applications that these models find in software engineering and programming languages.

For the purpose of this section, we categorize probabilistic source code models into three non-mutually exclusive categories based on the form of the modeled probability distribution and the type of the input/output they accept.

Code Generating Models are able to create full snippets of source code by sequentially generating its parts (*e.g.* tokens or AST nodes) in a probabilistic way;

Code Representational Models model some aspect of the code by learning an intermediate representation. This representation is subsequently used to predict elements of the code, properties of some code unit (*e.g.* types of variables) or another extrinsic property of the code, usually returning a probability distribution over those properties;

Pattern Mining Models infer in an unsupervised manner the latent structure within source code and thus are used for extracting a discrete number of reusable and human-interpretable patterns. These models may be seen as the instantiation of clustering-like models within this application domain.

The taxonomy presented here allows us to discuss various models at a high level. However, it should be noted that other categorizations are also possible. An orthogonal categorization of models of source code, can be derived based on the granularity at which they view source code.

Lexical models view source code as a sequence of tokens (or characters).

Syntactic models view source code as an (abstract) syntax tree.

Semantic models view source code as a graph (*e.g.* as a program dependence graph) of semantic relationships among its elements.

2.3.1 Code Generating Probabilistic Models of Source Code

Code generating probabilistic models of source code describe probability distributions over *all* possible productions of code. Given some training data \mathcal{D} and for some source code³ c and some, possibly empty, context $C(c)$ those models learn the probability

³To simplify the notation, we use c to imply an arbitrary representation of source code, such as token sequence, AST or graph.

distribution $P_{\mathcal{D}}(\mathbb{c}|C(\mathbb{c}))$ and are able to probabilistically generate source code given solely the context $C(\mathbb{c})$ by sampling $P_{\mathcal{D}}$. We categorize these models into three categories: language models, multimodal models and translation models. When $C(\mathbb{c}) = \emptyset$, the probability distribution $P_{\mathcal{D}}$ is a *language model of source code*. When $C(\mathbb{c})$ is a non-code modality (e.g. natural language), $P_{\mathcal{D}}$ describes a *code-generative multimodal model of source code*. When $C(\mathbb{c})$ is also source code, the probability distribution $P_{\mathcal{D}}$ is a *translation model of source code*. Apart from generating code, by definition, code generating probabilistic models of source code *score* source code assigning a non-zero probability to every possible snippet of code. This score, sometimes referred to as “naturalness” of the code, suggests how probable the code is under the learned model and depends on the training data \mathcal{D} .

Since all code generating models predict the complex structure of source code, they always make some simplifying assumptions about the generative process and iteratively predict elements of the source code to generate a full code unit (e.g. code file or method). Because of the highly structured nature of source code (and the simplifying assumptions made by each model), none of the existing models in the literature can generate code that *always* parses, compiles or can be executed. However, some of the models take into consideration the source code structure and impose additional constraints to remove some inconsistencies (e.g. [Maddison & Tarlow \(2014\)](#) generate variables that have already been declared within the scope).

2.3.1.1 Language Models

Language models (LMs) for source code are the most widely used form of code-generating models and have proven useful for many software engineering and programming language applications. Since code has been found to be “natural” ([Hindle et al., 2012](#)), language models have been widely used and researched. These models are heavily inspired by natural language processing (NLP) language models. Although LMs learn the high-level structure of programming languages fairly easily, predicting and generating source code identifiers (e.g. variable and method names) makes language modeling hard and interesting ([Allamanis & Sutton, 2013b](#); [Maddison & Tarlow, 2014](#)).

Token Language Models Token language models are common in this field. These models view source code as a sequence of tokens, *i.e.* $\mathbb{c} = t_1 \dots t_M$. Since predicting sequences is a complex problem most of the models make simplifying assumptions about

how code is generated, usually generating one token at a time, modeling $P(t_m|t_1 \dots t_M)$

The n -gram LM is the most widely used LM in the area of source code-generative probabilistic models and is one of the most effective practical LMs. N -gram models make the assumption that the next token can be predicted using only the previous $n - 1$ tokens. Formally, the probability of a token t_m , conditioned on all of the previous tokens $t_1 \dots t_{m-1}$, is a function only of the previous $n - 1$ tokens. Under this assumption, we can write

$$P_{\mathcal{D}}(\mathbb{C}) = P(t_1 \dots t_M) = \prod_{m=1}^M P(t_m|t_{m-1} \dots t_{m-n+1}). \quad (2.1)$$

To use this equation we need to know the conditional probabilities $P(t_m|t_{m-1} \dots t_{m-n+1})$ for each possible n -gram. This is a table of V^n numbers, where V is the number of possible lexemes. These are the *parameters* of the model that we learn from the training corpus. The simplest way to estimate the model parameters is to set $P(t_m|t_{m-1} \dots t_{m-n+1})$ to the proportion of times that t_m follows $t_{m-1} \dots t_{m-n+1}$. In practice, this simple estimator does not work well, because it assigns zero probability to n -grams that do not occur in the training corpus. Instead, n -gram models are trained using *smoothing* methods (Chen & Goodman, 1996). These methods find a principled way for assigning probability to unseen n -grams by extrapolating information from m -grams ($m < n$).

The use of n -gram LMs in software engineering has originated with the pioneering work of Hindle et al. (2012) who used an n -gram language model with Kneser-Ney (Chen & Goodman, 1996) smoothing. Most related research has followed this practice. Nguyen et al. (2013b) extended the n -gram LM by annotating the code tokens with parse information. Raychev et al. (2014) use concrete and abstract semantics of code, to create a token-level language model that treats language modeling as a combined synthesis and probabilistic modeling task. At the same time, Tu et al. (2014) noticed that source code has a high degree of *localness*, where tokens (*e.g.* variable names) are repeated often within close distance and extended previous work in speech and natural language processing (Kuhn & De Mori, 1990) adding a cache mechanism that assigns higher probability to tokens that have been observed most recently, achieving better performance compared to other n -gram LMs. Since n -gram LMs do *not* generate syntactically valid code, Raychev et al. (2014) add constraints — derived from program analysis — to the generative process, avoiding some incorrect code.

In the past few years, NLP research has created recurrent neural network LMs that improve upon the performance of the n -gram LM. Again, these models predict each token sequentially, but loosen the fixed-context-size assumption. Instead they

represent the current context using a distributed vector representation (discussed in Subsection 2.3.2). Following this trend, [Karpathy et al. \(2015\)](#) study a character-level LSTM of the Linux kernel. Similarly, [White et al. \(2015\)](#) and [Dam et al. \(2016\)](#) create token-level recurrent neural networks. Recently, [Bhoopchand et al. \(2016\)](#) used a token-level sparse pointer-based neural language model of Python that learns to copy recently declared identifiers, in order to capture very long-range dependencies of identifiers, showing that they can outperform standard LSTM language models. Although neural LMs usually have superior predictive performance, training neural LMs is significantly more costly compared to n -gram LMs.

Syntactic Models Syntactic (or structural) language models of code directly model the code ASTs. Thus, in contrast to n -gram LMs, they describe the process of generating trees. Such models make simplifying assumptions about how an AST is generated, usually following NLP models of natural language grammar trees. Most source code syntactic models sequentially generate nodes, starting from a predefined root node and generating the next node from top to bottom and left to right. When generating a tree node, syntactic models consider a subset of the tree that has already been generated as context of the generation process. [Maddison & Tarlow \(2014\)](#) create a log-bilinear neural network to predict the AST structure, using a distributed vector representation for the context. Additionally, they limit their model to respect variable scopes. This work is the first to suggest that probabilistic context free grammars (PCFGs) are not suitable for modeling source code. This is because PCFGs consider very limited context and thus are bad at modeling the highly verbose structure of ASTs. Recently, [Raychev et al. \(2016\)](#); [Bielik et al. \(2016\)](#) generalize PCFGs by annotating them with arbitrary synthesized programs that use features from the code also improve the performance over a PCFG. Similarly, [Wang et al. \(2016c\)](#) use an LSTM over the AST nodes for the same problem. [Allamanis & Sutton \(2014\)](#) also create a syntactic language model learning Bayesian tree substitution grammars (TSGs). However, although this model is a syntactic language model, they use it for unsupervised learning of code idioms (*i.e.* as a pattern mining model discussed in Subsection 2.3.3).

Semantic Language Models Semantic language models view source code as a graph with nodes representing code elements and the edges representing the relations among them. Generating highly structured graphs, including graph representations of source code, is hard and all generation processes make a multitude of simplifying assumptions. Within source code, [Nguyen & Nguyen \(2015\)](#) create a graph-based LM to suggest API completion. In contrast with the language models discussed so far, this model abstracts

some information about the code and the language model can only generate partial code.

Evaluation of Language Models Evaluation of source code language models is performed similarly to natural language processing, using perplexity, cross-entropy and word error rate. Cross-entropy H is the most common measure and is defined as the average number of bits per token required to “transmit” the code, *i.e.*

$$H(\mathbf{c}, P_{\mathcal{D}}) = -\frac{1}{M} \log_2 P_{\mathcal{D}}(\mathbf{c}) \quad (2.2)$$

where M is the number of tokens within \mathbf{c} . Note that the average is per-token by conventions, even when a non-token model is used. This makes comparisons across different models valid. In some cases, where the LM is used within an application, application-specific metrics are employed.

2.3.1.2 Translation Models

Inspired from statistical machine translation (SMT), translation models translate code written in one (source) language to another (target) language. These models use SMT techniques and frameworks — traditionally used in NLP. This work has found application within code migration (Aggarwal et al., 2015; Nguyen et al., 2015; Karaivanov et al., 2014) and pseudocode generation (Oda et al., 2015). Translation models, combine a language model $P_{\mathcal{D}}(\mathbb{t})$ to model the target language with a translation model $P_{\mathcal{D}}(\mathbb{s}|\mathbb{t})$ to match words between languages. These methods pick the optimal translation t^* such that

$$\mathbb{t}^* = \arg \max P_{\mathcal{D}}(\mathbb{t}|\mathbb{s}) = \arg \max P_{\mathcal{D}}(\mathbb{s}|\mathbb{t})P_{\mathcal{D}}(\mathbb{t}).$$

Again, these probabilistic generative models of source code do *not* necessarily produce valid source code, due to the simplifying assumptions they make in the language model $P_{\mathcal{D}}(\mathbb{t})$ and the translation model $P_{\mathcal{D}}(\mathbb{s}|\mathbb{t})$. Evaluation of translation models can be done with NLP evaluation metrics, such as BLEU (Papineni et al., 2002), or programming-related measures (*e.g.* does the translated code parse/compile?).

2.3.1.3 Multimodal Models

Code-generating multimodal models of source code correlate one or more non-code modalities to source code. These model have the form $P_{\mathcal{D}}(\mathbf{c}|\mathbf{m})$ where \mathbf{m} is a representation of one or more non-code modalities \mathbf{m} . Multimodal models have a close relation to representational models (discussed next in Subsection 2.3.2). Multimodal

code-generating models learn an intermediate representation of the non-code modalities and generate source code *given* that representation. In contrast, code representational models create an intermediate representation *of the code* and use it to predict a non-code modality. This stream of research has received limited attention because of the scarcity of available data. Multimodal models of source code, have been used for code synthesis, where given the non-code modalities, a conditional generative model of source code can be estimated (*e.g.* synthesis of source code given a natural language description by [Gulwani & Marron \(2014\)](#)). Another use of those models is to score the co-appearance of the modalities (*e.g.* in code search, to score the probability of some text given a textual query ([Allamanis et al., 2015b](#))). Multimodal models make two kinds of modeling assumptions. One set of assumptions concern the transformation of the input modality into some intermediate representation and the other assumptions for the code generating process exactly as in language models.

This area is closely related to semantic parsing in NLP, where the input modality \mathbf{m} is natural language text and the other modality \mathbf{c} is some form of executable instructions. However, research in this area is out-of-scope from this review, since it involves carefully crafted domain specific languages (DSLs) rather than full-scale programming languages. [Neubig \(2016\)](#) contains a list of relevant publications.

2.3.2 Representational Models of Source Code

Representational source code models learn an intermediate (not necessarily human-interpretable) representation of source code and use it to predict the probability distribution of code unit properties (*e.g.* variable types) or of some non-code artifact. These probabilistic source code models, learn the conditional probability distribution of some code \mathbf{c} as $P_{\mathcal{D}}(\pi|f(\mathbf{c}))$ where f is a function that transforms the code \mathbf{c} to an appropriate representation, possibly abstracting or removing some details. The target π is the property that is modeled. Representational source code models have diverse architectures and are often application-specific. Although the majority of the representational models are *discriminative* machine learning models, directly modeling $P_{\mathcal{D}}(\pi|f(\mathbf{c}))$, not all models follow this rule. Table 2.3 lists research related to representational source code models.

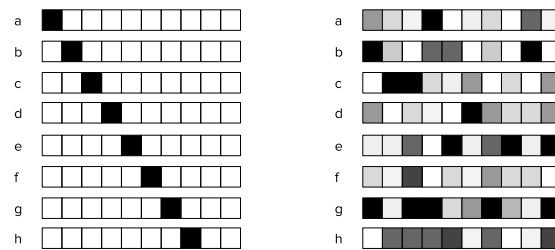


Figure 2.1: Local (left) vs. distributed (right) vector representations. Local representations use exactly one component of the vector representation for each element, while distributed representations distribute the meaning into multiple components.

Table 2.2: Research on Source Code Generating Models (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\mathbb{C}|\mathcal{C}(\mathbb{C}))$ and describe the process for generating source code. References annotated with * are also included in other categories.

Reference	Model Type	Kind	Model ($P_{\mathcal{D}}$)	Application
Aggarwal et al. (2015)	Translation	Token-Level	Phrase-based SMT	Migration
Allamanis & Sutton (2013b)	Language	Token-Level	n -gram	–
Allamanis et al. (2014) ⁴	Language	Token-Level & Variable Bindings	n -gram	Learning Conventions
Allamanis & Sutton (2014) ^{*5}	Language	Syntax Level	TSG	–
Allamanis et al. (2015b) [*]	Multimodal	Syntax-Level	Neural Network	Code Search/Synthesis
Bhatia & Singh (2016)	Language	Token-Level & Parse	Neural Network	Syntax Error Correction
Bhoopchand et al. (2016)	Language	Token-Level	Neural Network	Code Completion
Bielik et al. (2016)	Language	Syntax-Level	PCFG + learned annotations	Code Completion
Campbell et al. (2014)	Language	Token-Level	n -gram	Syntax Error Detection
Cerulo et al. (2015)	Language	Token-Level	HMM	Information Extraction
Dam et al. (2016)	Language	Token-Level	Neural Network	–
Gulwani & Marron (2014)	Multimodal	Syntax-based	Phrase-based Model	Text-to-Code
Gvero & Kuncak (2015)	Language	Syntax-Level	PCFG + Search	Code Synthesis

⁴Described in Chapter 4.

⁵Described in Chapter 7.

Table 2.2: Research on Source Code Generating Models (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\mathbb{C}|\mathcal{C}(\mathbb{C}))$ and describe the process for generating source code. References annotated with * are also included in other categories.

Reference	Model Type	Kind	Model ($P_{\mathcal{D}}$)	Application
Hellendoorn et al. (2015)	Language	Token-Level	n -gram	Code Review
Hindle et al. (2012)	Language	Token-Level	n -gram	Code Completion
Hsiao et al. (2014)	Language	Program Dependency Graph	n -gram	Program Analysis
Ling et al. (2016)	Multimodal	Token-Level	Neural Network	Code Synthesis
Liu (2016)	Language	Token-Level	n -gram	Obfuscation
Karaivanov et al. (2014)	Translation	Token-level + grammar constraints	Phrase-Based SMT	Migration
Karpathy et al. (2015)	Language	Character-Level	Neural Network	–
Kushman & Barzilay (2013)	Multimodal	Token-Level	CCGs	Code Synthesis
Maddison & Tarlow (2014)	Language	Syntax-Level with variable resolution	Neural Network	–
Menon et al. (2013)	Multimodal	Syntax-Level	PCFG + annotations	Code Synthesis
Nguyen et al. (2013a)	Translation	Token-level	Phrase-Based SMT	Migration
Nguyen et al. (2013b)	Language	Token-Level + parse information	n -gram	Autocompletion
Nguyen et al. (2015)	Translation	Token-level + parse information	Phrase-based SMT	Migration
Nguyen & Nguyen (2015)	Language	Partial Program Dependency Graph	n -gram	Code Completion
Oda et al. (2015)	Translation	Syntax & Token-Based	Tree-to-String + Phrase-based SMT	Pseudocode Generation

Table 2.2: Research on Source Code Generating Models (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\mathbb{C}|\mathcal{C}(\mathbb{C}))$ and describe the process for generating source code. References annotated with * are also included in other categories.

Reference	Model Type	Kind	Model ($P_{\mathcal{D}}$)	Application
Pham et al. (2016)	Language	Bytecode-Level	HMM	Autocompletion
Raychev et al. (2014)	Language	Token-Level + Semantic Constraints	n -gram + RNN	Code Completion
Ray et al. (2016)	Language	Token-Level	Cache n -gram	Bug Detection
Raychev et al. (2016)	Language	Syntax-Level	PCFG + learned annotations	Code Completion
Saraiva et al. (2015)	Language	Token-Level	n -gram	–
Sharma et al. (2015)	Language	Token-Level	n -gram	Information Extraction
Tu et al. (2014)	Language	Token-Level	Cache n -gram	Code Completion
Wang et al. (2016c)	Language	Syntax-Level	Neural Network	Code Completion
White et al. (2015)	Language	Token-Level	Neural Network	–
Yadid & Yahav (2016)	Language	Token-Level	n -gram	Information Extraction

2.3.2.1 Structured Prediction

Structured prediction refers to the problem of joint prediction of a set of dependent variables where an underlying structure (*e.g.* characterized by a graphical model) defines the relationships among the variables. Such problems have been widely studied within machine learning and NLP and are also present in source code. Structured prediction methods make a set of assumptions about the nature and the existence of dependencies among the predicted variables. Raychev et al. (2015) represent a program dependence graph as a conditional random field (CRF) and jointly predict the types or names of JavaScript variables. Li et al. (2016) learn representations for the nodes of the heap graph by considering the heap graph structure and the interdependencies among the nodes. Kremenek et al. (2007) use a factor graph to learn and enforce specification of resource usage (*e.g.* files).

2.3.2.2 Distributed Representations

Distributed representations (Hinton, 1984) are widely used in NLP to represent natural language units, such as words and paragraphs (Mikolov et al., 2013a; Le & Mikolov, 2014). Distributed representations refer to arithmetic vectors or matrices where the meaning of an element is *distributed* across multiple units (*e.g.* the “meaning” of a vector is distributed in its components). This is in contrast to local representations (see Figure 2.1), where each element is uniquely represented with exactly one unit. Distributed representations are commonly used in machine learning because they tend to generalize better. Models that learn distributed representations make the assumption that the elements being represented and their relations “fit” well within the number of represented dimensions.

Probabilistic source code models have used distributed representations. For example, models that use distributed vector representations define a function $\mathbb{C} \rightarrow \mathbb{R}^D$ that maps each code unit to a D -dimensional vector. Mou et al. (2016) learn distributed vector representations to classify source code into coding assignments. Allamanis et al. (2015a) learn distributed vector representations for variables and methods and use them to predict their names. Gu et al. (2016) learn distributed vector representations of natural language queries and use them to predict the relevant API sequences. Li et al. (2016) learn distributed vector representations for heap elements and use them to generate potential formal specifications for the code that produced them. Finally, Piech et al. (2015) learn distributed matrix representations of source code assignments and use them to

generate feedback for students.

Some code-generative models of code also use distributed representations internally. For example the work of [Maddison & Tarlow \(2014\)](#) and other neural language models (*e.g.* [Dam et al. \(2016\)](#)) use distributed representations to describe the context while sequentially generating code. [Ling et al. \(2016\)](#) and [Allamanis et al. \(2015b\)](#) combine the code-context distributed representation with a distributed representations of other modalities (*e.g.* natural language) to synthesize code.

2.3.3 Pattern Mining Models of Source Code

Pattern mining models of source code infer the latent structure of a probability distribution

$$P_{\mathcal{D}}(f(\mathbb{C})) = \sum_{\mathbf{I}} P_{\mathcal{D}}(f(\mathbb{C})|\mathbf{I})P(\mathbf{I}) \quad (2.3)$$

where f is some deterministic function that returns a (possibly partial, *e.g.* API calls only) view of the code and \mathbf{I} represent a set of latent variables that the model introduces and aims to infer. These models can be broadly thought as clustering source code into a finite set of groups. The goal of these probabilistic models of source code is to mine a finite set of human-interpretable patterns within the structure of source code, without using any annotations or supervision and present the mined patterns to software engineers. Applications of such models are common in the mining software repositories community and include documentation (*e.g.* API patterns), summarization and anomaly detection. There is a large amount of literature within unsupervised *non-probabilistic* models of source code taking advantage of data mining methods, such as frequent pattern mining and anomaly detection. These models are excluded from this discussion, since they are *not* probabilistic models of source code. Table 2.4 presents a list of related work in this area.

[Allamanis & Sutton \(2014\)](#) learn a tree substitution grammar using Bayesian non-parametrics and find groups of grammar productions that are more probable than predicted by a simple PCFG. They name the groups of grammar productions *idioms*. In a similar fashion, [Fowkes & Sutton \(2015\)](#) learn the latent variables of a graphical model to infer common API usage patterns. [Movshovitz-Attias & Cohen \(2015\)](#) infer the latent variables of a graphical model that models a software ontology.

Evaluation of pattern mining models is usually harder, since the discovered latent structure quality may be subjective. Thus, application-level metrics are often used.

Table 2.3: Research on Representational Models of Source Code (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\pi|f(\mathbb{C}))$. References annotated with * are also included in other categories.

Reference	Input Code Representation	Target Prediction (π)	Representation Type	Application
Allamanis et al. (2015a) ⁶	Token Context & Features	Variable/Method/Class Names	Distributed	Coding Conventions
Allamanis et al. (2015b)*	Natural Language & AST Context	Language Model (AST)	Distributed	Code Search
Allamanis et al. (2016d) ⁷	Code Tokens	Method Name	Distributed	Coding Conventions & Summarization
Bichsel et al. (2016)	Dependency Graph	Identifier Names	Graphical Model	Deobfuscation
Bruch et al. (2009)	Incomplete Object Usage	Object Usage	Local	Code Completion
Corley et al. (2015)	Tokens	Feature Location	Distributed	Feature Location
Dam et al. (2016)*	Tokens & Code Context	Language Model (Tokens)	Distributed	–
Gu et al. (2016)	Natural Language	API Sequence	Distributed	API Search
Gupta et al. (2017)	Tokens	Code Fix	Distributed	Code Fixing

⁶Described in Chapter 4 and Chapter 5.

⁷Described in Chapter 5.

Table 2.3: Research on Representational Models of Source Code (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\pi|f(\mathbb{C}))$. References annotated with * are also included in other categories.

Reference	Input Code Representation	Repre-	Target Prediction (π)	Representation Type	Application
Iyer et al. (2016)	Code Tokens		NL Description	Distributed	Code Summarization
Kremenek et al. (2007)	Partial PDG		Resource Ownership	Graphical Model	Pointer Ownership Bugs
Li et al. (2016)	Heap Graph		Separation Logic Formula	Distributed	Verification
Maddison & Tarlow (2014)*	Code Context		Language Model (AST)	Distributed	–
Mangal et al. (2015)	Logic Static Analysis + User Feedback	Analy-	Probabilistic Static Analysis	MaxSAT	Parametric Program Analysis
Movshovitz-Attias et al. (2013)	Code Tokens		Source Code Comments	Graphical Model	Comment Prediction
Mou et al. (2016)	AST (no identifiers)		Discrete Classification	Distributed	Coding Assignment Classification
Nguyen et al. (2016b)	API Sequences		API Sequences	Distributed	API Migration
Omar (2013)	Syntactic Context		Expressions	Graphical Model	Code Completion
Oh et al. (2015)	Features		Static Analysis Parameters	Static Analysis	Parametric Program Analysis
Piech et al. (2015)	Program + Source Features	State Code	Student Feedback	Distributed	Student Feedback

Table 2.3: Research on Representational Models of Source Code (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(\pi|f(\mathbb{C}))$. References annotated with * are also included in other categories.

Reference	Input Code Representation	Repre-	Target Prediction (π)	Representation Type	Application
Proksch et al. (2015)	Incomplete Usage	Object	Object Usage	Graphical Model	Code Completion
Raychev et al. (2015)	Dependency work	Net-	Variable Types/Names	Graphical Model	Type Inference
Wang et al. (2016a)	Code Tokens		Defect Classification	Language Model	Bug Detection
White et al. (2015)*	Tokens & Language Model (Tokens)		Distributed	–	
White et al. (2016)*	Tokens & AST		(pattern mining)	Distributed	Clone Detection
Zaremba & Sutskever (2014)	Code Characters		Execution Trace	Distributed	–

Table 2.4: Research on Pattern Mining Probabilistic Models of Source Code (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(f(\mathbb{C}))$. References annotated with * are also included in other categories.

Reference	Modeled Distribution $P_{\mathcal{D}}$	Model Type	Application
Allamanis & Sutton (2014) ^{*8}	AST	Graphical Model	Idiom Mining
Fowkes & Sutton (2015)	API Call Sequences	Graphical Model	API Mining
Movshovitz-Attias & Cohen (2015)	Software Ontology/Facts	Graphical Model	Knowledge-Base Mining
Fowkes et al. (2014)	Topic Hierarchy	Graphical Model	Code Summarization
Wang et al. (2016b)	Defect Classification	Distributed	Defect Prediction
White et al. (2016) [*]	Code Patterns	Distributed	Clone Detection

⁸Described in Chapter 7.

2.4 Applications

Probabilistic models of source code have found a wide range of applications in software engineering and programming language research. In this section, we review common application areas of those models. Table 2.5 presents a list of areas where probabilistic models of source code have been applied. Probabilistic models of source code provide a principled way of resolving uncertainty and ambiguity that is present in some applications. Such ambiguities arise from underspecified data or from inherently ambiguous information, such as natural language. A second set of domains that profit from probabilistic source code models are those that can use probabilistic inference to simplify or speed up an analysis that would otherwise be prohibitively complex to execute.

The aim of this section is not to review the application areas in detail but to explain the intersection of each areas with this field. For each application, we make an effort to describe the goals of the area and the interesting and important problems within that area. Then we focus on why probabilistic, machine learning based methods can be of use. For each area, we then discuss the existing research of probabilistic source code modeling and how performance is measured in an automated and quantifiable way.

2.4.1 Recommender Systems

Software engineering recommender systems ([Robillard et al., 2010, 2014](#)) are a wide range of tools that make smart recommendations to assist software engineers. A large part of those systems employ data mining and machine learning approaches on various software engineering artifacts. Probabilistic models of source code find application in source code-based recommender systems ([Mens & Lozano, 2014](#)), such as those that aid developers write or maintain code.

Machine learning-based recommender systems probabilistically reason about the intentions of the software engineer to aid them write or edit code. This is inherently “noisy” since it can neither be formulated clearly nor any developer would spend the time to clearly formulate their intentions before writing any code. Probabilistic source code model-based recommender systems use information from the *context* of the code to probabilistically reason about developer intentions. Although this is a challenging task, requiring to take into consideration both long and short-range context, machine learning methods are well fit for this problem providing a framework for probabilistic reasoning of developer intentions.

The most prominent recommender system and a feature commonly used in inte-

Table 2.5: Application Areas of Machine Learning Probabilistic Models of Source Code. The use of a probabilistic method allows to probabilistically reason about the uncertainty within the modeled domain and data.

Area	Modeled Domain	Source of Uncertainty	Performance Measures
Code Defects	Normal vs. anomalous usage of code constructs.	Sparsity of possible operations, code specifications.	Correlation with real faults
Code Migration	Mapping between different programming languages and APIs.	Multiple possible mappings, subtle differences in API usage, sparsity.	Functional equivalence, BLEU score
Code Search	Mapping from natural language and/or other code artifact queries to real code.	Ambiguities and partial information in queries. Non-fully resolved source code.	Information Retrieval Metrics (MRR)
Code Summarization	Selection of important aspects of code and possibly transformation to an appropriate summary format	Diversity of source code, domain specificity, uncertainty about code saliency.	Quality of Summary, BLEU/ROUGE score (if NL summary)
Code Synthesis	Possible programs that comply to some specification.	Partial or ambiguous (<i>e.g.</i> natural language description) information about program to be synthesized.	Synthesized program quality
Coding Conventions	Coding conventions (<i>e.g.</i> naming, formatting, syntactic) used within a project.	Non-strict usage of conventions within projects, domain variability, differences in domains.	Prediction/suggestion accuracy

Table 2.5: Application Areas of Machine Learning Probabilistic Models of Source Code. The use of a probabilistic method allows to probabilistically reason about the uncertainty within the modeled domain and data.

Area	Modeled Domain	Source of Uncertainty	Performance Measures
Documentation	Reusable patterns of source code.	Sparse and “noisy” usages of APIs and code constructs	Pattern quality, informativeness, usefulness
Program Analysis	Program properties	Partial or ambiguous information about program behavior during runtime and/or prohibitive complexity of an exact analysis	Prediction accuracy, precision, recall, false positive ratio
Recommender Systems	Developer intention when writing code through current context.	Partial information about developer intention within the domain of the developed software, long-distance context.	Suggestion Accuracy, MRR, Cross Entropy

grated development environments (IDEs) is *code completion*. All widely used IDEs, such as Eclipse, IntelliJ and Visual Studio, have some code completion features. According to [Amann et al. \(2016\)](#), code completion is the most used feature within an IDE. However, code completion tools usually return suggestions alphabetically by sorting valid options, without using any predictive models to predict how related each suggestion is. Statistical code completion aims to improve suggestions accuracy by learning probabilities over the suggestions and providing to the users a ranked list. Some systems focus on automatically completing specific constructs (*e.g.* method calls and parameters) while others try to complete all code tokens.

Statistical code completion was first studied by [Bruch et al. \(2009\)](#) who extract a set of features from code context to make predictions but are limited to suggesting method invocations and constructors. Later, [Proksch et al. \(2015\)](#) used Bayesian networks to improve suggestion accuracy. A version of this research is integrated into the default Eclipse IDE under the Eclipse Recommenders team.

Source code language models have implicitly and explicitly been used for code completion. [Hindle et al. \(2012\)](#) were the first to use a token-level n -gram LM for this purpose, using the previous $n - 1$ tokens to represent the completion context at each location. Later, [Tu et al. \(2014\)](#) used a cache n -gram LM and further improved the completion performance, showing that a local cache acts as a domain adapted n -gram. The authors provide an Eclipse plugin using their model ([Franks et al., 2015](#)). [Nguyen et al. \(2013b\)](#) augment the completion context with semantic information, improving the code completion accuracy of the n -gram LM. [Raychev et al. \(2014\)](#) exploit formal properties of the code in context to limit incorrect (but statistically probable) API call suggestions. Their method is the first to depart from simple statistical token completion towards statistical code synthesis of single statements. Apart from token-level language models for code completion, [Bielik et al. \(2016\)](#) and [Maddison & Tarlow \(2014\)](#) create AST level LMs that can be used for suggestion.

In contrast to the previously mentioned work which aims to predict source code, [Movshovitz-Attias et al. \(2013\)](#) create a recommender system to assist comment completion given a source code snippet, using a topic-like graphical model to model context information. Similarly, the work of [Allamanis et al. \(2014, 2015a, 2016d\)](#) can be seen as a recommender systems for suggesting names for variables, methods and classes, that uses relevant code tokens as the context.

Evaluation Metrics Research in recommender systems has used various metrics to quantify code completion performance:

Accuracy for Rank is the percent of suggestion points where the top x suggestions contain the correct prediction.

Mean Reciprocal Rank (MRR) is the average reciprocal rank of the correct recommendation at each suggestion point.

Cross Entropy For probabilistic recommender systems, the probability for each suggestion can be computed and therefore cross-entropy. Although cross entropy is correlated with suggestion accuracy and confidence, small improvements in cross entropy may not lead to improvements in accuracy.

Keystrokes Saved Computes the number of keystrokes saved by using a code completion system. This metric usually assumes that code completion suggestions are continuously presented to the user as she is typing. When the top suggestion is the target token, the user presses a single key (*e.g.* return) to complete the rest of the target token.

Memory and Speed For practical recommender systems, the (memory) size of the model and the required time for a single recommendation need to be taken into account. Therefore, the trade-offs between model accuracy, completion speed and size need to be taken into consideration.

These metrics provide good estimations on the predictive ability of recommender systems. However, in some cases the evaluations are not entirely representative of the usage of the systems. For example, one limitation of these metrics used for code completion recommender systems is that they assume that code is written sequentially, from the first token to the last one. In practice, however, rarely do developers write code in such a simplified way ([Proksch et al., 2016](#)).

2.4.2 Inferring Coding Conventions

The surface structure of the source code is formed by the coding conventions that the developers implicitly or explicitly impose during the development process ([Allamanis et al., 2014](#)). The goal of coding conventions is to improve code maintainability by making the code easy to comprehend. Enforcing coding conventions is also needed for educational purposes, *e.g.* in massive, open, online courses ([Glassman et al., 2015](#)). However, enforcing coding conventions — such as formatting and naming — is a tedious task and in some cases it cannot be easily codified in rule-based systems. Inferring

coding conventions with machine learning solves this problem by *learning* the conventions directly from a codebase and then enforcing them, in a probabilistic manner. This can help software teams to enforce coding conventions without the need to define rules or configure existing convention enforcing tools.

Machine learning models of source code that look at the surface structure (*e.g.* tokens, syntax) are inherently well-suited for this task. Using the source code as data, they can infer the emergent conventions while quantifying any uncertainty over those decisions. An important challenge in this application domain is the sparsity of the code constructs, caused by the diverse and non-repeatable form of source code within projects and domains. Allamanis et al. (2014, 2015a) exploit this fact to learn and suggest variable, method and class naming conventions, while Allamanis & Sutton (2014) mine conventional syntactic structures named *idioms*.

Related to coding conventions, Parr & Vinju (2016) learn a source code formatter from data by using a set of hand-crafted features from the AST using a k -NN classifier. White et al. (2016) use autoencoders and recurrent neural networks to detect code clones by finding snippets of code that share similar distributed representations.

2.4.3 Code Defects and Debugging

Probabilistic models of source code assign high probability to code that appears often in practice (*i.e.* is natural). Therefore, when models assign a very low probability to some code, this may signify a bug, similar to all anomaly detection methods in machine learning (Chandola et al., 2009). Finding defects is a core problem in software engineering and programming language research. The challenge in this domain rests in correctly characterizing source code that contains defects with high precision and recall. This is especially difficult because of the sparse and extremely diverse nature of source code.

Some limited work, suggests that the probability assigned by language models may be related to code defects. Allamanis & Sutton (2013b) suggest that n -gram LMs can be seen as complexity metrics and Ray et al. (2016) show that buggy code tends to have lower probability (be less “natural”) than average code.

Wang et al. (2016b) use deep belief networks to learn source code features for code defect prediction. Fast et al. (2014) and Hsiao et al. (2014) learn statistics from large amounts of code to detect potentially erroneous code and perform program analyses while Wang et al. (2016a) learn coarse-grained n -gram language models to detect

uncommon usages of code. Related to this work, is the work of [Campbell et al. \(2014\)](#) and [Bhatia & Singh \(2016\)](#) that use source code LMs to identify and correct syntax errors. Other data-mining based methods (*e.g.* [Wasylkowski et al. \(2007\)](#)) also exist, but are out-of-scope from this review.

[Liblit et al. \(2005\)](#); [Zheng et al. \(2006\)](#) present trace-based methods for statistical bug isolation. These methods aim to isolate the possible bug locations to assist debugging. [Kremenek et al. \(2007\)](#) learn factor graphs to predict resource-specific bugs by modeling the resource usage specifications.

Although probabilistic models of source code seem to be naturally fitted for finding defective code, this area has not yet seen much growth or real-life usage, possibly because of issues that arise from the high dimensionality and sparsity of code data.

2.4.4 Code Migration

Code migration refers to translating source code from one source language (*e.g.* Java) to another target language (*e.g.* C#). Although rule-based, rewriting systems may be used, it is a tedious process to maintain those rules. Statistical machine translation models, as described earlier, are well suited for this task, since they probabilistically learn to convert one language to another. However, because of the probabilistic nature of these models, they tend to produce invalid code. To restrict these errors during translation [Karaivanov et al. \(2014\)](#) and [Nguyen et al. \(2015\)](#) add semantic constraints to the translation process. Although these models learn powerful mappings between different language constructs such as APIs, they have only been used for translating between programming languages of similar paradigms and structure. This is an important limitation, requiring novel models to translate between languages of different types (*e.g.* Java to Haskell or assembly to C) or even languages of different memory management styles (*e.g.* C++ to Java). Evaluation of translation methods is done using BLUE ([Papineni et al., 2002](#)), exact match scores and measure of the syntactic or semantic correctness of the translated code.

2.4.5 Source Code and Natural Language

Linking natural language to source code has many useful applications, such as code synthesis, search and summarization. However, the ambiguity of natural language, the highly diverse and compositional nature of source code, and the layered abstractions being built in software makes connecting natural language and source code a hard

problem. Probabilistic machine learning models of source code, have already been useful in NLP by probabilistically modeling and resolving ambiguities. The application of these models to the combination of source code and natural language follows the same principle. Additionally, the intersection of source code and natural language is viewed as a way for grounding natural language within the NLP community. [Oda et al. \(2015\)](#) translate Python code to natural language pseudocode and use the BLEU score to evaluate their approach. Similarly, [Iyer et al. \(2016\)](#) design a neural attention model that can generate natural language summaries of source code, using BLEU score to evaluate their summaries and MRR to evaluate their model as a code search model. [Gulwani & Marron \(2014\)](#) use natural language to synthesize Excel formulas and evaluate on the rank of the correctly synthesized program. [Movshovitz-Attias et al. \(2013\)](#) create a comment-generative model, given some source code and evaluate it on its suggestion performance. This area is closely related to semantic parsing in NLP, where natural language is parsed into a program-like structure that is defined by a DSL. These models are out-of-scope from this review. For related work see [Neubig \(2016\)](#).

Code search — a common activity for software engineers ([Sadowski et al., 2015](#); [Amann et al., 2016](#)) — may also involve searching source code using natural language. Software engineering researchers have focused on the code search problem using information retrieval (IR) methods ([Gallardo-Valencia & Elliott Sim, 2009](#); [Holmes et al., 2005](#); [Thummalapenta & Xie, 2007](#); [Mcmillan et al., 2013](#)). While [Niu et al. \(2016\)](#) has used learning-to-rank methods but with manually extracted features. Within probabilistic modeling of source code, [Gu et al. \(2016\)](#) train a sequence-to-sequence neural network to map natural language into API sequences. [Allamanis et al. \(2015b\)](#) learn a bimodal, generative model of code, conditioned on natural language and use it to rank code search results. All code search-related methods use rank measures to evaluate performance.

2.4.6 Program Synthesis

Program synthesis aims to synthesize full or partial programs from some form of specification. When the specification is in the form of an ambiguous natural language description these models coincide with semantic parsing (see Subsection 2.4.5). The program synthesis (*e.g.* from examples or from a specification) has received great attention in programming language research. The core challenge in this area is searching the vast area of possible programs to find one that complies to the specification. Proba-

bilistic machine learning models help by learning probabilities over possible programs guiding the search process.

Research on programming by example (PBE) has combined machine learning methods with code synthesis. [Liang et al. \(2010a\)](#) use a graphical model to learn programs across similar tasks. [Menon et al. \(2013\)](#) learn a parameterized PCFG by using features from the input to speed up synthesis. [Singh & Gulwani \(2015\)](#) extract features from the synthesized program to learn a supervised classifier that can predict the correct program and use it to re-rank synthesis suggestions. Finally, the code completion work of [Raychev et al. \(2014\)](#) can be seen as a limited program synthesis of method invocations within specific locations.

2.4.7 Documentation and Summarization

Documentation is an important artifact within the software development process, allowing software engineers to find information they require when developing code. Software engineering research has a multitude of related work. Mining common API patterns is a recurring theme when mining software repositories and there is a large literature of non-probabilistic models (*e.g.* frequency-based models) for mining and synthesizing API patterns ([Buse & Weimer, 2012](#); [Xie & Pei, 2006](#)) which are out-of-scope of this review. Within this literature, there are a few probabilistic source code models that mine API sequences. [Gu et al. \(2016\)](#) maps natural language to commonly used API sequences, [Allamanis & Sutton \(2014\)](#) learn fine-grained source code idioms, that may include APIs. [Fowkes & Sutton \(2015\)](#) uses a graphical model to mine interesting API sequences.

Documentation is also related to information extraction from (potentially unstructured) documents. [Cerulo et al. \(2015\)](#) use a language model to detect code “islands” in free text. [Sharma et al. \(2015\)](#) use a language model over tweets to identify software-relevant tweets.

2.4.8 Program Analysis

Program analysis is a core research area in programming language research, with the goal to analyze programs and provide some indications about their properties such as their correctness. Various challenges arise in this domain. For example, sound analyses may return unacceptably large amounts of false positives or other analyses may require to combine multiple ambiguous data. Probabilistic models of source code, aim to use

probabilistic reasoning to alleviate these problems.

Raychev et al. (2015) use graphical models to predict the types of variables in JavaScript by learning usage patterns from existing code and combining ambiguous information return a probability distribution over variables types. Oh et al. (2015) and Mangal et al. (2015) use machine learning models to parameterize program analyses to reduce false positive ratio while maintaining high precision.

2.5 Conclusions

This chapter reviewed probabilistic models of source code. We presented a taxonomy of probabilistic machine learning source code models and their applications. The reader may appreciate that most of the research contained in this review was conducted within the past few years, indicating a growth of interest in this area among the machine learning, programming languages and software engineering communities. Probabilistic models of source code raise the exciting opportunity of *learning* from existing code, probabilistically reasoning about new source code artifacts and transferring knowledge between developers and projects.

Obstacles and Challenges Although this field has made considerable progress, many challenges still need to be overcome. One major obstacle is engineering systems that efficiently combine the probabilistic world of machine learning with the formal logic-based world of code analysis. Additionally, although finding a large amount of source code is relatively easy, it is increasingly hard to retrieve sophisticated representations of source code. For example, computing semantic properties of code is hard to do at a truly large scale. Similarly, retrieving representative run-time data from real-life programs is challenging to do even for a single project. Furthermore, the principle of reusability in software engineering creates a form of sparsity in the data, where it is rare to find multiple source code elements that perform exactly the same tasks. From a machine learning perspective this suggests there are many opportunities for creating new models and inference methods that are able to handle the structured, sparse and highly composable nature of source code data. From the perspective of software engineering, interesting research questions arise on creating usable tools that exploit the probabilistic reasoning capabilities that these models can offer. Finally, from a programming language research perspective, novel probabilistic formulations and representations may allow to revisit existing problems and provide probabilistic bounds on important problems or speed up existing formal methods.

Chapter 3

Background: Coding Conventions

“Programs share some attributes with essays. For essays, the most important question readers ask is, “What is it about?”. For programs, the main question is, “What does it do?”. In fact, the purpose should be sufficiently clear that neither question ever needs to be uttered [...] Both essays and lines of code are meant —before all else— to be read and understood by human beings.”

– Beautiful Code: Leading Programmers Explain How They Think. 2007

To program is to make a series of choices, ranging from design decisions — like how to decompose a problem into functions — to the choice of identifier names and how to format the code. Coding conventions is an important aspect of the software development process. This is attested by the multitude of books — targeted to practitioners — that have been written about this topic ([Martin, 2008](#); [McConnell, 2004](#); [Spinellis, 2003](#)) and the abundance of coding style guides available online ([Rossum et al., 2013](#); [Oracle, 1999](#); [Association et al., 2012](#); [AirBnb, 2015](#)).

A convention is “an equilibrium that everyone expects in interactions that have more than one equilibrium” ([Young, 1996](#)). Coding conventions arise out of the collision of the stylistic choices of programmers. A *coding convention* is a restriction *not* imposed by a programming language’s grammar. Nonetheless, these choices are important enough that they are enforced by software teams. Indeed, developers enforce such coding conventions rigorously, with roughly one third of code reviews containing feedback about following them ([Allamanis et al., 2014](#)).

Like the rules of society at large, coding conventions fall into two broad categories: *laws*, explicitly stated and enforced rules, and *mores*, unspoken common practice that emerges spontaneously. Mores pose a particular challenge: because they arise spontaneously from emergent consensus, they are inherently difficult to codify into a fixed set of rules, so rule-based formatters cannot enforce them, and even programmers themselves have difficulty adhering to all of the implicit mores of a codebase. Furthermore, popular code changes constantly, and these changes necessarily embody stylistic decisions, sometimes generating new conventions and sometimes changing existing ones. Conventions are pervasive in software, ranging from preferences about identifier names to preferences about formatting, object relationships, programming practices and design patterns.

Coding conventions decisions determine the readability of a program's source code, increasing a codebase's portability, its accessibility to newcomers, its reliability, and its maintainability (Oracle, 1999, §1.1). *"The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of an application so that you and others can easily read and understand the code. Good coding conventions result in precise, readable, and unambiguous source code that is consistent with other language conventions and as intuitive as possible."*¹ Apple's recent, infamous bug where a single line was wrongly indented in the SSL certificate handling (Arthur, 2014; Langley, 2014) exemplifies the impact that formatting can have on reliability. Maintainability is especially important since developers spend the majority (80%) of their time maintaining code (Bourque et al., 2014, §6).

This section reviews the related research and practice in coding conventions. First, we discuss naming (Section 3.1) and formatting (Section 3.2) conventions, that are widely discussed across the majority of the projects. We then focus on code idioms and design patterns (Section 3.3). Finally in Section 3.4, we review how coding conventions are used in practice, discussing related tools.

3.1 Naming Coding Conventions

Selecting an appropriate, descriptive name for an identifier is an important problem; names connect program source to its problem domain (Binkley et al., 2009; Lawrie et al., 2006b; Liblit et al., 2006; Takang et al., 1996) and appeal to the verbal abilities

¹MSDN Documentation [https://msdn.microsoft.com/en-us/library/aa733744\(v=vs.60\).aspx](https://msdn.microsoft.com/en-us/library/aa733744(v=vs.60).aspx)

of humans (Siegmund et al., 2014). Naming is a particularly active topic of concern among developers, for example, Allamanis et al. (2015a) found that almost *one quarter* of the code reviews contain discussions about naming. Developers constantly rename identifiers to reflect better reflect their roles (Arnaoudova et al., 2014), a task that 92% of their participants found “not straightforward”.

Naming in code has achieved a fair amount of research attention. High quality identifier names lie at the heart of software engineering (Anquetil & Lethbridge, 1999; Brooks, 1975; Caprile & Tonella, 2000; Deissenboeck & Pizka, 2006; Lawrie et al., 2007; Soloway & Ehrlich, 1984; Takang et al., 1996; Ohba & Gondow, 2005); they drive code readability and comprehension (Biggerstaff et al., 1993; Buse & Weimer, 2010; Caprile & Tonella, 2000; Lawrie et al., 2006a; Liblit et al., 2006; Takang et al., 1996). According to Brooy et al. (2005), identifiers represent the majority (70%) of source code tokens. Allamanis & Sutton (2013b) found that identifiers is the hardest set of tokens to be predicted within code. Eshkevari et al. (2011) and Arnaoudova et al. (2014) explored how identifiers change in code, while Lawrie et al. (2006a) studied the consistency of identifier namings. Abebe et al. (2011) discuss the importance of naming to concept location arguing that if the set of tokens used within a software system is relatively low, removing “lexicon smells” can improve feature location methods. Caprile & Tonella (2000) propose a framework for restructuring and renaming identifiers based on custom rules and dictionaries. Gupta et al. (2013) present part-of-speech tagging on split multi-word identifiers to improve software engineering tools. Because longer names are more informative (Liblit et al., 2006), these styles share an agglutination mechanism for creating multi-word names (Anquetil & Lethbridge, 1998; Ratiu & Deißeböck, 2007). Some programming languages, such as Go, even assign semantic effects to names, such as visibility. Additionally, several styles exist for engineering consistent identifiers (Binkley et al., 2009; Caprile & Tonella, 2000; Deissenboeck & Pizka, 2006; Simonyi, 1999).

There has been prior research into identifying poorly named artifacts. Høst & Østvold (2009) developed a technique for automatically inferring naming rules for methods based on the return type, control flow, and parameters of methods. Using these rules they found and reported “naming bugs” by identifying methods whose names contained rule violations. Naming bugs are mismatches between the map and stemmed and tagged method names and their predicates in a test set. Arnaoudova et al. (2013) presented a catalog of “linguistic anti-patterns” in code that lead to developers misunderstanding code and built a detector of such anti-patterns. Binkley et al. (2011) used

part of speech tagging to find field names that violate accepted patterns, *e.g.* the field `create_mp4` begins with a verb and implies an action which is a common pattern for a method rather than a field.

De Lucia et al. (2012) attempted to automatically name source code artifacts using LSI and LDA and found that this approach doesn't work as well as simpler methods such as using words from class and method names. Many studies of naming have also been conducted giving us insight into its importance. Butler et al. (2010) found that "flawed" identifier names (those that violate naming conventions or do not follow coding practice guidelines) are related to certain types of defects, such as the warnings from the FindBugs static analysis tool. Later they also examined the most frequent grammatical structures of method names using part of speech tagging (Butler et al., 2011). Lawrie et al. (2006a) and Takang et al. (1996) both conducted empirical studies and concluded that the quality of identifier names in code has a profound effect on program comprehension and good names are related to a concept mapping of the domain of the software. Liblit et al. (2006) explored how names in code *"combine together to form larger phrases that convey additional meaning about the code."* Arnaudova et al. (2014) studied identifier renamings, showing that naming is an important part of software construction. Additionally, in a survey of 94 developers, they found that about 68% of developers think that recommending identifiers would be useful. This is the problem that Allamanis et al. (2014, 2015a)² aim to solve. They create a machine learning framework that learns naming conventions directly from a codebase, without the need for providing any a priori knowledge about acceptable or unconventional naming styles.

3.2 Formatting Coding Conventions

Formatting decisions usually capture control flow and are important in facilitating code understanding (Martin, 2008). Figure 3.1 shows a sample of two different options for formatting style in Java. Although formatting decisions are highly subjective, maintaining a consistent style within a project is widely accepted. The importance of formatting conventions and especially indentation is indicated by the fact that some programming languages (*e.g.* Python) attach semantic meaning to formatting (*e.g.* indentation). Badly formatted code can even introduce bugs, as exemplified by the well-known Apple SSL bug, where a misleading indentation caused a security vulnerability (Arthur, 2014).

² Presented in this dissertation in Chapter 4 and Chapter 5.

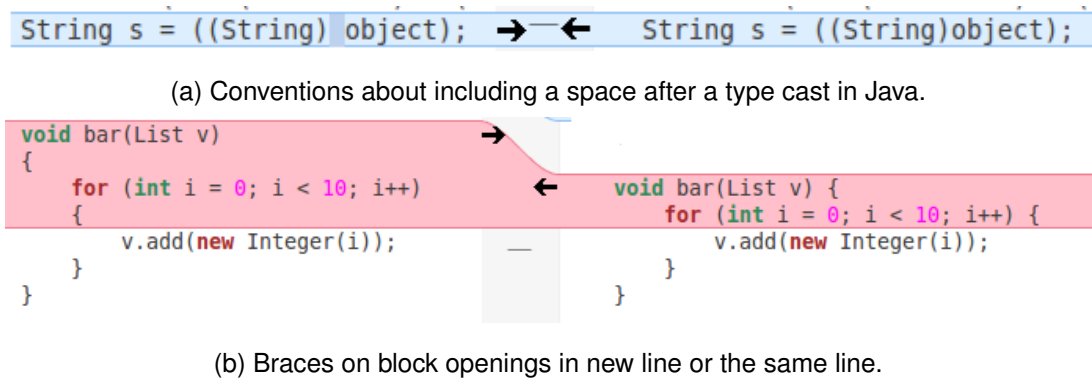


Figure 3.1: Different formatting coding conventions

Formatting conventions are not widely studied within the academic literature, although are widely enforced by practitioners. [Hindle et al. \(2009\)](#) find that indentation relates with the readability of the code. [Wang et al. \(2011\)](#) developed a heuristic-based method to automatically insert blank lines into methods (*i.e.* vertical spacing) to improve readability. [Allamanis et al. \(2014\)](#) infer the emergent formatting conventions of a project using an n -gram LM. Finally, [Parr & Vinju \(2016\)](#) create a machine learning-based formatter that learns to format the code by extracting features from the context (*e.g.* parse tree nodes) of each formatting decision point.

3.3 Coding Patterns

Apart from naming and formatting, conventions include the conventional use of various coding constructs. For example, by conventions the well-known pattern `for(i=0;i<N;i++)` is perfected in C when iterating through arrays compared to using a `while` loop. Different languages and projects have different conventions. [Fowler \(2009\)](#) presents a set of “code smells” *i.e.* patterns that should be avoided. Python core developer Raymond Hettinger, emphasizes how to convert classic coding constructs into more “pythonic” forms that are more “beautiful and idiomatic” ([Hettinger, 2013](#)). For example, instead of using a temporary variable for swapping the values of two variables `var1` and `var2` one can use the pythonic assignment

```
var1, var2 = var2, var1
```

which is more clear and conventional in Python.

Related to this research, is the work of [Allamanis & Sutton \(2014\)](#)³ who mine

³ Presented in this dissertation in Chapter 7.

syntax-level patterns that involve both basic code constructs and APIs. API-specific patterns can also be viewed as a form of coding conventions. The frequent use of specific patterns is highly encouraged and has resulted into multiple pieces of work that aim to mine API patterns (*e.g.* in sequences or graphs) (Acharya et al., 2007; Holmes et al., 2006; Zhong et al., 2009; Wang et al., 2013; Fowkes & Sutton, 2015). The pervasiveness of those patterns, allows a set of (statistical) tools such as API recommenders, code suggestion and completion systems that aim to help during editing, when a user may not know the name of an API she needs (Robillard et al., 2010), the parameters she should pass to the API (Zhang et al., 2012), or that the API call she is making needs to be preceded by another (Gabel, 2011; Uddin et al.; Wang et al., 2013; Zhong et al., 2009).

Research in this area has focused mostly on detecting potential API usage errors, via mining patterns from existing source code. Fast et al. (2014) learns statistics from a large scale corpus of Ruby to provide statistical linting, flagging uncommon usages of APIs. Wasylkowski et al. (2007); Gruska et al. (2010) create object usage models and learn a classifier to detect potentially erroneous code. Mishne et al. (2012) mine temporal API specifications and track the typestate of each variable. Livshits & Zimmermann (2005) analyzes source code history to find method calls that are added/removed simultaneously and common bug fixes, discovering application-specific coding conventions. Yang et al. (2006) mine temporal API rules from traces to discover bugs. Nguyen et al. (2009) mine graphs of API usages and detect anomalies within the code.

Design patterns is a set of coding conventions that discuss architectural patterns of the code and aim to introduce reusable and easily understandable concepts. Using those design patterns is a conventional practice, introducing a “vocabulary” that is well understood among developers (Beck & Cunningham, 1987). Gamma et al. (1995) introduce design patterns for object oriented programming. Hohpe & Woolf (2004); Buschmann et al. (2007) extend those patterns. There is a large amount of work around design patterns that would be impossible to exhaustively discuss here. See Bass (2007) for a more detailed description. Finally, Dong et al. (2009) present a review of tools for mining design patterns.

3.4 Enforcing Coding Conventions in Practice

Coding conventions are standard practice (Boogerd & Moonen, 2008; Hatton, 2004) among practitioners. They facilitate consistent measurement and reduce systematic er-

ror and generate more meaningful commits by eliminating trivial convention enforcing commits ([Wikipedia](#)). Some programming languages like Java and Python suggest specific coding styles ([Oracle, 1999](#); [Rossum et al., 2013](#)), while consortia and companies publish guidelines for others, like C ([Association et al., 2012](#); [Hatton, 2004](#)), C++ ([Google, 2010](#)) and JavaScript ([AirBnb, 2015](#)).

Many rule-based *code convention enforcers* (also known as linters) exist but are usually limited to enforcing formatting, rule-based naming conventions as well as some best practices and logical errors (*i.e.* lightweight static analysis tools). The name is due to `lint`, one of the first of such tools. Linters aim to flag suspicious language constructs (*e.g.* non-portable or non-idiomatic use of constructs) combining this with some static analysis, thus encoding *best practices* that are common conventions. Such practices become a convention throughout each community or a project and the practitioners most usually correlate them with code readability, maintainability and as a strategy to avoid erroneous behavior. For example, “W0703: *Catching too general exception*” of `PyLint` is related to the good practice of catching vague exceptions that practitioners correlate with buggy behavior. Linting tools relate to research on static analyses, although such analyses usually focus on identifying bugs, rather than non-conventional “bad” practices. Additionally, linters check for rule-based formatting and naming conventions. For example, `PyLint` checks if names match a simple set of regular expressions (*e.g.*, variable names must be lowercase separated with underscores, if necessary) and whether a generic exception is caught (*i.e.* a bad practice); `astyle`, `aspell` and `GNU indent`⁴ only format whitespace tokens. `gofmt` formats the code providing an authoritative decision on formatting style and “eliminating an entire class of argument” among developers but provides no guidance for naming or other conventions. In contrast to linters that emphasize on style, there is a spectrum of tools that perform static analysis to find common bugs or bad practices, such as `FindBugs` ([Ayewah et al., 2007](#)), `Coverity` ([Bessey et al., 2010](#)) and many others. These tools are usually not considered within the realm of coding conventions

3.5 Conclusions

In this chapter, we presented work related to coding conventions in research and in practice. These conventions are important for software engineers since they help them maintain a consistent and maintainable codebase. The necessity for conventions arises

⁴<http://astyle.sourceforge.net/> and <http://www.gnu.org/software/indent/>

from the unconstrained nature of programming languages that allows a multitude of stylistic choices for expressing identical code semantics. Coding conventions constrain this space and provide a simplified and unified language that helps developers communicate through code. The pervasiveness of coding conventions among practitioners and the multitude of available tools suggest that there are many open research challenges that touch both software engineering and programming language research one of which this thesis tackles.

Chapter 4

Learning Variable Naming Conventions

“Programs must be written for people to read, and only incidentally for machines to execute.”
– Abelson & Sussman, “Structure and Interpretation of Computer Programs”

This chapter introduces NATURALIZE, a framework that addresses the coding convention inference problem for local conventions, offering suggestions to increase the stylistic consistency of a codebase. NATURALIZE is descriptive, not prescriptive¹: it learns what programmers actually do. When a codebase does not reflect consensus on a convention, NATURALIZE recommends nothing, because it has not learned anything with sufficient confidence to make recommendations. The naturalness insight of [Hindle et al. \(2012\)](#), building on [Gabel & Su \(2010\)](#), is that most short code utterances, like natural language utterances, are simple and repetitive. Large corpus statistical inference can discover and exploit this naturalness to improve developer productivity and code robustness. We show that coding conventions are *natural* in this sense.

Learning from local context allows NATURALIZE to learn syntactic restrictions, or sub-grammars, on identifier names like camelcase or underscore, and to statistically group together contexts where a name is used, something that rule-based code formatters simply cannot do. NATURALIZE is unique in that it does not require upfront agreement on hard rules but learns soft rules that are implicit in a codebase.

¹Prescriptivism is the attempt to specify rules for correct style in language, *e.g.*, [Strunk Jr & White \(1979\)](#). Modern linguists studiously avoid prescriptivist accounts, observing that many such rules are routinely violated by noted writers.

Intuitively, NATURALIZE works by identifying identifier names that are surprising according to a probability distribution over code text. When surprised, NATURALIZE determines if it is sufficiently confident to suggest a renaming that is less surprising; it unifies the surprising choice with one that is preferred in similar contexts elsewhere in its training set. NATURALIZE is *not* automatic; it assists a developer, since its suggestions, are potentially semantically disruptive and must be considered and approved. NATURALIZE's suggestions enable a range of new tools to improve developer productivity and code quality: 1) A pre-commit script that rejects commits that excessively disrupt a codebase's conventions; 2) An Eclipse plugin that a developer can use to check whether her changes are unconventional; and 3) A style profiler that highlights the stylistic inconsistencies of a code snippet for a code reviewer.

NATURALIZE draws upon a rich body of tools from statistical natural language processing (NLP), but applies these techniques to a different kind of problem. NLP focuses on *understanding* and *generating* language, but does not ordinarily consider the problem of improving existing text. The closest analog is spelling correction, but that problem is easier because we have strong prior knowledge about common types of spelling mistakes. An important conceptual dimension of our suggestion problems also sets our work apart from mainstream NLP. In code, rare names often usefully signify unusual functionality, and need to be preserved. We call this the *sympathetic uniqueness principle* (SUP): unusual names should be preserved when they appear in unusual contexts. We achieve this by exploiting a special token UNK that is often used to represent rare words that do not appear in the training set. Our method incorporates SUP through a clean, straightforward modification to the handling of UNK. Because of the Zipfian nature of language, UNK appears in unusual contexts and identifies unusual tokens that should be preserved. Section 4.4 demonstrates the effectiveness of this method at preserving such names.

As NATURALIZE detects identifiers that violate code conventions and assists in renaming — the most common refactoring (Murphy-Hill et al., 2012) — it is the first tool we are aware of that uses NLP techniques to aid refactoring.

The techniques that underlie NATURALIZE are language independent and require only identifying identifiers, keywords, and operators, a much easier task than specifying grammatical structure. Thus, NATURALIZE is well-positioned to be useful for domain-specific or esoteric languages for which no convention enforcing tools exist or the increasing number of multi-language software projects such as web applications that intermix Java, CSS, HTML, and JavaScript.

To the best of our knowledge, this work is the first to address the coding convention inference problem, to suggest names to increase the stylistic coherence of code, and to provide tooling to that end.

The machine learning perspective Learning to predict the name of a variable is not just an interesting problem for software engineering, but a problem with deep implications in machine learning and artificial intelligence. Predicting the name of a variable requires some understanding of the role and function of the variable within its context. All names — including variable names — carry semantic meaning and since good variable names help humans understand code, learning to name variables should be helpful for machine learning models that want to “understand” code.

The main contributions of this chapter are:

- We built NATURALIZE, the first framework to address the *variable naming convention inference problem* and suggests changes to increase a codebase’s adherence to its own conventions;
- We present two machine learning-based methods that are able to learn coding conventions. We first present a simple n -gram language model and then a neural log-bilinear context model that we design specifically for the variable naming. The log-bilinear model can further suggest subtoken naming conventions.
- We offer three tools, built on NATURALIZE, all focused on release management, an under-tooled phase of the development process.
- NATURALIZE achieves 94% accuracy in its top suggestions for variable names.

Tools are available at groups.inf.ed.ac.uk/naturalize.

4.1 Motivating Example

Both industrial and open-source developers often submit their code for review prior to check-in (Rigby & Bird, 2013). Consider the example of the class shown in Figure 4.1 which is part of a change submitted for review by a Microsoft developer on February 17th, 2014. While there is nothing functionally wrong with the class, it violates the coding conventions of the team. A second developer reviewed the change and suggested that `res` and `str` do not convey parameter meaning well enough, the constructor line is too long and should be wrapped. In the checked-in change, all of these were addressed, with the parameter names changed to `queryResults` and `queryStrings`.

```
1 public class ExecutionQueryResponse
2         : ExecutionQueryResponseBasic<QueryResults>
3 {
4     public ExecutionQueryResponse(QueryResults res,
5         IReadOnlyCollection<string> str,
6         ExecutionStepMetrics metrics) : base(res, str, metrics) { }
7 }
```

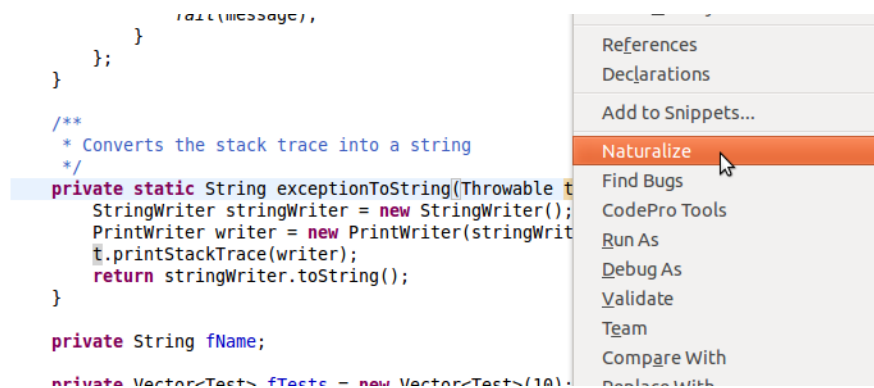
Figure 4.1: A C# class added by a Microsoft developer that was modified due to requests by a reviewer before it was checked in. Formatting of the snippet has been changed to fit the page

Consider a scenario in which the author had access to NATURALIZE. The author might highlight the parameter names and ask NATURALIZE to evaluate them. At that point it would have not only identified `res` and `str` as names that are inconsistent with the naming conventions of parameters in the codebase, but would also have suggested better names. After querying NATURALIZE about her stylistic choices, the author can then be confident that her change is consistent with the norms of the team and is more likely to be approved during review. Furthermore, by leveraging NATURALIZE, fellow project members would not need to be bothered by questions about conventions, nor would they need to provide feedback about conventions during review. We have observed that such scenarios frequently occur in code reviews ([Allamanis et al., 2014](#)). In Microsoft-internal code reviews about 38% of them discuss coding conventions while most of them (24% of the reviews) are just about naming. Similar results were also found on open-source projects.

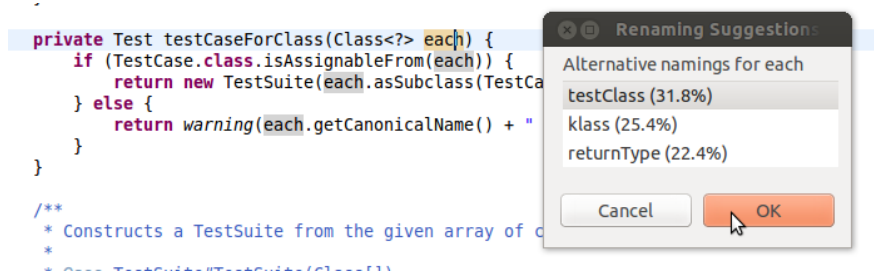
4.1.1 Use Cases and Tools

Coding conventions are an important aspect of software development (Chapter 3) and teams of developers strive to maintain a single, consistent coding style throughout each codebase. However, as the teams grow larger or new developers join, maintaining a single style becomes harder. In some cases, formal rules can be written, although this is not always possible. Our use cases aim to support new and existing members of software development teams to maintain a consistent style of a codebase.

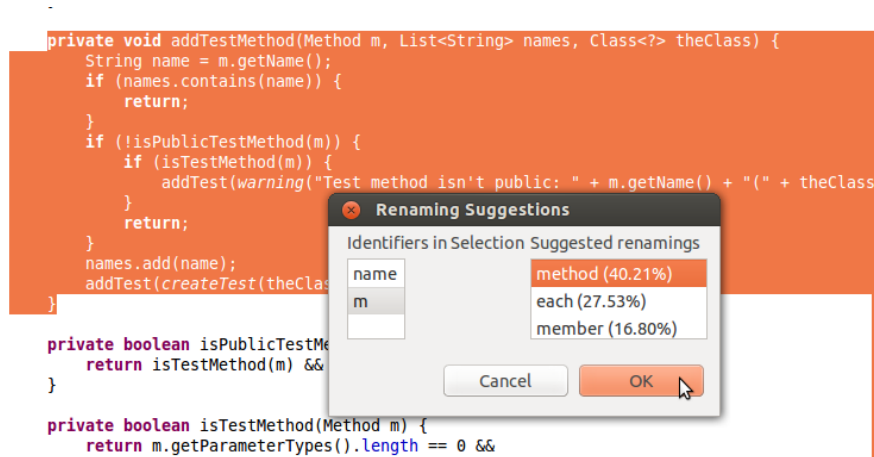
This leads us to target three use cases: 1) a developer preparing an individual commit or branch for review or promotion; 2) a release engineer while preparing a new



(a) The user right-clicks on the selected variable in the IDE and asks devstyle for suggestions.



(b) Once the user has requested NATURALIZE for suggestions a list of alternate names is shown. In the screenshot more conventional alternatives to the each parameter are presented.



(c) Users of devstyle can also select a range of code to get suggestions for all the variables within the selected code.

Figure 4.2: Screenshots of the devstyle Eclipse plugin. The plugin uses the NATURALIZE framework to suggest on-demand conventional alternative names to users. The plugin can be downloaded from the project website.

release trying to remove needless stylistic introduced by the new changes; and 3) a code reviewer wishing to consider how well a patch or branch obeys the project's norms.

Any code modification has a possibility of introducing bugs (Adams, 1984; Nagappan & Ball, 2007). This is certainly true of a system, like NATURALIZE, that is based on statistical inference, even when (as we always assume) all of NATURALIZE's suggestions are approved by a human. Because of this risk, the gain from making a change must be worth its cost. For this reason, our use cases focus on times when the code is already being changed. To support our use cases, we have built four tools:

devstyle A plugin for the Eclipse IDE that gives suggestions for identifier renaming both for a single identifier and for the identifiers in a selection of code.

styleprofile A code review assistant that produces a profile that summarizes the adherence of a code snippet to the coding conventions of a codebase and suggests renaming changes to make that snippet more stylistically consistent with a project.

stylish? A high precision *pre-commit script* for Git that rejects commits that have highly inconsistent and unnatural naming within a project.

Below we detail the use cases and the interactions in some more detail.

devstyle is an “intelligent” code assistant (akin to the “observer” role in pair programming). The IDE plugin offers two types of suggestions, single point suggestion under the mouse pointer and multiple point suggestion via right-clicking a selection. Screenshots from **devstyle** are shown in Figure 4.2. For single point suggestions, the user selects the variable where she requires a suggestion and **devstyle** displays a ranked list of alternatives to the selected name. If **devstyle** has no suggestions, it simply flashes the current name. If the user finds one of the suggestions useful, she selects it and **devstyle** renames the variable (α renaming). The user may also select a range (Figure 4.2c) and ask **devstyle** for suggestions for all the variables within the selected range (“multiple point suggestion”). This presents to the user suggestions for all the names of all the variables that appear at least once in the selected snippet. By default, for each variable, **devstyle** returns a list of the top k suggestions. By default, $k = 5$ based on usability considerations (Cowan, 2001; Miller, 1956). To accept a suggestion here, the user must first select a variable to modify, then select an option from its top alternatives.

styleprofile is a code review assistant that can help the author of a commit under review and the reviewers select more conventional variable names for the changed code.

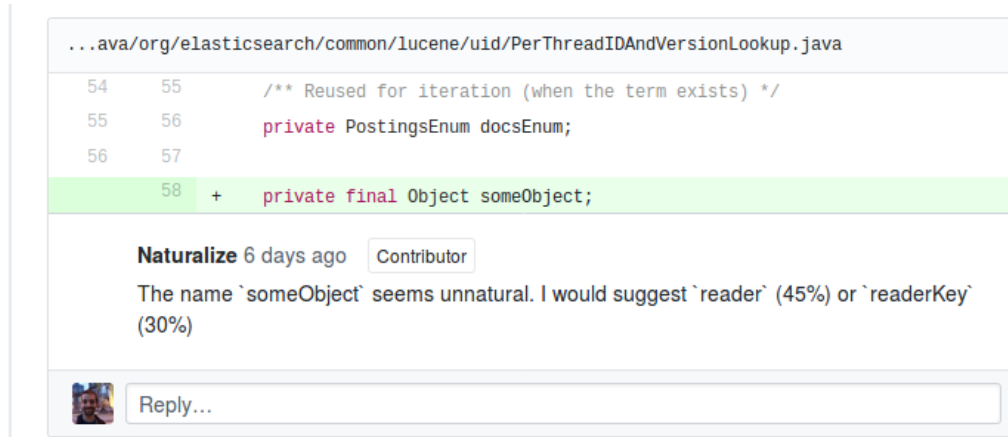


Figure 4.3: Mock-up of styleprofile within GitHub. The user has submitted a code review that contains the change shown in the diff. A styleprofile bot looks at the changed code and makes suggestions for alternatives only when the suggestion’s confidence is above a threshold.

The use case assumes that the author of the change has asked for a code review. For example, this may be a pull request in GitHub, a code review in Gerrit or another code review tool. A styleprofile bot is notified about the submission of a code review. It then retrieves the change and computes all suggestions *only* in the changed code. This makes sure that no irrelevant renaming suggestion is made. Whenever styleprofile is confident enough for a given variable, it automatically posts a comment suggesting alternative names for that variable. Figure 4.3 shows a mock-up of this use case within GitHub.

stylish? In some development environments, the suggestions of styleprofile could have been useful earlier, *i.e.* before the pull request was submitted. To achieve this, styleprofile can work as a pre-commit script, *i.e.* when the developer is about to commit some changes it draws the attention of the developer to variables within the changed code that could be renamed to a more conventional name. The developer can then decide whether to accept any of the suggestions or to ignore them.

NATURALIZE uses an existing codebase, called a training corpus, as a reference from which to learn conventions. Commonly, the training corpus will be the current codebase, so that NATURALIZE learns domain-specific conventions related to the current project. Alternatively, NATURALIZE comes with a pre-packaged suggestion model, trained on a corpus of popular, vibrant projects that presumably embody good coding conventions. Developers can use this engine if they wish to increase their codebase’s adherence to a larger community’s consensus on best practice. Projects that are just

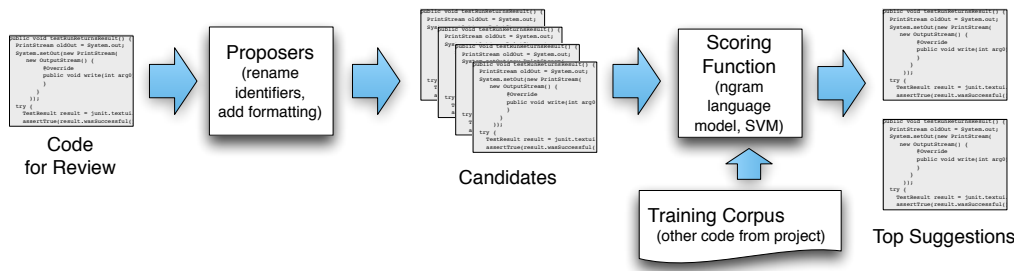


Figure 4.4: The architecture of NATURALIZE: a framework for learning coding conventions. A contiguous snippet of code is selected for review through the user interface. A set of *proposers* returns a set of candidates, which are modified versions of the snippet, e.g. with one local variable renamed. The candidates are ranked by a *scoring function*, such as an *n*-gram language model, which returns a small list of top suggestions to the interface, sorted by naturalness.

starting and have little or no code written can also use as the training corpus a pre-existing codebase, for example another project in the same organization, whose conventions the developers wish to adopt. Here, again, we avoid normative comparison of coding conventions, and do not force the user to specify their desired conventions explicitly. Instead, the user specifies a training corpus, and this is used as an *implicit* source of desired conventions. The NATURALIZE framework and tools are available at groups.inf.ed.ac.uk/naturalize.

4.2 The NATURALIZE Framework

In this section, we introduce the generic architecture of NATURALIZE, which can be applied to a wide variety of different types of conventions and is language independent. NATURALIZE is general and can be applied to any language for which a lexer and a parser exist, as token sequences and variable bindings (ASTs) are used during analysis. Figure 4.4 illustrates its architecture. The input is a code snippet to be naturalized. This snippet is selected based on the user input, in a way that depends on the particular tool in question. For example, in *devstyle*, if a user selects a local variable for renaming, the input snippet would contain all API nodes that reference that variable. The output of NATURALIZE is a short list of suggestions, which can be filtered, then presented to the programmer. In general, a suggestion is a set of snippets that may replace the input snippet. The list is ranked by a *naturalness score* that is defined below. Alternately, the system can return a binary value indicating whether the code is natural, so as to support

applications such as `stylish?`. The system makes no suggestion if it deems the input snippet to be sufficiently natural, or is unable to find good alternatives. This reduces the “Clippy effect” where users ignore a system that makes too many bad suggestions. In the next section, we describe each element in the architecture in more detail.

4.2.1 The Core of **NATURALIZE**

The architecture contains two main elements: proposers and the scoring function. The *proposers* modify the input code snippet to produce a list of *suggestion candidates* that can replace the input snippet. In the example from Figure 4.1, each candidate replaces all occurrences of `res` with a different name used in similar contexts elsewhere in the project, such as `results` or `queryResults`. In principle, many implausible suggestions could ensue, so, in practice, proposers may contain filtering logic to speed up the suggestion process.

A *scoring function* sorts these candidates according to a measure of naturalness. Its input is a candidate snippet, and it returns a real number measuring naturalness. Naturalness is measured with respect to a training corpus that is provided to **NATURALIZE** — thus allowing us to follow our guiding principle that naturalness must be measured with respect to a particular codebase. For example, the training corpus might be the set of source files A from the current application. A powerful way to measure the naturalness of a snippet is provided by probabilistic model of source code. We use $P_A(y)$ to indicate the probability that the model P , which has been trained on the corpus A , assigns to the code snippet y . The key intuition is that P_A is trained so that it assigns high probability to code in the training corpus, *i.e.*, snippets with higher probability are more like the training corpus, and presumably more natural. There are several key reasons why probabilistic models of source code are a powerful approach for modeling coding conventions. First, probability distributions provide an easy way to represent *soft* constraints about conventions. This allows us to avoid many of the pitfalls of inflexible, rule-based approaches. Second, because they are based on a learning approach, these models can flexibly adapt to and generalize from the conventions in an existing project. Intuitively, because P_A assigns high probability to variables that occur in the training corpus, it also assigns high probability to variables that are *similar* to those in the corpus. So the scoring function s tends to favor snippets that are stylistically consistent with the training corpus.

We score the naturalness of a snippet y with the probabilistic model P_A such that

$s(y, P_A) = \log P_A(y)$ where $s(x, P_A) > s(y, P_A)$ implies x is more “natural” than y . Where it creates no confusion, we write $s(y)$, eliding the second argument. When choosing between competing candidate snippets y and z , we need to know not only which candidate the model prefers, but how “confident” it is. We measure this by a *gap function* g , which is the difference in scores

$$g(y, z, P) = s(y, P) - s(z, P). \quad (4.1)$$

Because s is a log probability, g is the log ratio of probabilities between y and z . For example, when $g(y, z) > 0$ the snippet y is more natural — *i.e.*, less surprising according to the model — and thus is a better suggestion candidate than z . If $g(y, z) = 0$ then both snippets are equally natural.

Now we define the function $\text{suggest}(x, C, k, t)$ that returns the top candidates according to the scoring function. This function returns a list of top candidates, or the empty list if no candidates are sufficiently natural. The function takes four parameters: the input snippet x , the list $C = (c_1, c_2, \dots, c_r)$ of candidate snippets, and two thresholds: $k \in \mathbb{N}$, the maximum number of suggestions to return, and $t \in \mathbb{R}$, a minimum confidence value. The parameter k controls the size of the ranked list that is returned to the user, while t controls the *suggestion frequency*, that is, how confident NATURALIZE needs to be before it presents any suggestions to the user. Appropriately setting t allows NATURALIZE to avoid the Clippy effect by making no suggestion rather than a low quality one. Below, we present an automated method for selecting t .

The suggest function first sorts $C = (c_1, c_2, \dots, c_r)$, the candidate list, according to s , so $s(c_1) \geq s(c_2) \geq \dots \geq s(c_r)$. Then, it trims the list to avoid overburdening the user: it truncates C to include only the top k elements, so that $\text{length}(C) = \min\{k, r\}$. and removes candidates $c_i \in C$ that are not sufficiently more natural than the original snippet; formally, it removes all c_i from C where $g(c_i, x) < t$. Finally, if the original input snippet x is the highest ranked in C , *i.e.*, if $c_1 = x$, suggest ignores the other suggestions, sets $C = \emptyset$ to decline to make a suggestion, and returns C .

Binary Decision If an accept/reject decision on the input x is required, *e.g.*, as in `stylish?`, NATURALIZE must collectively consider all of the locations in x at which it could make suggestions. We propose a score function for this binary decision that measures how good is the best possible improvement that NATURALIZE is able to make. Formally, let L be the set of locations in x at which NATURALIZE is able to make suggestions, and for each $\ell \in L$, let C_ℓ be the system’s set of suggestions at ℓ . In general, C_ℓ contains name suggestions. Recall that P is the probabilistic model of source code.

We define the score

$$G(\mathbb{x}, P) = \max_{\ell \in L} \max_{c \in C_\ell} g(c, \mathbb{x}). \quad (4.2)$$

If $G(\mathbb{x}, P) > T$, then NATURALIZE rejects the snippet as being excessively unnatural. The threshold T controls the sensitivity of NATURALIZE to unnatural names and formatting. As T increases, fewer input snippets will be rejected, so some unnatural snippets will slip through, but as compensation the test is less likely to reject snippets that are in fact well-written.

Setting the Confidence Threshold The thresholds t in the suggest function and T in the binary decision function are on log probabilities of code, which can be difficult for users to interpret. Fortunately, these can be set *automatically* using the *false positive rate (FPR)*, i.e. the proportion of snippets \mathbb{x} that in fact follow convention but that the system erroneously rejects. We would like the FPR to be as small as possible, but, unless we wish the system to make no suggestions at all, we must accept some false positives. So we set a maximum acceptable FPR α , and search for a threshold t or T that ensures that NATURALIZE’s FPR is at most α . The principle is similar to statistical hypothesis testing. To make this work, we estimate the FPR for a given t or T . To do so, we select a random set of snippets from the training corpus, e.g., random method bodies, and compute the proportion of these snippets that are rejected using T . Again leveraging our assumption that our training corpus contains natural code, this proportion estimates the FPR. We use a grid search to find the greatest value of $T < \alpha$ ($t < \alpha$), the user-specified acceptable FPR bound. In principle, one could use this procedure to re-calibrate the threshold whenever the user changes the training corpus, although in practice we have found that a single threshold tends to yield similar FPR across our evaluation projects.

Suggesting Natural Names We now instantiate the core NATURALIZE framework for the task of suggesting natural identifier names. An instantiation of the framework for formatting conventions can be found in the published work of [Allamanis et al. \(2014\)](#). We start by describing the single suggestion setting. For concreteness, imagine a user of the `devstyle` plugin, who selects an identifier and asks `devstyle` for its top suggestions. It should be easy to see how this discussion can be generalized to the other use cases described in Subsection 4.1.1. Let v be the lexeme selected by the programmer. This lexeme could denote a variable, a method call, or a type, but in this chapter we are solely concerned with variables.

When a programmer binds a name to an identifier and then uses it, she implicitly

links together all the locations in which that name appears. Let L denote this set of locations, that is, the set of locations in the current scope in which the lexeme v is used. For example, if v denotes a local variable, then L_v would be the set of locations in which that local is used. Now, the input snippet is constructed by finding a snippet that subsumes all of the locations in L_v . Specifically, the input snippet is constructed by taking the lowest common ancestor in AST of the nodes in L_v . The proposers for this task retrieve a set of alternative names to v , which we denote A_v .

An interesting subtlety involves names that actually *should be* unique. Identifier names have a long tail, meaning that most names are individually uncommon. It would be undesirable to replace every rare name with common ones, as this would violate the sympathetic uniqueness principle. Fortunately, we can handle this issue in a subtle way: we convert rare names into the special UNK token. When we do this, UNK exists as a token in the LM, just like any other name. We simply allow NATURALIZE to return UNK as a suggestion, just like any other name. Returning UNK as a suggestion means that the model expects that it would be natural to use a rare name in the current context. The reason that this preserves rare identifiers is that the UNK token occurs in the training corpus specifically in unusual contexts where more common names were not used. Thus, if the input lexeme v occurs in an unusual context, this context is more likely to match that of UNK than of any of the more common tokens.

Multiple Point Suggestion It is easy to adapt the system above to the multiple point suggestion task. Recall (see Subsection 4.1.1) that this task is to consider the set of identifiers that occur in a region \mathbb{x} of code selected by the user, and highlight the lexemes that are least natural in context. For single point suggestion, the problem is to rank different alternatives, *e.g.*, different variable names, for the same code location, whereas for multiple point suggestion, the problem is to rank different code locations against each other according to how much they would benefit from improvement. In principle, a score function could be good at the single source problem but bad at the multiple source problem, *e.g.* if the score values have a different dynamic range when applied at different locations.

We adapt NATURALIZE slightly to address the multiple point setting. For all identifier names v that occur in \mathbb{x} , we first compute the candidate suggestions S_v as in the single suggestion case. Then the full candidate list for the multiple point suggestion is $S = \cup_{v \in \mathbb{x}} S_v$; each candidate arises from proposing a change to one name in \mathbb{x} . For the scoring function, we need to address the fact that some names occur more commonly in \mathbb{x} than others, and we do not want to penalize names solely because they occur more

often. So we normalize the score according to how many times a name occurs. Formally, a candidate $c \in S$ that has been generated by changing a name v in the locations L_v , we use the score function $s'(c) = |L_v|^{-1} s(c)$.

4.3 Choices of Scoring Function

The generic framework described in Section 4.2 can, in principle, employ a wide variety of machine learning or NLP methods for its scoring function. Indeed, a large portion of the statistical and deep learning-based NLP literature focuses on probability distributions over text, including language models, probabilistic grammars, topic models and distributed representations. We choose to use two probabilistic models. As a baseline model we use the n -gram language models, because previous work of [Hindle et al. \(2012\)](#) has shown that they are particularly able to capture the naturalness of code. We also design a novel neural probabilistic log-bilinear context model that takes advantage of the success of log-bilinear language models ([Mnih & Teh, 2012](#)) and is specifically tailored for source code.

4.3.1 Using the n -gram Language Model

One of the most effective practical LMs is the n -gram language model (see subsection 2.3.1.1). In this work, we use the n -gram LM with the stupid backoff smoothing ([Brants et al., 2007](#)). For our purposes and in our initial experiments stupid backoff performs as well as other more sophisticated smoothing methods and gives a speedup necessary for real-time training and use.

Implementation When we use an n -gram model, we can compute the gap function $g(y, z)$ very efficiently. This is because when g is used within suggest, ordinarily the code snippets y and z will be similar, *i.e.*, the input snippet and a candidate revision differ only on the suggested tokens that are renamed. The key insight is that in an n -gram model, the probability $P(y)$ of a snippet $y = (y_1 y_2 \dots y_N)$ depends only on the multiset of n -grams that occur in y , that is,

$$NG(y) = \{y_i y_{i+1} \dots y_{i+n-1} \mid 0 \leq i \leq N - (n - 1)\}. \quad (4.3)$$

An equivalent way to write a n -gram model is

$$P(y) = \prod_{a_1 a_2 \dots a_n \in NG(y)} P(a_n | a_1, a_2, \dots a_{n-1}). \quad (4.4)$$

Since the gap function is $g(y, z) = \log[P(y)/P(z)]$, any n -grams that are members both of $NG(y)$ and $NG(z)$ cancel, so to compute g , we only need to consider those n -grams not in $NG(y) \cap NG(z)$. Intuitively, this means that, to compute the gap function $g(y, z)$, we need to examine the n -grams containing the locations where the snippets y and z differ. This is a very useful optimization if y and z are long snippets that differ in only a few locations.

When training an LM, we take measures to deal with *rare* lexemes, since, by definition, we do not have much data about them. We use a preprocessing step — a common strategy in language modeling — that builds a vocabulary with all the identifiers that appear more than once in the training corpus. Let $\text{count}(v, b)$ return the number of appearances of token v in the codebase b . Then, if a token has $\text{count}(v, b) \leq 1$ we convert it to a special token, which we denote UNK. Then we train the n -gram model as usual. The effect is that the UNK token becomes a catchall that means the model expects to see a rare token, even though it cannot be sure which one.

Proposing Alternative Names One strategy for proposing alternative names is to suggest all possible tokens in the vocabulary of the n -gram LM. However, we found this approach to be relatively slow. To reduce the number of candidates, for every location $\ell \in L_v$ in the snippet x , we take a moving window of length $m \leq n$ around ℓ and copy all the m -grams w_i that contain that token. Call this set C_v the context set, *i.e.*, the set of m -grams w_i of x that contain the token v . Now we find all m -grams in the training set that are similar to an m -gram in C_v but that have some other lexemes substituted for v . Formally, we set A_v as the set of all lexemes v' for which $\alpha v \beta \in C_v$ and $\alpha v' \beta$ occurs in the training set. This guarantees that if we have seen a lexeme in at least one similar context, we place it in the alternatives list. Additionally, we add to A_v the special UNK token and remove all tokens that are not valid identifier names (*e.g.* reserved keywords, operators, *etc.*). Once we have constructed the set of alternative names, the candidates are a list S_v of snippets, one for each $v' \in A_v$, in which all occurrences of v in x are replaced with v' .

An n -gram model works well because, intuitively, it favors names that are common *in the context* of the input snippet. As we demonstrate in Section 4.4, this does *not* reduce to simply suggesting the most common names, such as `i` and `j`. For example, suppose that the system is asked to propose a name for `res` in line 3 of Figure 4.1. The n -gram model is highly unlikely to suggest `i`, because even though the name `i` is common it rarely appears in the contexts such as “QueryResults `i` ,”.

4.3.2 Log-bilinear Context Models of Code

As an alternative model we present a neural log-bilinear context model for code, inspired by neural probabilistic language models for natural language, which have seen many recent successes (Mnih & Hinton, 2007; Kiros et al., 2013; Mikolov et al., 2013a; Maddison & Tarlow, 2014). A particularly impressive success of these models has been that they assign words to continuous vectors that support analogical reasoning. For example, $\text{vector}(\text{'king'}) - \text{vector}(\text{'man'}) + \text{vector}(\text{'woman'})$ results in a vector close to $\text{vector}(\text{'queen'})$ (Mikolov et al., 2013a,b). Although many of the basic ideas have a long history (Bengio et al., 2003), this class of model is receiving increasing interest because of higher computational power from graphical processing units (GPUs) and because of more efficient learning algorithms such as noise contrastive estimation (Gutmann & Hyvärinen, 2012; Mnih & Teh, 2012).

Intuitively, our model assigns to every identifier name used in a project a continuous vector in a high dimensional space, in such a way that identifiers with similar vectors, or “embeddings”, tend to appear in similar contexts. Then, to name a variable, we select the name that is most similar in this embedding space to an embedding computed from its context. In this way, our model realizes Firth’s famous dictum, “You shall know a word by the company it keeps”. This slogan encapsulates the *distributional hypothesis* (Jurafsky, 2000), that semantically similar words tend to co-occur with the same other words. Two words are distributionally similar if they have similar distributions over surrounding words. For example, even if the words “hot” and “cold” never appear in the same sentence, they will be distributionally similar if they both often co-occur with words like “weather” and “tea”. The distributional hypothesis is a cornerstone of much work in computational linguistics, but we are unaware of previous work that explores whether this hypothesis holds in source code. Earlier work on the naturalness of code (Hindle et al., 2012) found that code tends to repeat constructs and exploited this repetition for prediction, but did not consider the *semantics* of tokens. In contrast, the distributional hypothesis states that *semantically similar* tokens are recognized because they tend to be distributionally similar.

Indeed, we qualitatively show in Section 4.5 that our context model produces embeddings that demonstrate implicit semantic knowledge about the similarity of identifiers. For instance, it successfully discovers matching components of names, which we call *subtokens*, like `min` and `max`, and `height` and `width`. We later use the same model in Chapter 5 and show that it distinguishes getters and setters, assigns function names

with similar functionality (like `grow` and `resize`) to similar locations.

Furthermore, to allow us to suggest neologisms, we introduce a new *subtoken context model* that exploits the internal structure of identifier names. In this model, we predict names by breaking them into parts, which we call subtokens, such as `get`, `create`, and `Height`, and then predicting names one subtoken at a time. The subtoken model automatically infers conventions about the internal structure of identifier names, such as “an exception variable ends with an `Exception`”, or “an boolean variable or method name starts with `is`”.

Log-bilinear models Here we briefly describe the necessary background for neural log-bilinear models that are widely used for neural LMs (Bengio et al., 2003). These models predict the next token y_m using a neural network that takes the previous tokens (*i.e.* the context of the current prediction) as input. This allows the network to flexibly learn which tokens provide much information about the immediately following token, and which tokens provide very little. Unlike an n -gram model, a neural LM makes it easy to add general long-distance features of the context into the prediction — we simply add them as additional inputs to the neural network. In our work, we focus on a simple type of neural LM that has been effective in practice, namely, the log-bilinear (LBL) LM (Mnih & Hinton, 2007). We start with a general treatment of log-linear models considering models of the form

$$P(t|\mathbf{c}) = \frac{\exp(s_\theta(t, \mathbf{c}))}{\sum_{t'} \exp(s_\theta(t', \mathbf{c}))} \quad (4.5)$$

where t is the target predicted quantity (the name of a variable for our purposes) and \mathbf{c} is the information (*i.e.* the “context”) that is used to predict t . Intuitively, s_θ is a function that indicates how much the model likes to see both t and \mathbf{c} together, the $\exp()$ maps this to be always positive, and the denominator ensures that the result is a probability distribution. This choice is very general. For example, if s_θ is a linear function of the features in \mathbf{c} , then the discriminative model is simply a logistic regression.

Log-bilinear models learn a map from every possible target t to a vector $\mathbf{q}_t \in \mathbb{R}^D$, and from each context \mathbf{c} to a vector $\hat{\mathbf{r}}_{\mathbf{c}} \in \mathbb{R}^D$. We interpret these as locations of each context and each target t in a D -dimensional space; these locations are called *embeddings* or *distributed vector representations*. The model predicts that the target t is more likely to appear in context \mathbf{c} if the embedding \mathbf{q}_t of the target is similar to that $\hat{\mathbf{r}}_{\mathbf{c}}$ of the context. To encode this in the model, we choose

$$s_\theta(t, \mathbf{c}) = \hat{\mathbf{r}}_{\mathbf{c}}^\top \mathbf{q}_t + b_t, \quad (4.6)$$

where b_t is a scalar bias which represents how commonly t occurs regardless of the context. To understand this equation intuitively, note that, if the vectors $\hat{\mathbf{r}}_{\mathbf{c}}$ and \mathbf{q}_t had norm 1, then their dot product is simply the cosine of the angle between them. So s_θ , and hence $P(t|\mathbf{c})$, is larger if either vector has a large norm, if b_t is large, or if $\hat{\mathbf{r}}_{\mathbf{c}}$ and \mathbf{q}_t have a small angle between them, that is, if they are more similar according to the cosine similarity metric.

To complete this description, we define the maps $t \mapsto \mathbf{q}_t$ and $c \mapsto \hat{\mathbf{r}}_{\mathbf{c}}$. For the targets t , the most common choice is to simply include the vector \mathbf{q}_t for every t as a parameter of the model. That is, the training procedure has the freedom to learn an arbitrary map between t and \mathbf{q}_t . For the contexts \mathbf{c} , this choice is not possible, as there are too many possible contexts. Instead, a common choice (Maddison & Tarlow, 2014; Mnih & Teh, 2012) is to represent the embedding $\hat{\mathbf{r}}_{\mathbf{c}}$ of a context as the sum of embeddings of the elements c_i within it, that is,

$$\hat{\mathbf{r}}_{\mathbf{c}} = \sum_{i=1}^{|\mathbf{c}|} C_i \mathbf{r}_{c_i}, \quad (4.7)$$

where $\mathbf{r}_{c_i} \in \mathbb{R}^D$ is a vector for each element in the context that is included in the model parameters. The variable i indexes every element in the context \mathbf{c} , so if the same element occurs multiple times in \mathbf{c} , then it appears multiple times in the sum. The matrix C_i is a diagonal matrix that serves as a scaling factor depending on the position of a element within the context. The values in C_i for each i are also included in the learned model parameters.

To summarize, log-bilinear models make the assumption that every target and every context can be mapped in a D -dimensional space. There are two kinds of embedding vectors: those directly learned (*i.e.* the parameters of the model) and those computed from the parameters of the model. To indicate this distinction, we place a hat on $\hat{\mathbf{r}}_{\mathbf{c}}$ to indicate that it is computed from the model parameters, whereas we write \mathbf{q}_t without a hat to indicate that it is a parameter vector that is learned directly by the training procedure. These models can also be viewed as a three-layer neural network, in which the input layer encodes all of the elements in \mathbf{c} using a 1-of- V encoding, the hidden layer outputs the vectors \mathbf{r}_{c_i} for each element in the context, and the output layer computes the score functions $s_\theta(t, \mathbf{c})$ and passes them to a softmax nonlinearity. For details on the neural network representation, see Bengio et al. (2003).

To learn these parameters, it has recently been shown (Mnih & Teh, 2012) that an alternative to the maximum likelihood method called noise contrastive estimation (NCE) (Gutmann & Hyvärinen, 2012) is effective. NCE allows us to train the model

while avoiding the need to explicitly compute the normalization constant, which is a computationally costly process, especially on large sizes of vocabulary. NCE measures how well the model $P(t|\mathbf{c})$ can distinguish the real data in the training set from “fantasy data” that is generated from a simple noise distribution. At a high level, this can be viewed as a black box alternative to maximum likelihood that measures how well the model fits the training data and in the limit of infinite fantasy data, the method converges to maximum likelihood. We optimize the model parameters using stochastic gradient descent.

Model Description Now we present a neural network, a novel log-bilinear (LBL) LM for code, which we call a *log-bilinear context model*. The key idea is that log-bilinear models make it especially easy to exploit long-distance information; *e.g.* when predicting the name of a variable, it is useful to take into account all of the name of the including method. We model long-distance context via a set of *feature functions*, such as “Whether the method where the variable is located contains the subtoken `add`”, “Whether the return type of the current method is `int`,” and so on. The log-bilinear context model combines these features with the local context.

Suppose that we are trying to predict a code token t given some context \mathbf{c} . The selection of \mathbf{c} is a design decision and it could be any ordered set of elements. Following, log-bilinear language models, we define \mathbf{c} as a sequence of context tokens $\mathbf{c} = (c_0, c_1, \dots, c_N)$. We assume that \mathbf{c} contains all of the other tokens in the file that are relevant for predicting t ; *e.g.* tokens from the body of the method where t is defined and used. A common design decision is that tokens in \mathbf{c} that are nearest to the target t are treated specially. Suppose that t occurs in position i of the file, that is, if the file is the token sequence t_1, t_2, \dots , then $t = t_i$. Then the *local context* is the set of tokens that occur within K positions of t , that is, the set $\{t_{i+k}\}$ for $-K \leq k \leq K, k \neq 0$. Therefore, the *local context* includes tokens that occur both before and after t .

The overall form of the context model will follow the generic form in Equation 4.5 and Equation 4.6, except that the context representation $\hat{\mathbf{r}}_{\mathbf{c}}$ is defined differently. In the context model, we define $\hat{\mathbf{r}}_{\mathbf{c}}$ using two different types of context: local and global. First, the local context is handled in a very similar way to the log-bilinear LM. Each possible lexeme v is assigned to a vector $\mathbf{r}_v \in \mathbb{R}^D$, and, for each token t_k that occurs within K tokens of t in the file, we add its representation \mathbf{r}_{t_k} into the context representation.

The global context is handled using a set of *features*. Each feature is a binary function based on the context tokens \mathbf{c} , such as the examples described at the beginning of this section. Formally, each feature f maps a \mathbf{c} value to either 0 or 1. [Maddison &](#)

Contexts:

```
final boolean isDone = false;
```

$$C_{-2}\mathbf{r}_{\text{final}} + C_{-1}\mathbf{r}_{\text{boolean}} + C_1\mathbf{r}_{=} + C_2\mathbf{r}_{\text{false}}$$

```
while ( ! isDone ) {
```

$$C_{-2}\mathbf{r}_{(} + C_{-1}\mathbf{r}_{!} + C_1\mathbf{r}_{)} + C_2\mathbf{r}_{\text{}} \}$$

Features:

```
boolean, in:MethodBody, final
```

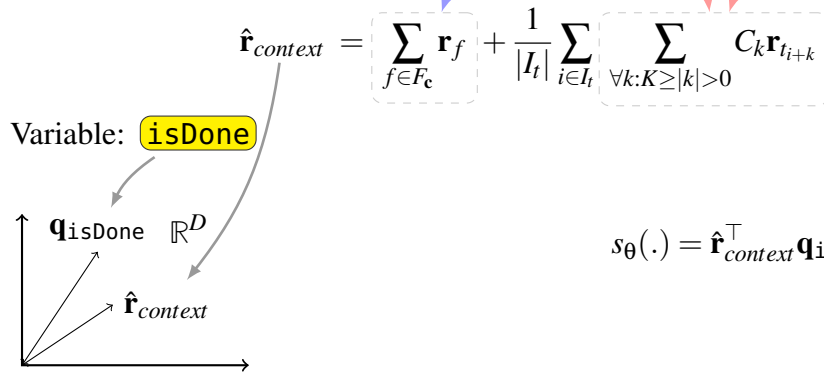
$$\mathbf{r}_{\text{boolean}} + \mathbf{r}_{\text{in:MethodBody}} + \mathbf{r}_{\text{final}}$$


Figure 4.5: Visual explanation of the representation and computation of context in the D -dimensional space as defined in Equation 4.8; Subsection 4.3.2 explains the sum over the I_t locations. Each source code token and feature maps to a learned D -dimensional vector in continuous space. The token-vectors are multiplied with the position-dependent context matrix C_i and summed, then added to the sum of all the feature-vectors. The resulting vector is the D -dimensional representation of the current source code identifier. Finally, the inner product of the context and the identifier vectors is added to a scalar bias b , producing a score for each identifier. This neural network is implemented by mapping its equations into code.

Tarlow (2014) use a similar idea to represent features of a syntactic context, that is, a node in an AST. Here, we extend this idea to incorporate arbitrary features of long-distance context tokens \mathbf{c} . The first column of Table 4.2 presents the full list of features that we use.

To learn an embedding, we assign each feature function to a single vector in the continuous space, in the same way as we did for tokens. Mathematically, let F be the set of all features in the model, and let $F_{\mathbf{c}}$, for a context \mathbf{c} , be the set of all features $f \in F$ with $f(\mathbf{c}) = 1$ (i.e. are active). For each feature $f \in F$, we learn an embedding $\mathbf{r}_f \in \mathbb{R}^D$, which is included as a parameter to the model in exactly the same way that \mathbf{r}_t was for the language modeling case.

Now, we can formally define a *context model of code* as a probability distribution $P(t|\mathbf{c})$ that follows the form (4.5) and (4.6), where $\hat{\mathbf{r}}_{\mathbf{c}} = \hat{\mathbf{r}}_{\text{context}}$, where $\hat{\mathbf{r}}_{\text{context}}$ is

$$\hat{\mathbf{r}}_{\text{context}} = \sum_{f \in F_{\mathbf{c}}} \mathbf{r}_f + \sum_{\forall k: K \geq |k| > 0} C_k \mathbf{r}_{t_{i+k}}, \quad (4.8)$$

where, as before, C_k is a position-dependent $D \times D$ diagonal context matrix that is also learned during training². Intuitively, this equation sums the embeddings of each token t_k that occurs near t in the file, and sums the embeddings of each feature function f that returns true (i.e., 1) for the context \mathbf{c} . Once we have this vector $\hat{\mathbf{r}}_{\text{context}}$, just as before, we can select a token t such that the probability $P(t|\mathbf{c})$ is high, which happens exactly when $\hat{\mathbf{r}}_{\text{context}}^\top \mathbf{q}_t$ is high — in other words, when the embedding \mathbf{q}_t of the proposed target t is close to the embedding $\hat{\mathbf{r}}_{\text{context}}$ of the context.

Figure 4.5 gives a visual explanation of the probabilistic model. This figure depicts how the model assigns probability to the variable `isDone` if the preceding two tokens are `final boolean` and the succeeding two are `= false`. Reading from top to bottom, the figure describes how the continuous embedding of the context is computed. Following the dashed (pink) arrows, the tokens in the local context are each assigned to D -dimensional vectors $\mathbf{r}_{\text{final}}$, $\mathbf{r}_{\text{boolean}}$, and so on, which are added together (after multiplication by the C_{-k} matrices that model the effect of distance), to obtain the effect of the local context on the embedding $\hat{\mathbf{r}}_{\text{context}}$. The solid (blue) arrows represent the global context, pointing from the names of the feature functions that return true to the continuous embeddings of those features. Adding the feature embeddings to the local context embeddings yields the final context embedding $\hat{\mathbf{r}}_{\text{context}}$. The similarity between this vector and embedding of the target vector $\mathbf{q}_{\text{isDone}}$ is computed using a dot product,

²Note that k can be positive or negative, so that in general $C_{-2} \neq C_2$.

which yields the value of $s_\theta(\text{isDone}, \mathbf{c})$ which is necessary for computing the probability $P(\text{isDone}|\mathbf{c})$ via Equation 4.5. We employ NCE for our log-bilinear model.

Multiple Target Tokens Up to now, we have presented the model in the case where we are renaming a target token t that occurs at only one location. In practical cases, when suggesting variable names, we need to take all of the occurrences of a name into account, as when we used the n -gram LM. When a token t appears at a set of locations L_t , we compute the context vectors $\hat{\mathbf{r}}_{context}$ separately for each token t_i , for $i \in L_t$, then average them, *i.e.*

$$\hat{\mathbf{r}}_{context} = \sum_{f \in F_c} \mathbf{r}_f + \frac{1}{|L_t|} \sum_{i \in L_t} \sum_{\forall k: K \geq |k| > 0} C_k \mathbf{r}_{t_i+k} \quad (4.9)$$

When we do this, we carefully rename all occurrences of t to a special token called SELF to remove t from its own context.

Proposing Alternative Names Similar to the n -gram LM we need a method for proposing alternative names. For the LBL model, we simply use the whole vocabulary (*i.e.* all variable names in the training set). This does *not* incur any extra costs since the model always computes the probability of all tokens in the last (softmax) step.

4.3.3 Subtoken Context Models of Code

A limitation of all of the previous models is that they are unable to predict *neologisms*, that is, unseen identifier names that have not been used in the training set. The reason for this is that we allow the map from a lexeme v to its embedding \mathbf{q}_v to be arbitrary (*i.e.* without learning a functional form for the relationship), so we have no basis to assign continuous vectors to identifier names that have not been observed. In this section, we sidestep this problem by exploiting the internal structure of identifier names, resulting in a new model which we call a *subtoken context model*.

The subtoken context model exploits the fact that identifier names are often formed by concatenating words in a phrase, such as `elementList` or `isDone`. We call each of the smaller words in an identifier a *subtoken*. We split identifier names into subtokens based on camel case and underscores, resulting in a set of subtokens that we use to compose new identifiers. To do this, we exploit the summation trick we used in $\hat{\mathbf{r}}_{context}$. Recall that we constructed this vector as a sum of embedding vectors for particular features in the context. Here, we define the embedding of a target vector to be the sum of the embeddings of its subtokens.

Let t be the token that we are trying to predict from a context \mathbf{c} . As in the context model, \mathbf{c} can contain tokens before and after t , and tokens from the global context. In the subtoken model, we additionally suppose that t is split up into a sequence of M subtokens, that is, $t = s_1 s_2 \dots s_M$, where s_M is always a special END subtoken that signifies the end of the subtoken sequence and s_1 is special START token. That is, the context model now needs to predict a sequence of subtokens in order to predict a full identifier. We begin by breaking up the prediction one subtoken at a time, using the chain rule of probability: $P(s_1 s_2 \dots s_M | \mathbf{c}) = \prod_{m=1}^M P(s_m | s_1 \dots s_{m-1}, \mathbf{c})$. Then, we model the probability $P(s_m | s_1 \dots s_{m-1}, \mathbf{c})$ of the next subtoken s_m given all of the previous ones and the context. Since preliminary experiments with an n -gram version of a subtoken model showed that n -grams did not yield good results, we employ a log-bilinear model

$$P(s_m | s_1 \dots s_{m-1}, \mathbf{c}) = \frac{\exp\{s_\theta(s_m, s_1 \dots s_{m-1}, \mathbf{c})\}}{\sum_{s'} \exp\{s_\theta(s', s_1 \dots s_{m-1}, \mathbf{c})\}}. \quad (4.10)$$

As before, $s_\theta(s_m, s_1 \dots s_{m-1}, \mathbf{c})$ can be interpreted as a score, which can be positive or negative and indicates how much the model “likes” to see the subtoken s_m , given the previous subtokens and the context. The exponential functions and the denominator are a mathematical device to convert the score into a probability distribution.

We choose a bilinear form for s_θ , with the difference being that in addition to tokens having embedding vectors, *subtokens* have embeddings as well. Mathematically, we define the score as

$$s_\theta(s_m, s_1 \dots s_{m-1}, \mathbf{c}) = \hat{\mathbf{r}}_{\text{SUBC}}^\top \mathbf{q}_{s_m} + b_{s_m}, \quad (4.11)$$

where $\mathbf{q}_{s_m} \in \mathbb{R}^D$ is an embedding for the subtoken s_m , and $\hat{\mathbf{r}}_{\text{SUBC}}$ is a continuous vector that represents the previous subtokens and the context. To define a continuous representation $\hat{\mathbf{r}}_{\text{SUBC}}$ of the context, we break this down further into a sum of other embedding features as

$$\hat{\mathbf{r}}_{\text{SUBC}} = \hat{\mathbf{r}}_{\text{context}} + \hat{\mathbf{r}}_{\text{SUBC-TOK}}. \quad (4.12)$$

In other words, the continuous representation of the context breaks down into a sum of two vectors: the first term $\hat{\mathbf{r}}_{\text{context}}$ represents the effect of the surrounding tokens — both local and global — and is defined exactly as in the context model via (4.8).

The new aspect is how we model the effect of the previous subtokens $s_1 \dots s_{m-1}$ in the second term $\hat{\mathbf{r}}_{\text{SUBC-TOK}}$. We handle this by assigning each subtoken s a second embedding vector $\mathbf{r}_s \in \mathbb{R}^D$ that represents its influence when used as a previous subtoken; we call this a *history embedding*. We weight these vectors by a diagonal matrix C_{-k}^{SUBC} , to

allow the model to learn that subtokens have decaying influence the farther that they are from the token that is being predicted. Putting this all together, we define

$$\hat{\mathbf{r}}_{\text{SUBC-TOK}} = \sum_{i=1}^M C_{-i}^{\text{SUBC}} \mathbf{r}_{s_{m-i}}. \quad (4.13)$$

This completes the definition of the subtoken context model. To sum up, the parameters of the subtoken context model are (a) the target embeddings \mathbf{q}_s for each subtoken s that occurs in the data, (b) the history embeddings \mathbf{r}_s for each subtoken s , (c) the diagonal weight matrices C_{-m}^{SUBC} for $m = 1, 2, \dots, M$ that represent the effect of distance on the subtoken history (we use $M = 3$, yielding a 4-gram-like model on subtokens) and the parameters that we carried over from the log-bilinear context model: (d) the local context embeddings \mathbf{r}_t for each token t that appears in the context, (e) the local context weight matrices C_{-k} and C_k for $-K \leq k \leq K, k \neq 0$, and (f) the feature embeddings \mathbf{r}_f for each feature $f \in F$ of the global context. We estimate all of these parameters from the training corpus.

Although this may seem a large number of parameters, this is typical for language models, *e.g.*, consider the V^5 parameters, if V is the number of lexemes, required by a 5-gram language model.

Generating Neologisms A final question is “Given the context \mathbf{c} , how do we find the lexeme t that maximizes $P(t|\mathbf{c})$?”. Previous models could answer this question simply by looping over all possible lexemes in the model, but this is impossible for a subtoken model, because there are infinitely many possible neologisms. So we employ beam search (see [Russell & Norvig \(1995\)](#) for details) to find the B tokens (*i.e.* subtoken sequences) with the highest probability.

4.3.4 Source Code Features for Context Models

Finally, we describe the features we use to capture global context. Identifying software measures and features that effectively capture semantic properties like comprehensibility or bug-proneness is a seminal software engineering problem that we do not tackle in this work. Here, we have selected measures and features heavily used in the literature and industry. The first column of Table 4.2 defines the features we used. In more detail, “Variable Type” tracks whether the type is generic, its type after erasure, and, if the type is an array, its size. “AST Ancestors” tracks the AST ancestors nodes of the location of the variable declaration, “Declaration Modifiers” includes any modifiers of the variable declaration (*e.g.* `final`, `volatile`, *etc.*). Finally, the method, class, superclass

and interface name subtokens include all the subtokens in the names of the containing method, containing class and the names of the implemented interfaces and inherited superclasses.

The features of a target variable name are its *global features*; we assign a \mathbf{r}_f vector to each of them; this vector is added in the left summation of Equation 4.8 if a feature’s indicator function f returns 1 for a particular variable (*i.e.* if $f \in F_c$). Although features are binary, we describe some — like the modifiers of a declaration, the node type of a AST, etc. — as categorical. All categorical features are converted into binary using a 1-of-K encoding.

4.4 Evaluation

Data We picked the top active Java GitHub projects on January 22nd 2015. We obtained the most popular projects by taking the sum of the z -scores of the number of watchers and forks of each project, using the GitHub Archive. Starting from the top project, we selected the top 20 projects excluding projects that were in a domain that was previously selected. We also included only projects with more than 50 collaborators and more than 500 commits. The projects along with short descriptions are shown in Table 4.1. We used this procedure to select a mature, active, and diverse corpus with large development teams. Finally, we split the files uniformly into a training (70%) and a test (30%) set.

Methodology We train all models on each of the training sets formed over the files of each project, retrieving one trained log-bilinear model per project. To evaluate the models, for each of the test files and for each variable (all identifiers that resolve to the same symbol), we compute the global features and local context of the location of the identifier and ask the model to predict the actual target token the developer used (which is unknown to the model). Due to the use cases we consider (Subsection 4.1.1), models that are deployed with a “confidence filter”, that is, the model will only present a suggestion to the user when the probability of the top ranked name is above some threshold. This is to avoid annoying the user with low-quality suggestions. To reflect this in our evaluation, we measure the degree to which the quality of the results changes as a function of the threshold. Rather than reporting the threshold, which is not directly interpretable, we instead report the *suggestion frequency*, which is the percentage of names in the test set for which the model decides to make a prediction for a given threshold.

Table 4.1: Java Evaluation Projects from GitHub. Ordered by popularity.

Name	Git SHA	Description
elasticsearch	d3e10f9	REST Search Engine
Android-Universal-Image-Loader	19ce389	Android Library
spring-framework	2bf6b41	Application Framework
libgdx	789aa0b	Game Development Framework
storm	bc54e8e	Distributed Computation
zxing	71d8395	Barcode image processing
netty	3b1f15e	Network App Framework
platform_frameworks_base	f19176f	Android Base Framework
bigbluebutton	02bc78c	Web Conferencing
junit	c730003	Testing Framework
rxjava	cf5ae70	Reactive JVM extensions
retrofit	afd00fd	REST client
clojure	f437b85	Programming Language
dropwizard	741a161	RESTful web server
okhttp	0a19746	HTTP+SPDY client
presto	6005478	Distributed SQL engine
metrics	4984fb6	Metrics Framework
spring-boot	b542aa3	App Framework Wrapper
bukkit	f210234	Minecraft Mod API
nokgiri	a93cde6	HTML/XML/CSS parser

To measure the quality of a suggestion, we compute the F1 score and the accuracy for the retrieval over the subtokens of each correct token. Thus, all methods are given partial credit if the predicted name is not an exact match but shares subtokens with the correct name. F1 is the harmonic mean of precision and recall and is a standard measure (Manning et al., 2008) because it is conservative: as a harmonic mean, its value is influenced most by the *lowest* of precision and recall. We also compute the accuracy of each prediction: a prediction is correct when the model predicts exactly (exact match) the actual token. When computing the F1 score for suggestion rank $k > 1$, we pick the precision, recall, and F1 of the rank $l \leq k$ that results in the highest F1 score.

Because this evaluation focuses on popular projects, the results may not reflect performance on a low quality project in which many names are poor. For such projects, we recommend training on a different project that has high quality names, but leave evaluating this approach to future work. Alternatively, one could argue that, because we measure whether the model can reproduce existing names, the evaluation is too harsh: if a predicted name does not match the existing name, it could be equally good, or even an improvement. Nonetheless, matching existing names in high quality projects, as we do, still provides evidence of overall suggestion quality, and, compared to a user study, an automatic evaluation has the great advantage that it allows efficient comparison of a larger number of different methods.

Finally, during training, we substitute rare identifiers, subtokens and features (*i.e.* those seen less than two times in the training data) with a special UNK token. During testing, when any of the models suggests the UNK token, we do not make any suggestions; that is, the UNK token indicates that the model expects a neologism that it cannot predict. For the subtoken model, during testing, we may produce suggestions that contain UNK subtokens, which are ignored in the evaluation. In the unlikely case that a context token $t_{i+k} = t_i$ (*i.e.* is the same token), we replace t_{i+k} with a special SELF token. This makes sure that the context of the model includes no information about the target token.

Training Parameters For the n -gram LM we use $n = 5$ and prune all tokens that appear only once in our dataset. We use stupid backoff (Brants et al., 2007) for smoothing. For the log-bilinear model, we used learning rate 0.07, $D = 50$, minibatch size 100, dropout 20% (Srivastava et al., 2014), generated 10 distractors for each sample for each epoch in NCE and trained for a maximum of 25 epochs picking the parameters that achieved the maximum likelihood in a held out validation set (the 10% of the training data). The context size was set to $K = 6$ and subtoken context size was

set to $M = 3$. Before the training started, parameters were initialized around 0 with uniform additive noise (scaled by 10^{-4}). The bias parameters b were initialized such that $P(t|c)$ matches the empirical (unigram) distribution of the tokens (or subtokens for the subtoken model). All the hyperparameters except for D were tuned using Bayesian optimization on bigbluebutton. The parameter D is special in that as we increase it, the performance of each model increases monotonically (assuming a good validation set and no overfitting), with diminishing returns. Also, an increase in D increases the computational complexity of training and testing each model. We picked $D = 50$ that resulted in a good trade-off of the computational complexity *vs.* performance.

4.4.1 Quantitative Evaluation

Single Point Variable Naming Figure 4.6 shows the performance of the baseline n -gram model along with the performance of the other neural models for variable names. For low frequency of suggestions (high confidence decisions), the neural models overperform the n -gram-based suggestions. This is expected since such models perform better than plain n -gram models in NLP (Mnih & Teh, 2012). Additionally, the features give a substantial performance increase over the models that lack features.

The subtoken model performs worse compared to the token-level LBL model for suggestion frequencies higher than 6%. This is to be expected, since the subtoken model has to make a sequence of increasingly uncertain decisions, predicting each subtoken sequentially, increasing the possibility of making a mistake at some point. For suggestion frequencies lower than 6% the performance of the subtoken model is slightly better compared to the token-level model, thanks to its ability to generate novel identifiers.

It should be noted that Figure 4.6 presents the results when using the suggestion filtering that includes UNK suggestions. Essentially, the models are able to predict UNK in rare contexts when they have sufficient indication that a rare or surprising token should be used. This explains why the curves stop after some suggestion frequency where no more non-UNK suggestions can be made. In Figure 4.7, we plot the same statistic ignoring UNK suggestions and forcing the models to *always* make a suggestion. The results now differ, showing that the subtoken model performs better for suggestions frequencies that are less than 20%, while the n -gram LM performs reasonably well. The difference between Figure 4.6 and Figure 4.7 suggests that the LBL models are more “conservative”, assigning higher probabilities to UNK and making fewer suggestions.

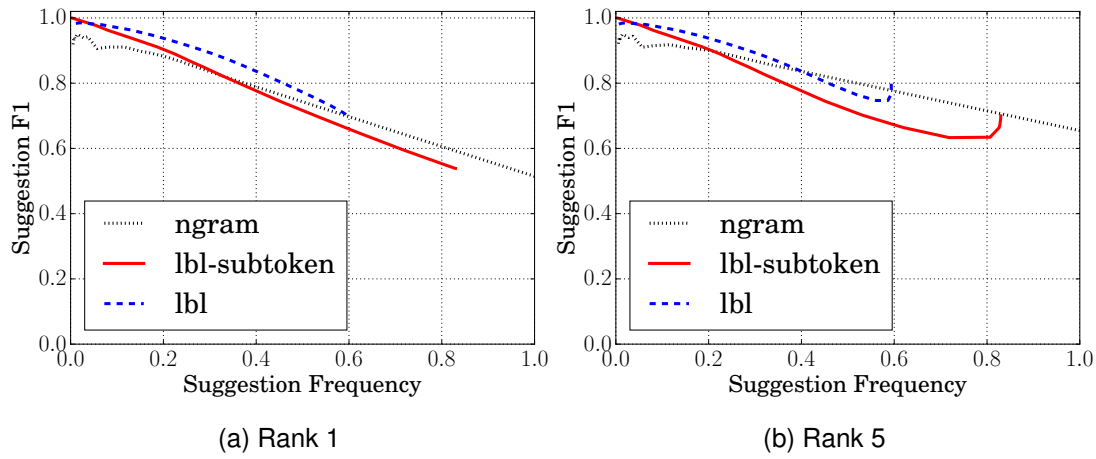


Figure 4.6: Single point suggestion for evaluation projects. Here we show the performance of the methods, when using the suggestion filtering method. The lines stop when there are no more confident suggestions to be made.

This will be also illustrated from the *stylish?* evaluation later in this section. One may note that the LBL models at high suggestion frequencies improve their performance. This suggests that these models tend to underestimate the confidence of some suggestions, although they are correct. This could have to do with variable names that appear in rarer contexts. In the future, better estimation of the confidence and its variance is needed. In addition, when we ignore UNK suggestions (Figure 4.7) the subtoken LBL model achieves better performance because it is able to get good partial credit for correct subtokens. This suggests that the subtoken model has learned successfully subtoken conventions of variable names (*e.g.* that a member field name may by-convention start with the `m_` prefix).

A natural question is how the features improve upon the performance of the LBL model. In Table 4.2 we show the absolute performance increase over a model that contains no features. We computed Table 4.2 over only three classes because of the cost of retraining the model one feature at a time. Looking at Table 4.2 one may see how each feature affects the performance of the models over the baseline neural model with no features at rank $k = 5$. First, we observe that the features help mostly at high suggestion frequencies. This is due to the fact that for high-confidence (low suggestion frequency) decisions the models are already good at predicting those names. Additionally, combining all the features yields a performance increase, suggesting that for variable names, only the combination of the features gives sufficiently better information about variable naming.

Table 4.2: Absolute increase in performance for each type of feature compared to the normal and subtoken models with no features at 5% suggestion frequency and at 20% suggestion frequency for rank $k = 5$. Averages from clojure, elasticsearch and libgdx, chosen uniformly at random from all projects in our corpus.

Feature	Absolute F1 Increase (%)				Absolute Accuracy Increase (%)			
	Simple		Subtoken		Simple		Subtoken	
	@5%	@20%	@5%	@20%	@5%	@20%	@5%	@20%
AST Ancestors One feature per parent and grand-parent AST nodes of declaration	-0.3	-1.0	0.9	2.6	2.0	0.7	0.8	2.5
Method, Class, Superclass and Interface Subtokens One feature for common identifiers subtokens	-1.2	0.1	0.0	1.0	1.1	1.8	0.1	0.8
Declaration Modifiers one feature per modifier	-0.1	-0.1	0.7	1.9	2.2	0.7	0.2	1.4
Variable Type one feature for each type and features denoting if the type is generic, array, etc.	-0.2	-0.3	0.6	3.8	2.1	1.5	0.3	3.3
<i>All</i>	1.2	4.8	-0.8	5.8	2.1	5.0	-0.9	5.7

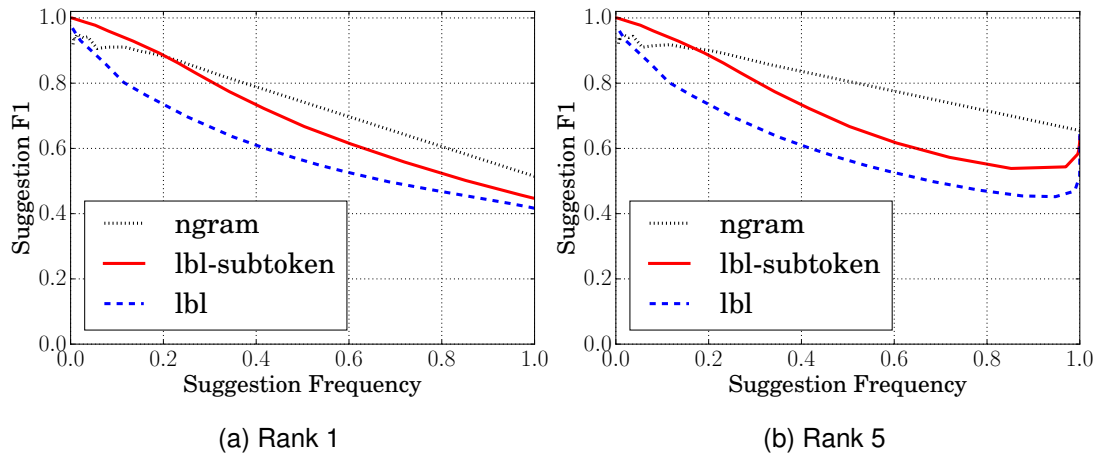


Figure 4.7: Single point suggestion for evaluation projects. Here we show the performance of the methods, *without* using the suggestion filtering method. This forces the models to make suggestions even when UNK is of higher probability.

Rejection Script Evaluation Now we evaluate the ability of *stylish?* to discriminate between code selections that follow conventions well from those that do not, by mimicking commits that contain unconventional variable names. Uniformly at random, we selected all methods from each project that contain at least one variable identifier, then uniformly (50%), we either made no changes or perturbed one identifier to UNK. This method for mimicking commits is a worst case scenario for *stylish?*. The models view the UNK just as an unseen name, rather as a name that is already known for the “preferences” of its context. For example, perturbing a variable named `color` to `i` is significantly more unconventional (and thus easy to detect by *stylish?*) compared to the generic UNK name.

We run *stylish?* and record whether the perturbed snippet is rejected because of its names. *stylish?* is unaware of the perturbation (if any) made to the snippet. Figure 4.8 reports NATURALIZE’s rejection performance as ROC curves. In each curve, each point corresponds to a different choice of threshold T , and the x -axis shows false positive rate (FPR), and the y -axis shows true positive rate (TPR), the proportion of the perturbed snippets that we correctly rejected. NATURALIZE achieves high precision for low recall, making it suitable for use as a filtering pre-commit script. When the false positive rate (FPR) is at most 0.05, we are able to correctly reject 17% of the snippets, using our LBL model. However, the performance is lower for higher FPR suggesting that future work is needed to improve performance.

Junk Names A junk name is a semantically uninformative name used in disparate

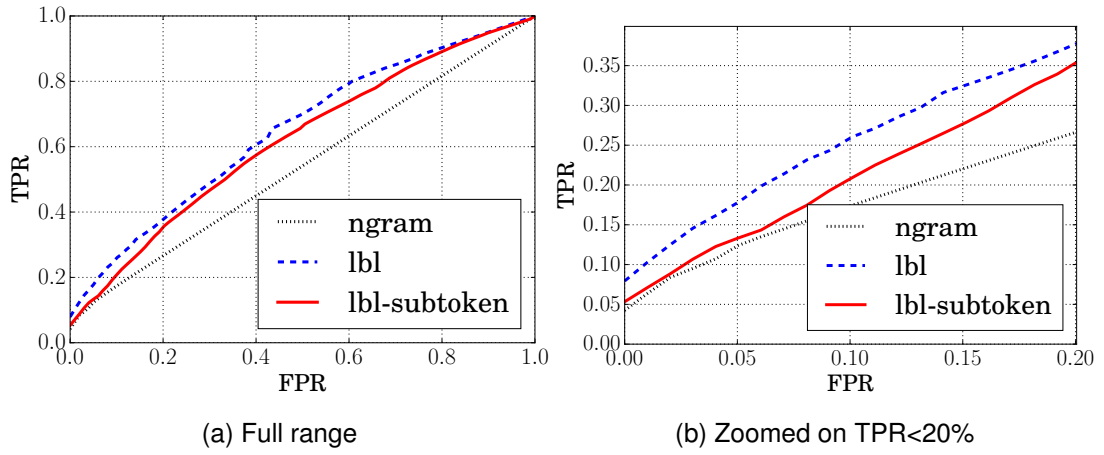


Figure 4.8: Receiver Operating Curve (ROC) for rejection script. Given a random code snippet, with probability 50% we perturbing a variable to UNK or leave everything unchanged. We then ask *stylish?* to either reject or accept the given snippet. The graph shows for a given FPR that a developer can tolerate the percent of the of perturbed snippets that *stylish?* correctly reject.

contexts. It is difficult to formalize this concept: for instance, in almost all cases, *foo* and *bar* are junk names, while *i* and *j*, when used as loop counters, are semantically informative and therefore not junk. Despite this, most developers “know it when they see it.”

One might at first be concerned that NATURALIZE would often suggest junk names, because junk names appear in many different n -grams in the training set. We argue, however, that in fact the opposite is the case: NATURALIZE actually *resists* suggesting junk names. This is because if a name appears in too many contexts, it will be impossible to predict a unsurprising follow-up, and so code containing junk names will have lower probability, and therefore worse score.

To evaluate this claim, we randomly rename variables to junk names in each project to simulate a low quality project. Notice that we are simulating a low quality *training* set, which should be the worst case for NATURALIZE. We measure how our suggestions are affected by the proportion of junk names in the training set. To generate junk variables we use a discrete Zipf’s law with slope $s = 1.08$, the slope empirically measured for all identifiers in our evaluation corpus. We verified the Zipfian assumption in previous work (Allamanis & Sutton, 2013b). Figure 4.9 shows the effect on our suggestions as the evaluation projects are gradually infected with more junk names. The framework successfully avoids suggesting junk names, proposing them at a lower frequency than they exist in the perturbed codebase.

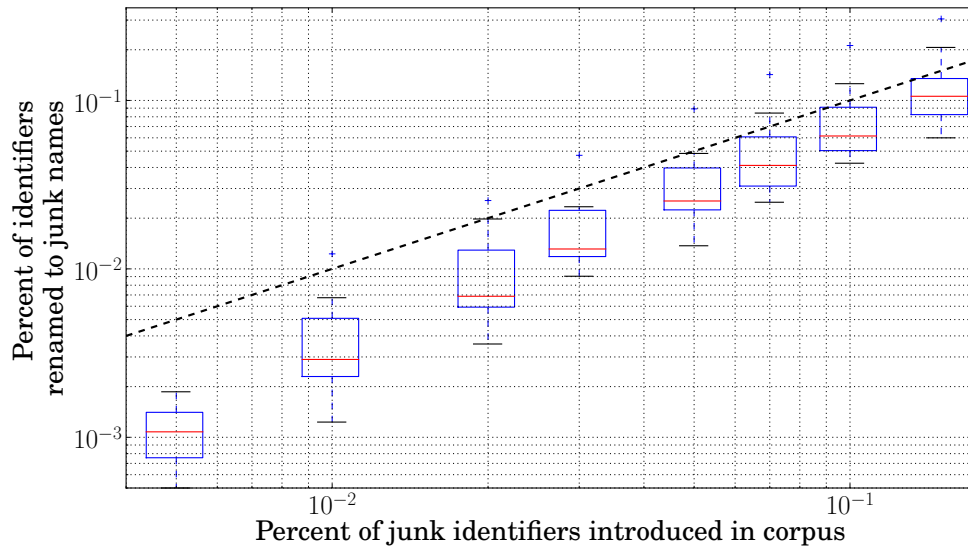


Figure 4.9: Is NATURALIZE robust to low-quality corpora? The x -axis shows percentage of identifiers perturbed to junk names to simulate low quality corpus. The y -axis is percentage of resulting low quality suggestions. Note log-log scale. The dotted line shows $y = x$. The boxplots are across the 10 evaluation projects.

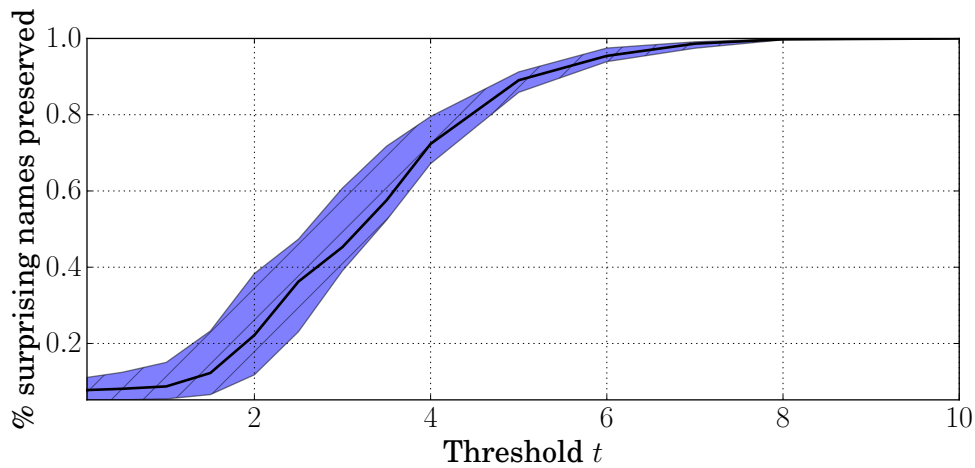


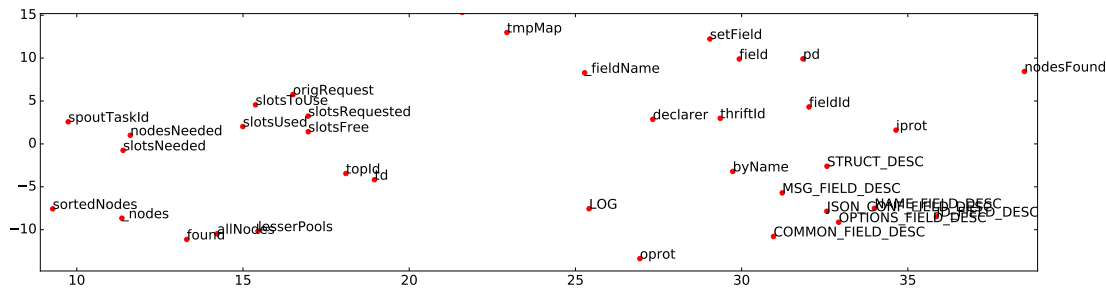
Figure 4.10: NATURALIZE does not cause the “heat death” of a codebase: we evaluate the percent of single suggestions made on UNK identifiers that preserve the surprising name. The threshold t on the x -axis controls the suggestion frequency of suggest; lower t gives suggest less freedom to decline to make low-quality suggestions.

4.4.2 Suggestions Accepted by Projects

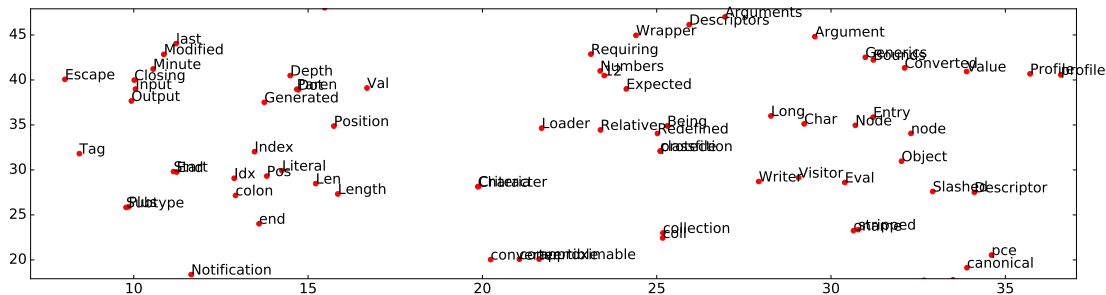
Using NATURALIZE’s `styleprofile`, we identified high confidence renamings and submitted 18 of them as patches to the 5 evaluation projects that actively use GitHub. Four projects merged our pull requests (14 of 15 commits); the last ignored them without comment. Developers in the projects that accepted NATURALIZE’s patches found the NATURALIZE useful: one said “Wow, that’s a pretty cool tool!”. JUNIT did not accept two of the suggested renamings as-is. Instead, the patches sparked a discussion. Its developers concluded that another name was more meaningful in one case and that the suggested renaming of another violated the project’s explicit naming convention: “Renaming `e` to `t` is no improvement, because we should consistently use `e`.”. We then pointed them to the code locations that supported NATURALIZE’s suggestion. This triggered them to change all the names that had caused the suggestion in the first place — NATURALIZE pointed out an inconsistency, previously unnoticed, that improved the naming in the project. Our project webpage has links to these discussions and they are also included in Appendix B. This suggests that NATURALIZE can be useful to developers and provide good suggestions that improve the stylistic consistency. A large-scale user study is future work.

4.5 Learned Representations

In this section, we evaluate the log-bilinear model qualitatively, by visualizing the learned embeddings. All of the LBL models that we have described assign tokens, features, and subtokens to *embeddings*, which are locations in a D -dimensional continuous space. These locations have been selected by the training procedure to explain statistical properties of tokens, but it does not necessarily follow that the embeddings capture anything about the semantics of names. To explore this question, we examine qualitatively whether names that appear semantically similar to us are assigned to similar embeddings, by visualizing the continuous embeddings assigned to names from a few projects. This raises the immediate difficulty of how to visualize vectors in a $D = 50$ dimensional space. Fortunately, there is a rich literature in statistics and machine learning about *dimensionality reduction* methods that map high dimensional vectors to two-dimensional vectors while preserving important properties of the original space. There are various ideas behind such techniques, such as preserving distances or angles between nearby points, or minimizing the distance between each point and its image in the 2D space.



(a) Variable embeddings visualization for the strom project. The plot shows that similar variable names are embedded nearby in the space. For example, the field descriptors ending in `_DESC` are grouped together and the `slot` related collections are also close together.



(b) Variable subtoken embeddings visualization for the spring-framework. The log-bilinear model learns embeddings for each subtoken, relating implicitly similar tokens. For example, the tokens `profile` and `Profile` are close together, although the model has no knowledge of their textual similarity. Similarly, `Len` and `Length`, `coll` and `collection` are also grouped together. The model also groups semantically similar words *e.g.* `Input` and `Output` together.

Figure 4.11: Variable tokens and subtokens embeddings visualization using t-SNE (van der Maaten & Hinton). t-SNE preserves distances but not angles. Note that different executions of t-SNE would return potentially different visualizations.

Classical techniques for dimensionality reduction include principal components analysis (PCA) and multidimensional scaling. We will also employ a more modern method called t-SNE (van der Maaten & Hinton).

Figure 4.11 shows two hand-picked visualizations of the 50-dimensional embeddings learned by the neural model. The 50 dimensional embeddings are projected onto the two-dimensional space using t-SNE. The graphs presented in Figure 4.11 are necessarily hand-picked examples from a much larger space. Additionally, since t-SNE has a stochastic behavior, the visualizations are also only one possible 2D projection of the 50-dimensional space. The models learn to place semantically similar (sub)tokens close together. This is true for names that are lexicographically close (*e.g.*

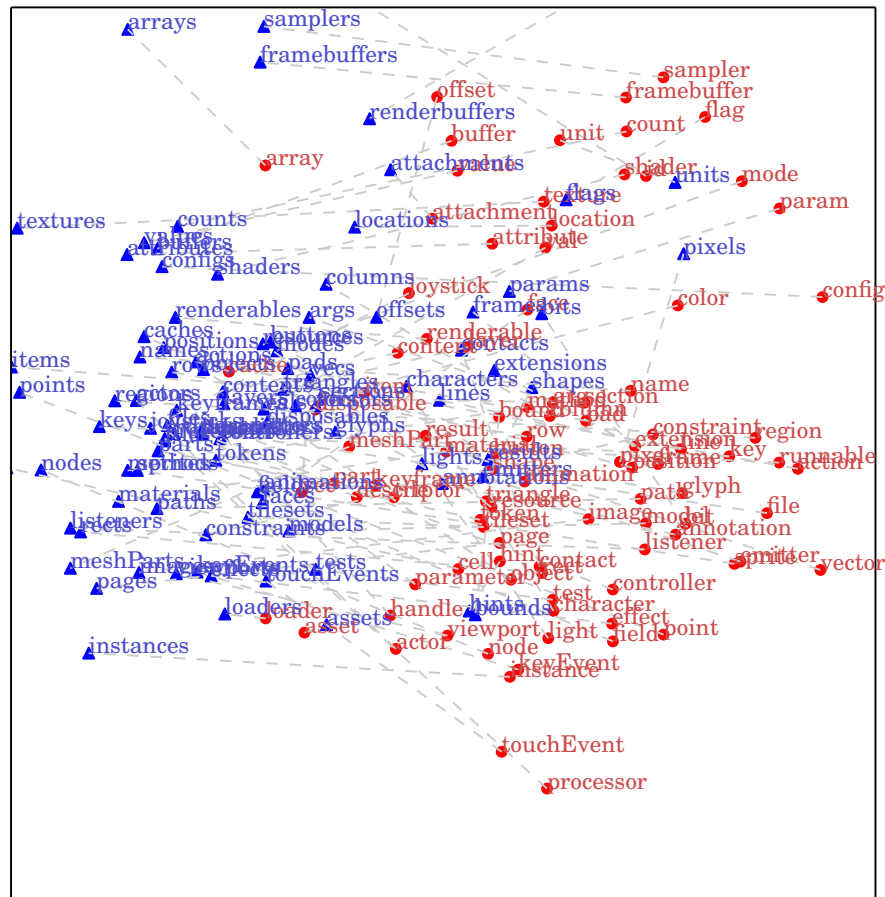


Figure 4.12: A 2D linear projection, using PCA, of the embeddings of singular and plural names in `libgdx`. Pairs are connected with a dotted line. The embeddings mostly separate singular and plural names. We expect most of the plural variables to refer to Collections of objects whose names appear in singular.

profile and Profile), but also for more semantically related names (*e.g.* nodesNeeded and slotsNeeded).

Additionally we examine the nearest neighbors of tokens in the D -dimensional space. This type of analysis avoids the risk, inherent in any dimensionality reduction method, that important information is lost in the projection from D dimensions to 2D. Table 4.3 shows some identifiers on a different project, clojure, for each identifier giving a list of other identifiers that are nearest in the continuous space. The nearest neighbors of a token t are those tokens v such that the inner product of the embeddings, that is, $\mathbf{q}_t^\top \mathbf{q}_v$, is maximized. We choose this measure because it most closely matches the notion of similarity in the model. Again, we are using the log-bilinear context model without subtoken information. We again see that the nearest neighbors in the continuous space seem to have similar semantic function such as the triple `fieldName, methodName,`

Table 4.3: Examples of nearest neighbors in the continuous space for variable names in clojure. Ordered by higher inner product $\mathbf{q}_{t_1}^\top \mathbf{q}_{t_2}$ where t_1 is in the first column and t_2 in the second.

Identifier	Nearest Neighbors (ordered by distance)
fieldName	className, methodName, target, method, methods
returnType	sb, typ, type, methodName, t
keyvals	items, seq, form, rest, valOrNode
params	paramType, ctor, methodName, args, arg

Table 4.4: Closely related (sub-)tokens for libgdx variables. The top 10 pairs that have the highest $\mathbf{q}_{t_1}^\top \mathbf{q}_{t_2}$ are shown. For the subtoken model some numeral pairs (e.g. 9–8) are omitted.

LBL Model	LBL–Subtoken Model
camera – cam	6 – 5
padBottom – padLeft	Height – Width
dataOut – dataIn	swig – class
localAnchorA – localAnchorB	Min – Max
bodyA – bodyB	shape – collision
framebuffers – buffers	Left – Right
worldWidth – worldHeight	camera – cam
padRight – padLeft	TOUCH – KEY
jarg7 – jarg6_	end – start
spriteBatch – batch	loc – location

and `className` or the names `returnType`, `typ`, and `type`.

Table 4.4 takes this analysis a bit further. This table shows the “nearest nearest neighbors”: those pairs of tokens or subtokens that are closest in the embedding space out of all possible pairs of tokens. On the left column, we see pairs of close neighbors from the feature-based log-bilinear context model without subtokens. These have many similar pairs, such as `width` and `height`. It is striking how many of these pairs contain similar subtokens *even though this model does not contain subtokens*. Moving to the subtoken model, the right column of Table 4.4 shows pairs of subtokens that are closest in the embedding space. The model learns that pairs like numerals, `Min` and `Max`, and `Height` and `Width` should be placed near to each other in the continuous space. This is further evidence that the model is learning semantic similarities given only statistical relationships between tokens.

We can also attempt to be a bit more specific in our analysis. In this we are inspired by Mikolov et al. (2013a), who noticed that adding together two of their embeddings of natural language words often yielded a compositional semantics — *e.g.* embedding(“*Paris*”) - embedding(“*France*”) + embedding(“*Vietnam*”) yielded a vector whose nearest neighbor was the embedding of “*Hanoi*”. To attempt something similar for source code, we consider semantic relationships that *pairs* of identifiers have with each other.

For Figure 4.12, we project the D -dimensional embeddings to 2D using PCA rather than t-SNE. Unlike t-SNE, PCA is a linear method, that is, the mapping between the D -dimensional points and the 2D points is linear. Therefore, if groups of points are separated by a plane in the 2D space, then we know that they are separated by a plane in the higher-dimensional space as well. In Figure 4.12, we match pairs of variable names in the libgdx project in which one name (the “plural name”) equals another name (the “singular name”) plus the character `s`. The Java convention that `Collection` objects are often named by plural variable names motivates this choice. Although this mapping is more noisy than the last, we still see that plural names tend to appear on the left side of the figure, and singular names on the right. From this exploration we conclude that the continuous locations of each name seem to be capturing semantic regularities. Readers who wish to explore further can view the embeddings at <http://groups.inf.ed.ac.uk/cup/naturalize/>.

Even though the continuous embeddings are learned from context alone, these visualizations suggest that these embeddings also contain, to some extent, *semantic* information about which identifiers are similar. This suggests that local and global context do

provide information that can be represented and exploited, that is, semantically similar names are used in similar contexts. It is especially striking that we have consistently found that nearby tokens in the continuous space tend to share subtokens, even when the model does not include subtoken information. The right column of Table 4.4 reinforces this point since it shows that, when we do use the subtoken model, nearby pairs of subtokens in the continuous space seem to be meaningfully related.

Finally, it can be objected that this type of analysis is necessarily subjective. When backed and validated by quantitative analysis of Section 4.4, however, this analysis provides visual insight, gained from looking at the embedding vectors.

4.6 Conclusions

In this section, we presented NATURALIZE, a framework that suggests conventional variable names to reduce unnecessary stylistic diversity within source code projects. We presented three different machine learning models that score and alternative names and evaluated them on popular, high-quality and real-life projects. Our subtoken log-bilinear model can accurately suggest previously unseen names and capture subtoken-level coding conventions. In addition we found that the log-bilinear models produce qualitatively interesting representations of variable names.

Learning to name variables is an interesting problem with practical applications for software engineers. NATURALIZE's performance suggests that variable naming assistance tools can be deployed in practical scenarios and are able to make accurate suggestions maintaining a small false positives ratio. However, the variable naming problem is far from being solved. The performance at large suggestion frequencies (see Figure 4.6) suggests that future work can improve upon the performance of the presented models. In addition, continuous embeddings of identifiers have many other potential applications in software engineering, such as exploration of linguistic anti-patterns ([Arnaoudova et al., 2013](#)), code search and feature location ([Rubin & Chechik, 2013](#)).

Chapter 5

Learning Method Naming Conventions

“So, beautiful code is lucid, it is easy to read and understand; its organization, its shape, its architecture reveals intent as much as its declarative syntax does.”

– Vikram Chandra, *The Beauty of Code*

Language starts with names. While programming, developers must name variables, parameters, functions, classes, and files. They strive to choose names that are meaningful and conventional, *i.e.* consistent with other names used in related contexts in their codebase. Indeed, leading industrial experts, including [Beck \(2007\)](#), [McConnell \(2004\)](#), and [Martin \(2008\)](#), have stressed the importance of identifier naming in software. Finding good names for programming language constructs is difficult; poor names make code harder to understand and maintain ([Lawrie et al., 2006a](#); [Takang et al., 1996](#); [Liblit et al., 2006](#); [Arnaoudova et al., 2015](#)). Empirical evidence suggests that poor names also lead to software defects ([Butler et al., 2009](#); [Abebe et al., 2012](#)). Code maintenance exacerbates the difficulty of finding good names, because the appropriateness of a name changes over time: an excellent choice, at the time a construct is introduced, can degrade into a poor name, as when a variable is used in new context or a function’s semantics changes.

Names of methods are particularly important, and can be difficult to choose. [Høst & Østvold \(2009\)](#) eloquently captured their importance:

“Methods are the smallest named units of aggregated behavior in most conventional programming languages and hence the cornerstone of abstraction.”

Semantically distinct method names are the basic tools for reasoning about program

behavior. Programmers directly think in terms of these names and their compositions, since a programmer chose them for the units into which the programmer decomposed a problem. Moreover, method names can be hard to change, especially when they are used in an API. When published in a popular library, method naming decisions of external APIs are especially rigid and poor names can make the library hard to use.

In this chapter, we suggest that modern statistical tools allow us to automatically suggest descriptive, idiomatic method names to programmers. We tackle the *method naming* problem: the problem of inferring a method’s name from its body. As developers spend approximately half of their development time trying to understand and comprehend code during maintenance alone (Corbi, 1989), any progress toward solving the method naming problem will improve the comprehensibility of code (Takang et al., 1996) increasing programmer productivity (Hendrix et al., 2002).

In Chapter 4, we introduced the NATURALIZE framework, which learns the coding conventions used in a codebase and tackles one naming problem programmers face — that of naming variables — by exploiting the “naturalness” or predictability of code (Hindle et al., 2012). However, the method naming problem is much more difficult than the variable naming problem, because the appropriateness of method names depends not solely on their uses but also on their internal structure, *i.e.* their body. An adequate name must describe not just what the method *is*, but what it *does*. Variable names, by contrast, can often be predicted solely from a few tokens of local context; for example, it is easy to predict the variable name that follows the tokens `for (int`. Because method names must be functionally descriptive, they often have rich internal structure: method names are often verb phrases. But this means that method names are often *neologisms*, that is, names not seen in the training corpus. Existing probabilistic models of source code, including the *n*-gram models used in NATURALIZE, cannot suggest neologisms. These aspects of the method naming problem severely exacerbate data sparsity because this means we need to consider larger context which necessarily means that any individual context will be observed less often. In addition, it is rare to find the same functionality implemented multiple times, since if this were the case, this functionality would have been encapsulated in a library to be reused directly. Therefore, the method naming problem requires models that can better exploit the structure of code, taking into account long-range dependencies and modeling the method body precisely, while minimizing the effects of data sparsity.

Method Naming as Code Summarization This problem resembles a summarization task, where the method name is viewed as the summary of the code. However, method

naming is drastically different from natural language summarization, because unlike natural language, source code is unambiguous and highly structured. Furthermore, a good summary needs to explain *how* the code instructions compose into a higher-level meaning and not naïvely explain what the code does. This necessitates learning higher-level patterns in source code that uses both the structure of the code and the identifiers to detect and explain complex code constructs. Our method naming may also be viewed as a translation task, in the same way that any summarization problem can be viewed as translation. But a significant difference from translation is that the input source code sequence tends to be very large (72 on average in our data) and the output summary very small (3 on average in our data). The length of the input sequence necessitates the extraction of both temporally invariant attention features and topical sentence-wide features and — as we show in this chapter — existing neural machine translation techniques yield sub-optimal results.

Haiduc et al. (2010b) showed that natural language text summarization does *not* work well for code and such techniques must be adapted to be effective. They later developed summaries that are used to improve comprehension (Haiduc et al., 2010a). Sridhara et al. (2011) used idioms and structure in the code of methods to generate high level abstract summaries. While they don’t suggest method names, they discuss how their approach may be extended to provide them. Sridhara et al. (2010); Sridhara (2012) also showed how to generate code summaries appropriate for comments within the code (*e.g.* as method headers). For more work in this area, Eddy et al. (2013); Nazara et al. (2015) provide surveys of code summarization methods.

Machine Learning for Naming Methods Deep learning for structured prediction problems, in which a sequence (or more complex structure) of predictions need to be made given another input sequence, presents special difficulties, because not only are the input and output high-dimensional, but the dimensionality is not fixed in advance. Recent research has tackled these problems using neural models of attention (Mnih et al., 2014), which have had great recent successes in machine translation (Bahdanau et al., 2015) and image captioning (Xu et al., 2015). Attention models have been successful because they separate two concerns: predicting which input locations are most relevant to each part of the output as it is generated; and actually predicting an output location given the most relevant inputs.

In this chapter, we suggest that many domains — including method naming— contain translation-invariant features that can help to determine the most useful locations for attention. For example, in a research paper, the sequence of words “in this paper,

we suggest” often indicates that the next few words will be important to the topic of the paper. As another example, suppose a neural network is trying to predict the name of a method in the Java programming language from its body. If we know that this method name begins with `get` and the method body contains a statement `return _____ ;`, then whatever token fills in the blank is likely to be useful for predicting the rest of the method name. Previous architectures for neural attention are not constructed to learn translation-invariant features specifically.

In this chapter, we introduce a neural convolutional attention model, that includes a convolutional network within the attention mechanism itself. Convolutional models are a natural choice for learning translation-invariant features while using only a small number of parameters and for this reason have been highly successful in non-attentional models for images (LeCun et al., 1998; Krizhevsky et al., 2012) and text classification (Kalchbrenner et al., 2014). But to our knowledge they have not been applied within an attention mechanism. Convolutional networks can robustly learn to detect long and short-range patterns in a computationally efficient way. Our model uses a set of convolutional layers — without any pooling — to detect patterns in the input and identify “interesting” locations where attention should be focused.

Furthermore, source code presents the challenge of out-of-vocabulary words. Each new software project and each new source file introduces new vocabulary about aspects of the software’s domain, data structures, and so on. This vocabulary often does not appear in the training set. To address this problem, we introduce a *copy mechanism*, which uses the convolutional attention mechanism to identify important tokens in the input even if they are out-of-vocabulary tokens that do not appear in the training set. The decoder, using a meta-attention mechanism, may choose to copy tokens directly from the input to the output sequence, resembling the functionality of Vinyals et al. (2015).

Use Cases Our suggestion model can be embedded within a variety of tools to support code development and code review. Some of these scenarios are detailed in Subsection 4.1.1 but are also mentioned here for self-containedness. During development, suppose that the developer is adding a method to an existing project. After writing the body, the developer may be unsure if the name she chose is descriptive and conventional within the project. Our model suggests alternative names from patterns it learned from other methods in the project. During code review, our model can highlight those names to which our model assigns a (very) low score. In either case, the system has two phases: a training phase, which takes as input a training set of source files (*e.g.*

the current revision of the project) and returns a trained neural network model that can suggest names; and a testing or deployment phase, in which the input is a trained neural network and the source code of a method, and the output is a ranked list of suggested names. Any suggestion system has the potential to suffer from what we have called the “Clippy effect” (see [Allamanis et al. \(2014\)](#) and Chapter 4), in which too many low quality suggestions alienate the user. To prevent this, our suggestion model also returns a numeric score that reflects its degree of confidence in its suggestion; practical tools would only make a suggestion to the user if the confidence were sufficiently high.

Other use cases of our machine learning model can be found in the realm of code summarization and retrieval. Apart from the obvious use as a summarization technique that can generate “captions” of code and aid program comprehension, our tool can be useful for code search. By acting as a query expansion method the network can suggest new natural language terms that may be relevant for a given snippet of code.

Contributions The key contributions of this chapter are:

- a novel convolutional attention network architecture that combines two convolutional attention mechanisms and a meta-attention mechanism to successfully learn to predict method summaries in the form of descriptive method names at the subtoken level;
- a comprehensive approach to the method naming problem, with interest both in the machine learning and software engineering community;
- a comprehensive evaluation of four competing algorithms on real-world data that demonstrates the advantage of our method compared to standard attention mechanisms; and
- a comparison between our feature-less convolutional attention network and the log-bilinear model presented in Chapter 4 showing how the convolutional attention networks achieves better performance without any external hand-crafted features.

5.1 A Convolutional Attention Model

Our convolutional attention model receives as input a code snippet \mathbb{c} . This snippet may be the body of a method or any other snippet. We then tokenize \mathbb{c} and split each identifier

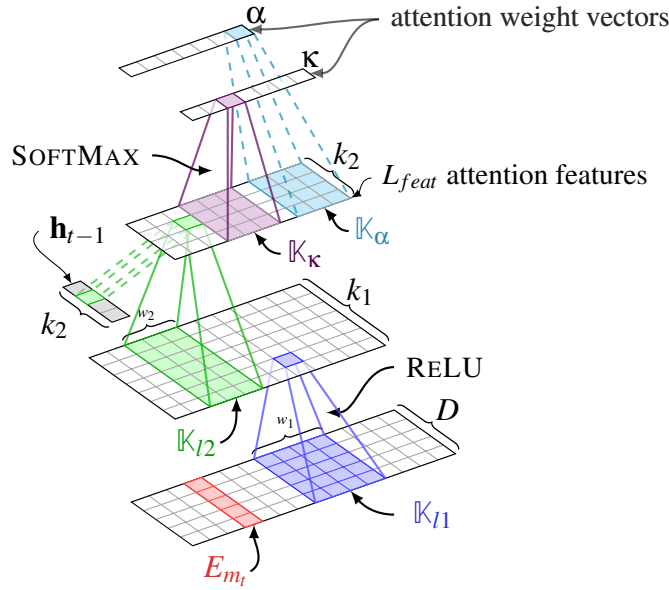


Figure 5.1: The architecture of the convolutional attention network. **attention_features** learns location-specific attention features given an input sequence $\{m_i\}$ and a context vector \mathbf{h}_{t-1} . Given these features **attention_weights**—using a convolutional layer and a SOFTMAX—computes the final attention weight vectors such as α and κ in this figure.

token into subtokens.¹ We then have an input $\mathbf{c} = [\langle c \rangle, c_1, \dots, c_N, \langle /c \rangle]$, where $\langle c \rangle$ and $\langle /c \rangle$ are two special start and end tokens. Given \mathbf{c} , our networks learns to output an extreme summary in the form of a concise method name. The summary is a sequence of subtokens $\mathbf{m} = [\langle m \rangle, m_1, \dots, m_M, \langle /m \rangle]$, where $\langle m \rangle$ and $\langle /m \rangle$ are the special start and end symbols of every subtoken sequence. For example, in the `shouldRender` method (top left of Table 5.3) the input code subtokens are

$$\mathbf{c} = [\langle c \rangle, \text{try}, \{, \text{return}, \text{render}, \text{requested}, |, |, \text{is}, \text{continuous}, \dots, \langle /c \rangle]$$

while the target output is

$$\mathbf{m} = [\langle m \rangle, \text{should}, \text{render}, \langle /m \rangle].$$

The neural network predicts each summary subtoken sequentially and it models the probability distribution $P(m_t | m_0, \dots, m_{t-1}, \mathbf{c})$ for all $t > 0$. Information about the previously produced subtokens m_0, \dots, m_{t-1} is passed into a recurrent neural network that

¹Subtokens — as in Chapter 4 — refer to the parts of a source code token *e.g.* `getInputStream` has the `get`, `Input` and `Stream` subtokens. We split subtokens on camelCase, snake_case and around numerals.

represents the input state with a vector \mathbf{h}_{t-1} . Our convolutional attention neural network (Figure 5.1) uses the input state \mathbf{h}_{t-1} and a series of convolutions over the embeddings $E_{\mathbf{m}}$ of the tokens \mathbf{c} to compute a matrix of attention features L_{feat} , that contains one vector of attention features for each sequence position (Figure 5.1). The resulting features are used to compute one or more normalized attention vectors (e.g. α in Figure 5.1) which are distributions over input token locations containing a weight (in $(0, 1)$) for each subtoken in \mathbf{c} . Finally, given the attention weights, a context representation is computed and is used to predict the probability distribution over the targets m_i . This model can be thought as a generative bimodal model of summary text given a code snippet.

5.1.1 Learning Attention Features

We describe our model from the bottom up (Figure 5.1). First we discuss how to compute the attention features L_{feat} from the input \mathbf{c} and the previous hidden state \mathbf{h}_{t-1} . The basic building block of our model is a convolutional network (LeCun et al., 1990; Collobert & Weston, 2008) for extracting position and context-dependent features. The input to **attention_features** is a sequence of code subtokens \mathbf{c} of length $\text{LEN}(\mathbf{c})$ and each location is mapped to a matrix of attention features L_{feat} , with size $(\text{LEN}(\mathbf{c}) + \text{const}) \times k_2$ where const is a fixed amount of padding. The intuition behind **attention_features** is that given the input \mathbf{c} , it uses convolution to compute k_2 features for each location. By then using \mathbf{h}_{t-1} as a multiplicative gating-like mechanism, only the currently relevant features are kept in L_2 . In the final stage, we normalize L_2 . **attention_features** is described with the following pseudocode:

attention_features (code tokens \mathbf{c} , context \mathbf{h}_{t-1})

```

 $C \leftarrow \text{LOOKUPANDPAD}(\mathbf{c}, E)$ 
 $L_1 \leftarrow \text{RELU}(\text{CONV1D}(C, \mathbb{K}_{l1}))$ 
 $L_2 \leftarrow \text{CONV1D}(L_1, \mathbb{K}_{l2}) \odot \mathbf{h}_{t-1}$ 
 $L_{feat} \leftarrow L_2 / \|L_2\|_2$ 
return  $L_{feat}$ 

```

Here $E \in \mathbb{R}^{|V| \times D}$ contains the D -dimensional embedding of each subtoken in names and code (i.e. all possible c_i s and m_i s). The two convolution kernels are $\mathbb{K}_{l1} \in \mathbb{R}^{D \times w_1 \times k_1}$ and $\mathbb{K}_{l2} \in \mathbb{R}^{k_1 \times w_2 \times k_2}$, where w_1, w_2 are the window sizes of the convolutions and RELU refers to a rectified linear unit (Nair & Hinton, 2010). The vector $\mathbf{h}_{t-1} \in \mathbb{R}^{k_2}$ represents information from the previous subtokens $m_0 \dots m_{t-1}$. CONV1D performs a one-dimensional (throughout the length of sentence \mathbf{c}) narrow convolution. Note that the

input sequence \mathbf{c} is padded by LOOKUPANDPAD. The size of the padding is such that with the narrow convolutions, the attention vector (returned by **attention_weights**) has exactly $\text{LEN}(\mathbf{c})$ components. The \odot operator is the elementwise multiplication of a vector and a matrix, *i.e.* $B = A \odot \mathbf{v}$ for $\mathbf{v} \in \mathbb{R}^M$ and A a $M \times N$ matrix, $B_{ij} = A_{ij}v_i$. We found the normalization of L_2 into L_{feat} to be useful during training avoiding issues where all the output values of the RELU were below the activation point. We believe it helps because of the widely varying lengths of inputs \mathbf{c} . Note that no pooling happens in this model; the input sequence \mathbf{c} is of the same length as the output sequence (modulo the padding).

To compute the final attention weight vector — a vector with non-negative elements and unit norm — we define **attention_weights** as a function that accepts L_{feat} from **attention_features** and a convolution kernel \mathbb{K} of size $k_2 \times w_3 \times 1$. **attention_weights** returns the normalized attention weights vector with length $\text{LEN}(\mathbf{c})$ (*e.g.* α and κ in Figure 5.1) and is described by the following pseudocode:

```
attention_weights (attention features  $L_{feat}$ , kernel  $\mathbb{K}$ )
    return SOFTMAX(CONV1D( $L_{feat}$ ,  $\mathbb{K}$ ))
```

Computing the State \mathbf{h}_t Predicting the full summary \mathbf{m} is a sequential prediction problem, where each subtoken m_t is sequentially predicted given the previous state containing information about the previous subtokens $m_0 \dots m_{t-1}$. The state is passed through $\mathbf{h}_t \in \mathbb{R}^{k_2}$ computed by a Gated Recurrent Unit (Cho et al., 2014) *i.e.*

```
GRU(current input  $\mathbf{x}_t$ , previous state  $\mathbf{h}_{t-1}$ )
     $\mathbf{r}_t \leftarrow \sigma(\mathbf{x}_t W_{xr} + \mathbf{h}_{t-1} W_{hr} + \mathbf{b}_r)$ 
     $\mathbf{u}_t \leftarrow \sigma(\mathbf{x}_t W_{xu} + \mathbf{h}_{t-1} W_{hu} + \mathbf{b}_u)$ 
     $\mathbf{c}_t \leftarrow \tanh(\mathbf{x}_t W_{xc} + \mathbf{r}_t \odot (\mathbf{h}_{t-1} W_{hc}) + \mathbf{b}_c)$ 
     $\mathbf{h}_t \leftarrow (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \mathbf{c}_t$ 
    return  $\mathbf{h}_t$ 
```

During testing the next state is computed by $\mathbf{h}_t = \text{GRU}(E_{m_t}, \mathbf{h}_{t-1})$ *i.e.* using the embedding of the current output subtoken m_t . For regularization during training, we use a trick similar to Bengio et al. (2015) and with probability equal to the dropout rate we compute the next state as $\mathbf{h}_t = \text{GRU}(\hat{\mathbf{n}}, \mathbf{h}_{t-1})$, where $\hat{\mathbf{n}}$ is the predicted embedding. This helps the network to become more robust by learning to recover from “mistakes” it may make.

5.1.2 Simple Convolutional Attention Model

We now use the components described above as building blocks for our extreme summarization model. We first build **conv_attention**, a convolutional attention model that uses an attention vector α computed from **attention_weights** to weight directly the embeddings of the tokens in \mathbf{c} and compute the predicted target embedding $\hat{\mathbf{n}} \in \mathbb{R}^D$. It returns a distribution over all subtokens in V .

```

conv_attention (code  $\mathbf{c}$ , previous state  $\mathbf{h}_{t-1}$ )
   $L_{feat} \leftarrow \mathbf{attention\_features}(\mathbf{c}, \mathbf{h}_{t-1})$ 
   $\alpha \leftarrow \mathbf{attention\_weights}(L_{feat}, \mathbb{K}_{att})$ 
   $\hat{\mathbf{n}} \leftarrow \sum_i \alpha_i E_{c_i}$ 
   $\mathbf{n} \leftarrow \text{SOFTMAX}(E \hat{\mathbf{n}}^\top + \mathbf{b})$ 
  return TODIST( $\mathbf{n}, V$ )

```

where $\mathbf{b} \in \mathbb{R}^{|V|}$ is a bias vector and TODIST returns a probability distribution over the subtokens in $v_i \in V$ assigning to each v_i probability n_i . We train this model using maximum likelihood. Generating from the model works as follows: starting with the special $m_0 = \langle m \rangle$ subtoken and \mathbf{h}_0 , at each timestep t the next subtoken m_t is generated using the probability distribution \mathbf{n} returned by **conv_attention** ($\mathbf{c}, \mathbf{h}_{t-1}$). Given the new subtoken m_t , we compute the next state $\mathbf{h}_t = \text{GRU}(E_{m_t}, \mathbf{h}_{t-1})$. The process stops when the special $\langle /m \rangle$ subtoken is generated.

5.1.3 Copy Convolutional Attention Model

We extend **conv_attention** by using an additional attention vector κ as a copying mechanism that can suggest out-of-vocabulary subtokens. In our data a significant proportion of the output subtokens (about 35%) appear in \mathbf{c} . Motivated by this, we extend **conv_attention** and allow a direct copy from the input sequence \mathbf{c} into the summary. Now the network when predicting m_t , with probability λ copies a token from \mathbf{c} into m_t and with probability $1 - \lambda$ predicts the target subtoken as in **conv_attention**. Essentially, λ acts as a *meta-attention* mechanism. When copying, a token c_i is copied into m_t with probability equal to the attention weight κ_i . The process is the following:

```

copy_attention (code  $\mathbf{c}$ , previous state  $\mathbf{h}_{t-1}$ )
   $L_{feat} \leftarrow \mathbf{attention\_features}(\mathbf{c}, \mathbf{h}_{t-1})$ 
   $\alpha \leftarrow \mathbf{attention\_weights}(L_{feat}, \mathbb{K}_{att})$ 
   $\kappa \leftarrow \mathbf{attention\_weights}(L_{feat}, \mathbb{K}_{copy})$ 
   $\lambda \leftarrow \max(\sigma(\text{CONV1D}(L_{feat}, \mathbb{K}_\lambda)))$ 

```


$$\begin{aligned}\hat{\mathbf{n}} &\leftarrow \sum_i \alpha_i E_{c_i} \\ \mathbf{n} &\leftarrow \text{SOFTMAX}(E \hat{\mathbf{n}}^\top + \mathbf{b}) \\ \text{return } &\lambda \cdot \text{POS2DIST}(\kappa, \mathbf{c}) + (1 - \lambda) \cdot \text{ToDIST}(\mathbf{n}, V)\end{aligned}$$

where σ is the sigmoid function, \mathbb{K}_{att} , \mathbb{K}_{copy} and \mathbb{K}_λ are different convolutional kernels, $\mathbf{n} \in \mathbb{R}^{|V|}$, $\alpha, \kappa \in \mathbb{R}^{\text{LEN}(\mathbf{c})}$, POS2DIST returns the copy mechanism attention as a probability distribution over the subtokens in \mathbf{c} (which may include out-of-vocabulary subtokens) assigning probability κ_i to each code subtoken c_i . Finally, the predictions of the two attention mechanisms are merged, returning a probability distribution for all potential target subtokens in $V \cup \mathbf{c}$ and interpolating over the two attention mechanisms, using the (adaptive) meta-attention weight λ . Note that α and κ are analogous attention weights but are computed from different kernels, and that \mathbf{n} is computed exactly as in **conv_attention**. Note that although the usage of POS2DIST and ToDIST may seem unnecessarily complicated, they are needed for allowing the model to predict out-of-vocabulary (OOV) subtokens. If our model predicts an OOV subtoken, we compute the next state as $\mathbf{h}_t = \text{GRU}(E_{\text{UNK}}, \mathbf{h}_{t-1})$.

Objective To obtain signal for the copying mechanism and λ , we use a supervised objective. We input a binary vector $\mathbb{1}_{\mathbf{c}=m_t}$ to **copy_attention**. $\mathbb{1}_{\mathbf{c}=m_t}$ is of size $\text{LEN}(\mathbf{c})$ where each component is one if the code subtoken is identical to the current target subtoken m_t . We can then compute the probability of a correct copy over the marginalization of the two mechanisms, *i.e.*

$$P(m_t | \mathbf{h}_{t-1}, \mathbf{c}) = \lambda \sum_i \kappa_i \mathbb{1}_{c_i=m_t} + (1 - \lambda) \mu r_{m_t} \quad (5.1)$$

where the first term is the probability of a correct copy (weighted by λ) and the second term is the probability of the target token m_t (weighted by $1 - \lambda$). The term $\mu \in (0, 1]$ acts as a supervised penalty to avoid the situation where the model's simple attention gets credit for predicting an UNK although the current subtoken m_t can be predicted exactly by the copy mechanism. In all other cases, we set $\mu = 1$. When penalizing UNK predictions, we arbitrarily used $\mu = e^{-10}$, although variations did not affect performance.

5.1.4 Predicting Names

To predict a full method name, we use a hybrid breath-first search and beam search. We start from the special $m_0 = \langle \mathbf{m} \rangle$ subtoken and maintain a (max-)heap of the highest probability partial predictions so far. At each step, we pick the highest probability prediction and predict its next subtokens, pushing them back to the heap. When the

</m> subtoken is generated the suggestion is moved onto the list of suggestions. Since we are interested in the top k suggestions, at each point, we prune partial suggestions that have a probability less than the current best k th full suggestion. To make the process tractable, we limit the partial suggestion heap size and stop iterating after 100 steps.

5.2 Evaluation

Dataset Collection We are interested in the method naming problem where we “summarize” a source code snippet into a short and concise method-like name. This is a general problem with multiple applications. Such a functionality may be useful within the widely used “extract method” refactoring (Silva et al., 2016) within an IDE, where our model suggests a name for the extracted code. Suggesting alternative names can also be useful within the code review process (Section 4.1). Such summarization methods can also find use within search engines that require to interpret natural language queries about code or perform query expansion-like techniques. Finally, summarization can be useful for educational purposes where an summary is generated to aid code comprehension. However, no dataset containing summaries of arbitrary snippets of source code currently exists. Therefore, it is natural to consider existing method (function) bodies as our snippets and the method names picked by the developers as our target extreme summaries.

To collect a dataset of good quality, we cloned 11 open-source Java projects from GitHub. We obtained the most popular projects by taking the sum of the z -scores of the number of watchers and forks of each project, using GHTorrent (Gousios & Spinellis, 2012). We selected the top 11 projects that contained more than 10MB of source code files each and use libgdx as a development set. These projects have thousands of forks and stars, being widely known among software developers. The projects along with short descriptions are shown in Table 5.1. We used this procedure to select a mature, large, and diverse corpus of real source code. For each file, we extract the Java methods, removing methods that are overridden, are abstract or are the constructors of a class. We find the overridden methods by an approximate static analysis that checks for inheritance relationships and the `@Override` annotation. Overridden methods are removed, since they are highly repetitive and their names are easy to predict. Any full tokens that are identical to the method name (*e.g.* in recursion) are replaced with a special SELF token. We split and lowercase each method name and code token into subtokens $\{m_i\}$ and $\{c_i\}$ on camelCase and snake_case. The dataset and code can be found at

Table 5.1: Open-source Java projects used and F1 scores achieved. F1 measures the retrieval performance over the subtokens of each method name.

Project Name	Git SHA	Description	F1							
			tf-idf		Bahdanau et al. (2015)		conv_attention		copy_attention	
			Rank 1	Rank 5	Rank 1	Rank 5	Rank 1	Rank 5	Rank 1	Rank 5
cassandra	53e370f	Distributed Database	40.9	52.0	35.1	45.0	46.5	60.0	48.1	63.1
elasticsearch	485915b	REST Search Engine	27.8	39.5	20.3	29.0	30.8	45.0	31.7	47.2
gradle	8263603	Build System	30.7	45.4	23.1	37.0	35.3	52.5	36.3	54.0
hadoop-common	42a61a4	Map-Reduce Framework	34.7	48.4	27.0	45.7	38.0	54.0	38.4	55.8
hibernate-orm	e65a883	Object/Relational Mapping	53.9	63.6	49.3	55.8	57.5	67.3	58.7	69.3
intellij-community	d36c0c1	IDE	28.5	42.1	23.8	41.1	33.1	49.6	33.8	51.5
libgdx*	156f7c1	Game Dev Framework	41.8	53.1	40.2	46.3	47.0	60.5	50.2	63.0
liferay-portal	39037ca	Portal Framework	59.6	70.8	55.4	70.6	63.4	75.5	65.9	78.0
presto	4311896	Distributed SQL query engine	41.8	53.2	33.4	41.4	46.3	59.0	46.7	60.2
spring-framework	826a00a	Application Framework	35.7	47.6	29.7	41.3	35.9	49.7	36.8	51.9
wildfly	c324eaa	Application Server	45.2	57.7	32.6	44.4	45.5	61.0	44.7	61.7

* libgdx was used for hyperparameter optimization.

<http://groups.inf.ed.ac.uk/cup/codeattention/>.

Experimental Setup To measure the quality of our suggestions, we compute two scores. *Exact match* is the percentage of the method names predicted exactly, while the F1 score is computed in a per-subtoken basis. When suggesting summaries, each model returns a ranked list. We compute exact match and F1 at rank 1 and 5, as the best score achieved by any one of the top suggestions (*i.e.* if the fifth suggestion achieves the best F1 score, we use this one for computing F1 at rank 5). Using BLEU (Papineni et al., 2002) would have been possible, but it would not be different from F1 given the short lengths of our output sequences (3 on average). We train and test each model separately per project. This is because each project’s domain varies widely and little information can be transferred among them, due to the principle of code reusability of software engineering. We note that we attempted to train a single model using all project training sets but this yielded significantly worse results for all algorithms. For each project, we split the *files* (top-level Java classes) uniformly at random into training (65%), validation (5%) and test (30%) sets. We optimize hyperparameters using Bayesian optimization with Spearmint (Snoek et al., 2012) maximizing F1 at rank 5.

For comparison, we use two algorithms: a tf-idf algorithm that computes a tf-idf vector from the code snippet subtokens and suggests the names of the nearest neighbors using cosine similarity. We also use the standard attention model of Bahdanau et al. (2015) that uses a biRNN and fully connected components, that has been successfully used in machine translation. We perform hyperparameter optimizations following the same protocol on libgdx.

Training. To train **conv_attention** and **copy_attention** we optimize the objective using stochastic gradient descent with RMSProp and Nesterov momentum (Sutskever et al., 2013; Hinton et al., 2012). We use dropout (Srivastava et al., 2014) on all parameters, parametric leaky RELUs (Maas et al., 2013; He et al., 2015) and gradient clipping. Each of the parameters of the model is initialized with normal random noise around zero, except for **b** that is initialized to the log of the empirical frequency of each target token in the training set. For **conv_attention** the optimized hyperparameters are $k_1 = k_2 = 8$, $w_1 = 24$, $w_2 = 29$, $w_3 = 10$, dropout rate 50% and $D = 128$. For **copy_attention** the optimized hyperparameters are $k_1 = 32$, $k_2 = 16$, $w_1 = 18$, $w_2 = 19$, $w_3 = 2$, dropout rate 40% and $D = 128$.

Table 5.2: Evaluation metrics averaged across test projects. LBL refers to the log-bilinear models of Chapter 4. Note that the LBL models has additional information in the form of hand-crafted features such as features about the method signature. The hand-crafted features are discussed in Subsection 5.2.3.

At Rank:	F1 (%)		Exact Match (%)		Precision (%)		Recall (%)	
	1	5	1	5	1	5	1	5
tf-idf	40.0	52.1	24.3	29.3	41.6	55.2	41.8	51.9
Bahdanau et al. (2015)	33.6	45.2	17.4	24.9	35.2	47.1	35.1	42.1
conv_attention	43.6	57.7	20.6	29.8	57.4	73.7	39.4	51.9
copy_attention	44.7	59.6	23.5	33.7	58.9	74.9	40.1	54.2
LBL	15.3	46.0	13.6	25.7	15.6	53.2	15.3	43.3
LBL Subtoken	37.1	51.9	15.1	24.1	53.2	64.1	34.1	47.1

5.2.1 Quantitative Evaluation

Table 5.1 shows the F1 scores achieved by the different methods for each project while Table 5.2 shows a quantitative evaluation, averaged across all test projects. By “Standard Attention” we refer to the machine translation model of [Bahdanau et al. \(2015\)](#). The tf-idf algorithm seems to be performing very well, showing that the bag-of-words representation of the input code is a strong indicator of its name. Interestingly, the standard attention model performs worse than tf-idf in this domain, while **conv_attention** and **copy_attention** perform the best. The copy mechanism gives a good F1 improvement at rank 1 and a larger improvement at rank 5. Although our convolutional attention models have an exact match similar to tf-idf, they achieve a much higher precision compared to all other algorithms. This is because they successfully learn about subtoken conventions, such as the fact that a method returning a boolean may usually start with `is`. As we also observed in Chapter 4, such models tend to be more conservative when making predictions, returning UNK predictions more often than other models, resulting in higher precision.

These differences in the data characteristics could be the cause of the low performance achieved by the model of [Bahdanau et al. \(2015\)](#), which had great success in machine translation. Although source code snippets resemble natural language sentences, they are more structured, much longer and vary greatly in length. In our training sets, each method has on average 72 tokens (median 25 tokens, standard deviation 156) and the output method names are made up from 3 subtokens on average ($\sigma = 1.7$).

OOV Accuracy We measure the out-of-vocabulary (OOV) word accuracy as the percentage of the out-of-vocabulary subtokens (*i.e.* those that have not been seen in the training data) that are correctly predicted by **copy_attention**. On average, across our dataset, 4.4% of the test method name subtokens are OOV. Naturally, the standard attention model and tf-idf have an OOV accuracy of zero, since they are unable to predict those tokens. On average we get a 10.5% OOV accuracy at rank 1 and 19.4% at rank 5. This shows that the copying mechanism is useful in this domain and especially in smaller projects that tend to have more OOV tokens. We note that OOV accuracy varies across projects, presumably due to different coding styles. Finally, we should also note that although the copying mechanism is useful for OOV subtokens, our results suggest that it also helps with non-OOV subtokens, increasing the confidence of generating some subtokens.

Topical vs. Time-Invariant Feature Detection The difference of the performance between the **copy_attention** and the standard attention model of Bahdanau et al. (2015) raises an interesting question. What does **copy_attention** learn that cannot be learned by the standard attention model? One hypothesis is that the biRNN of the standard attention model fails to capture long-range features, especially in very long inputs. To test our hypothesis, we shuffle the subtokens in libgdx, essentially removing all features that depend on the sequential information. Without any local features all models should reduce to achieving performance similar to tf-idf. Indeed, **copy_attention** now has an F1 at rank 1 that is +1% compared to tf-idf (presumably because the output GRU acts as a language model over the summary), while the standard attention model further worsens its performance getting an F1 score (rank 1) of 26.2%, compared to the original 41.8%. This suggests that the biRNN fails to capture long-range topical attention features, which is probably also the reason that it fails to beat tf-idf in this summarization task.

A simpler \mathbf{h}_{t-1} Since the target summaries are quite short, we tested a simpler alternative to the GRU, assigning $\mathbf{h}_{t-1} = W \times [G_{m_{t-1}}, G_{m_{t-2}}]$, where $G \in \mathbb{R}^{D \times |V|}$ is a new embedding matrix (different from the embeddings in E) and W is a $k_2 \times D \times 2$ tensor. This model is simpler and slightly faster to train and resembles the log-bilinear language model of Mnih & Teh (2012). This models achieves similar performance to **copy_attention**, reducing F1 by less than 1%. We believe that this happens because summaries tend to be quite short and therefore the long-term memory of RNN architectures does not provide a significant advantage.

Table 5.3: A sample of handpicked snippets and the respective suggestions that illustrate some interesting challenges of the domain and how the **copy_attention** model handles them or fails. Note that the algorithms do *not* have access to the signature of the method but only to the body. Examples taken from the libgdx Android/Java graphics library test set.

<code>boolean shouldRender()</code>	<code>void reverseRange(Object[] a, int lo, int hi)</code>
<pre> 1 try { 2 return renderRequested isContinuous; 3 } finally { 4 renderRequested = false; 5 }</pre>	<pre> 1 hi--; 2 while (lo < hi) { 3 Object t = a[lo]; 4 a[lo++] = a[hi]; 5 a[hi--] = t; 6 }</pre>
<p><u>Suggestions:</u> ▶is,render (27.3%) ▶is,continuous (10.6%)</p> <p>▶is,requested (8.2%) ▶render,continuous (6.9%)</p> <p>▶get,render (5.7%)</p>	<p><u>Suggestions:</u> ▶reverse,range (22.2%) ▶reverse (13.0%)</p> <p>▶reverse,lo (4.1%) ▶reverse,hi (3.2%) ▶merge,range (2.0%)</p>
<code>int createProgram()</code>	<code>VerticalGroup right()</code>
<pre> 1 GL20 gl = Gdx.gl20; 2 int program = gl.glCreateProgram(); 3 return program != 0 ? program : -1;</pre>	<pre> 1 align = Align.right; 2 align &= ~Align.left; 3 return this;</pre>
<p><u>Suggestions:</u> ▶create (18.36%) ▶init (7.9%) ▶render (5.0%)</p> <p>▶initiate (5.0%) ▶load (3.4%)</p>	<p><u>Suggestions:</u> ▶left (21.8%) ▶top (21.1%) ▶right (19.5%)</p> <p>▶bottom (18.5%) ▶align (3.7%)</p>

Table 5.3: A sample of handpicked snippets and the respective suggestions that illustrate some interesting challenges of the domain and how the **copy_attention** model handles them or fails. Note that the algorithms do *not* have access to the signature of the method but only to the body. Examples taken from the libgdx Android/Java graphics library test set.

<code>boolean isBullet()</code>	<code>float getAspectRatio()</code>
<pre> 1 return (m_flags & e_bulletFlag) 2 == e_bulletFlag; </pre> <p><u>Suggestions:</u> ▶is (13.5%) ▶is,bullet (5.5%) ▶is,enable (5.1%) ▶enable (2.8%) ▶mouse (2.7%)</p>	<pre> 1 return (height == 0) ? 2 Float.NaN : width / height; </pre> <p><u>Suggestions:</u> ▶get,UNK (9.0%) ▶get,height (8.7%) ▶get,width (6.5%) ▶get (5.7%) ▶get,size (4.2%)</p>
<code>int minRunLength(int n)</code>	<code>JsonWriter pop()</code>
<pre> 1 if (DEBUG) assert n >= 0; 2 int r = 0; 3 while (n >= MIN_MERGE) { 4 r = (n & 1); 5 n >>= 1; 6 } 7 return n + r; </pre> <p><u>Suggestions:</u> ▶min (43.7%) ▶merge (13.0%) ▶pref (1.9%) ▶space (1.0%) ▶min,all (0.8%)</p>	<pre> 1 if (named) throw 2 new IllegalStateException(UNKSTRING); 3 stack.pop().close(); 4 current = stack.size == 0 ? 5 null : stack.peek(); 6 return this; </pre> <p><u>Suggestions:</u> ▶close (21.4%) ▶pop (10.2%) ▶first (6.5%) ▶state (3.8%) ▶remove (2.2%)</p>

Table 5.3: A sample of handpicked snippets and the respective suggestions that illustrate some interesting challenges of the domain and how the **copy_attention** model handles them or fails. Note that the algorithms do *not* have access to the signature of the method but only to the body. Examples taken from the libgdx Android/Java graphics library test set.

Rectangle setPosition(float x, float y)	float surfaceArea()
<pre>1 this.x = x; 2 this.y = y; 3 return this;</pre>	<pre>1 return 4 * MathUtils.PI * 2 this.radius * this.radius;</pre>
<u>Suggestions:</u> ▶set (54.0%) ▶set,y (12.8%) ▶set,x (9.0%) ▶set,position (8.6%) ▶set,bounds (1.68%)	<u>Suggestions:</u> ▶dot,radius (26.5%) ▶dot (13.1%) ▶crs,radius (9.0%) ▶dot,circle (6.5%) ▶crs (4.1%)

5.2.2 Qualitative Evaluation

Figure 5.2a shows a visualization of a small method that illustrates how **copy_attention** typically works. At the first step, it focuses its attention at the whole method and decides upon the first subtoken. In a large number of cases this includes subtokens such as `get`, `set`, `is`, `create` *etc.* In the next steps the meta-attention mechanism is highly confident about the copying mechanism ($\lambda = 0.97$ in Figure 5.2a) and sequentially copies the correct subtokens from the code snippet into the name. We note that across many examples the copying mechanism tends to have a significantly more focused attention vector κ , compared to the attention vector α . Presumably, this happens because of the different training signals of the attention mechanisms.

A second example of **copy_attention** is seen in Figure 5.2b. Although due to space limitations this is a relatively short method, it illustrates how the model has learned both time-invariant features and topical features. It correctly detects the `==` operator and predicts that the method has a high probability of starting with `is`. Furthermore, in the next step (prediction of the `m2` bullets subtoken) it successfully learns to ignore the `e` prefix (prepended on all enumeration variables in that project) and the `flag` subtoken that does not provide useful information for the summary.

Table 5.3 presents a set of hand-picked examples from `libgdx` that show interesting challenges of the method naming problem and how our **copy_attention** handles them. In the `shouldRender` method, understandably, the model does *not* distinguish between `should` and `is` — both implying a `boolean` return value — and instead of `shouldRender`, `isRender` is suggested. The correct decision could have been made if the model had adequate knowledge about the linguistic structure of the returned variable `renderRequested`, where the `requested` (verb in past participle) suggests that the method should start with a `should` rather than an `is`.

The `getAspectRatio`, `surfaceArea` and `minRunLength` examples show the challenges of describing a previously unseen abstractions. Although a human might have known that the width to height ratio is called “aspect ratio”, the algorithm has no knowledge about this abstraction. Interestingly, the model correctly recognizes that a novel (UNK) token should be predicted after `get` in `getAspectRatio`. In contrast, the `getPosition` method is correctly predicted (as the fourth choice), suggesting that the notion of “position” is learned from the model. These examples illustrate a very interesting and important challenge for machine learning, *i.e.* how models can learn and reason about abstractions and new concepts.

Finally, a surprising result, `reverseRange` is predicted correctly, because of the structure of the code, even though no code tokens contain the summary subtokens. This suggests that the model has learned to recognize patterns in the usage of variables and body's control structure, not just paying attention to identifier names.

5.2.3 Comparison with Log-Bilinear Model

The log-bilinear models presented in Section 4.2 can also be used for the method naming problem. In this section, we are interested in comparing the performance of the log-bilinear model to the convolutional attention model presented earlier. However, a direct comparison of the results would be unfair, since the log-bilinear model gets more information (*e.g.* the signature of the method), albeit in a less structured way. Since the log-bilinear model gets more information, this disadvantages the convolutional network that could have also received this information (having access to more information, generally leads to improved performance). However, we are interested in showing how adding features (*e.g.* the return type) but removing the structure of the code tokens affects the performance on the method naming problem.

For the log-bilinear models, we add a set of hand-crafted features that potentially provide valuable information about the name of each method. These features are detailed below.

AST Ancestors are the types of the parent and grandparent AST nodes of the method declaration. These features indicate if a method belongs to a class or a nested (inner) class.

Cyclomatic Complexity The cyclomatic complexity ([McCabe, 1976](#)) (clipped at the maximum value of 10) gives some indication about the complexity of the implementation within the method. For example, we expect this to be useful discriminating a simple getter method with complexity of 1 from an implementation of some complex logic which would have a much larger cyclomatic complexity.

Subtokens of Names of Containing Class, Superclass and Interfaces The subtokens of the name of the class where the method is declared and the subtokens of the names of the superclass the class inherits and interfaces it implements may provide valuable information about the domain and the functionality of the named method.

Containing Class’ Field Subtokens Any subtokens of the fields of the class declaring the method give information about the class’ functionality and could be useful for determining the method’s function and thus name.

Method Body Subtokens The subtokens of all the tokens within the body of the method are necessarily informative about the method’s name. This feature essentially retrieves the information that the tf-idf baseline uses (Table 5.2).

Declaration Modifiers This feature includes any modifiers (*e.g.* `private` or `final`) of the method declaration.

Return Types The return type of a method is probably the single most important feature, since it provides important indications about the method’s name. For example, a function that returns `boolean` may commonly start with the `is` subtoken.

Sibling Method Name Subtokens The sibling method name subtokens also provide some indications about the name of a method, since they provide some indications about the class’ functionality.

Number of Arguments This feature indicates the number of arguments that the method receives. For example, this feature is useful for detecting methods that do not receive any arguments (*e.g.* getters).

Exceptions Thrown this feature returns the types of the exceptions that the method may throw. Such features may be useful for determining the method functionality, *e.g.* if an `IOException` is thrown, it suggests that the method deals with I/O.

In previous work (Allamanis et al., 2015a), we found that all these features help to some extent, where the class, superclass and interface name subtokens and the return type features were the most helpful.

Discussion Table 5.2 presents a comparison of the performance of the simple LBL — predicting whole method names — and the subtoken LBL model compared to the models previously presented. Both LBL models perform worse compared to the convolutional attention model. It is unsurprising to see that the subtoken model performs better than the simple LBL. This can be attributed to the fact that method names have a high rate of neologisms and suggesting previously used names is not very useful. In contrast, the subtoken LBL performs slightly worse compared to the convolutional attention model, showing that it has also learned subtoken naming conventions using

the provided features. This clearly indicates the informativeness of the features used by the LBL models but also the importance of the structure of the method content that the convolutional attention model has learned to exploit.

We note, that the neural log-bilinear model learns to name methods using a large set of hand-crafted features that are available upon parsing the code and include the whole class context as well as the method signature. In contrast, the convolutional attention model learns to name a method without any features and using solely the body tokens while still achieving better performance. As future work, we observe that adding extra features — when available — to the convolutional attention model will further improve performance on the method naming problem.

5.3 Learned Representations

The log-bilinear model discussed in Subsection 5.2.3 learns easily-visualizable distributed vector representations. Similar to Section 4.5, we present here the representations learned by the log-bilinear model. In contrast, the subtoken LBL model and the convolutional attention network (Section 5.1), since they predict single subtokens, do not produce embeddings for the full method name.

Figure 5.3 displays the vectors assigned to a few method names from a typical project (elasticsearch). Each point represents the \mathbf{q} vector of the indicated token. To interpret this, recall that the model uses the \mathbf{q}_t vectors to predict whether token t will occur in particular context. Therefore, tokens t and t' with similar vectors \mathbf{q}_t and $\mathbf{q}_{t'}$ are tokens that the model expects to occur in similar contexts. These embeddings were generated from the log-bilinear context model — that is, without using subtokens — so the model has no information about which tokens are textually similar. Rather, the only information that the model can exploit is the contexts in which the tokens are used. Despite this, we notice that many of the names which are grouped together seem to have similar functions. For example, there is a group of `assertXXXX` methods on the left hand side. Especially striking is the clump of construction methods on the right-hand side `newDoubleArray`, `newIntArray`, `newLongArray`, and so on. It is also telling that near this clump, the names `grow` and `resize` are also close together. Analysis reveals that these names do indeed seem to name methods of different classes that seem to have similar functionality. Our previous work (Allamanis et al., 2014) indicates that developers often prefer such entities to have consistent names.

For Figure 5.4 (as in Section 4.5), we project the D -dimensional embeddings to

Attention Vectors				$\lambda(\%)$
set	α	<code><c>{ this . use <u>Browser</u> Cache = use Browser Cache ; }</c></code>	1.2	
(m_1)	κ	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>		
use	α	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>	97.4	
(m_2)	κ	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>		
browser	α	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>	96.9	
(m_3)	κ	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>		
cache	α	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>	58.3	
(m_4)	κ	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>		
END	α	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>	6.6	
(m_5)	κ	<code><c>{ this . use Browser Cache = use Browser Cache ; }</c></code>		

(a) Visualization for `setUseBrowserCache` in `libgdx`.

Attention Vectors				$\lambda(\%)$
is	α	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>	1.2	
(m_1)	κ	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>		
bullet	α	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>	43.6	
(m_2)	κ	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>		
END	α	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>	17.4	
(m_3)	κ	<code><c>{ return (mFlags & eBulletFlag) == eBulletFlag ; }</c></code>		

(b) Visualization for `isBullet` in `libgdx`. The **copy_attention** captures location-invariant features and the topolity of the input code sequence.

Figure 5.2: Visualizations for **copy_attention**, used to compute $P(m_t | m_0 \dots m_{t-1}, \mathbf{c})$. The darker the color of a subtoken, they higher its attention weight. This relationship is linear. Yellow indicates the attention weight of the **conv_attention** component, while purple the attention of the copy mechanism. Since the values of α are usually spread across the subtokens the colors show a normalized α , i.e. $\alpha / \|\alpha\|_\infty$. In contrast, the copy attention weights κ are usually very peaky and we plot them as-is. Underlined subtokens are out-of-vocabulary. λ shows the meta-attention probability of using the copy attention κ vs. the convolutional attention α . More visualizations of `libgdx` methods can be found at <http://groups.inf.ed.ac.uk/cup/codeattention/>.

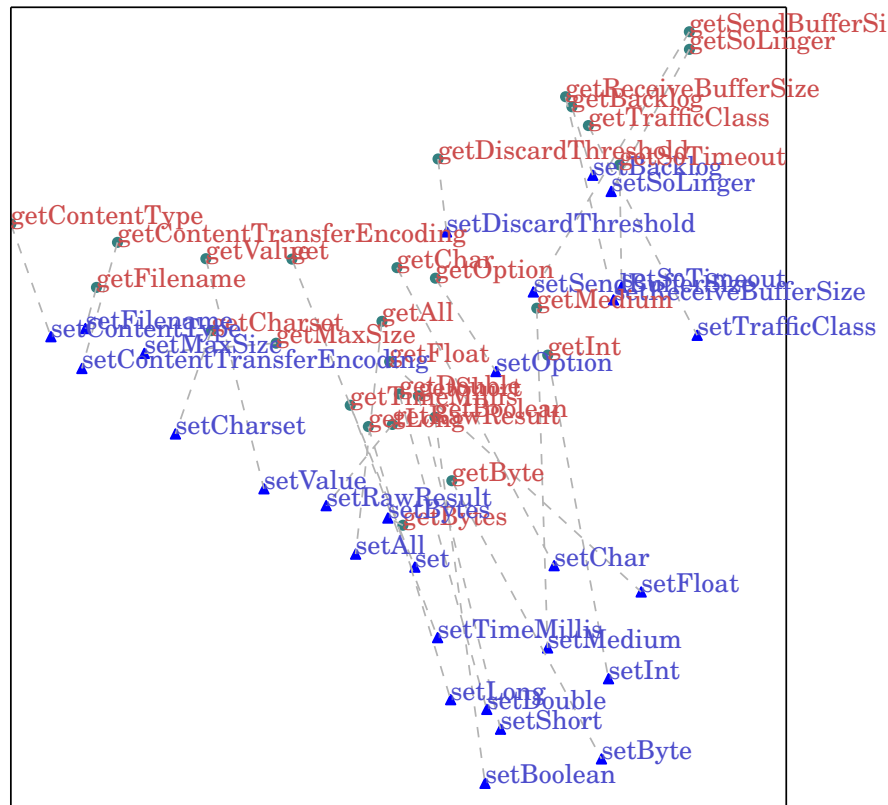


Figure 5.4: A 2D linear projection, using PCA, of the embeddings of setters (blue) and getters (red) for method declarations. Matched getter/setting pairs are connected with a dotted line. The embeddings seem to separate setters from the getters.

names is important for code comprehension and maintainability. Machine learning models — like the one presented in this chapter — can help suggest new conventional names when developers are writing new code or during code review. In addition, such models can help with learning method naming conventions within projects and detect linguistic anti-patterns (Arnaoudova et al., 2015).

From a machine learning perspective, modeling and understanding source code artifacts can have a direct impact in software engineering. The problem of learning to name methods is a first step towards the more general goal of machine learning methods for inferring representations of source code that will allow to probabilistically reason about code, resulting in useful software engineering tools that will help code construction and maintenance.

Chapter 6

Learning Continuous Semantic Representations of Symbolic Expressions

Learning about coding conventions requires knowledge about code semantics. In the previous chapters, we presented methods that used syntactic code features to infer implicit semantic information about coding conventions. However, syntactic differences usually obscure semantics similarities. For example, it is easy to imagine two syntactically different but semantically equivalent snippets of code (*e.g.* two sorting routines). Conversely, two syntactically similar snippets may be semantically very different as commonly exhibited by single-line bugs.

Learning about conventional semantic operations can help us reason about code that is found in software systems. In this chapter, we make the first steps towards learning continuous representations of the semantics of code. These continuous representations will be useful for machine learning methods that need to learn to represent and probabilistically reason about code semantics. Although this is a hard problem, it has important implications when learning (semantic) coding conventions. For example, imagine an operation that is performed with a looping structure. A developer may use a large set of syntactic constructs to express an underlying operation. For example, she may use recursion, `goto` instructions, `for`, or `while` loops to perform an operation with identical semantics. However, if we wish to create machine learning systems that can detect semantic conventions (*e.g.* to detect bugs in semantically conventional operations) the “sparsity” introduced by the multitude the possible syntactic ways of expressing the same operation will confuse our machine learning models and obscure the actual

conventional semantics.

For example, we will illustrate this issue through the well-known and conventional sorting semantic operation. Sorting is a semantically conventional operation, compared to other rare (and probably buggy) operations such as “sort all elements except from one random element” and therefore we could build a probabilistic model to detect non-conventional semantic operations that may signify implementation bugs. At the same time, there are many possible ways to syntactically express a sorting routine. Therefore we need machine learning models that accurately model code semantics by using the syntactic form of source code and learn identical representations for semantically identical but syntactically diverse implementations, while learn different representations for semantically varying but syntactically similar code.

Unfortunately, existing machine learning models cannot achieve this and this chapter presents some early steps towards this direction. Existing machine learning research has extensively focused on learning about *declarative knowledge* but little attention has been given to *procedural knowledge*, *i.e.* knowledge about how to do things, which can be complex yet difficult to articulate explicitly. Only recently, the goal of building systems that learn procedural knowledge has motivated many architectures for learning representations of a single algorithm (Graves et al., 2014; Reed & de Freitas, 2016; Kaiser & Sutskever, 2016). These methods generally learn from execution traces of programs (Reed & de Freitas, 2016) or input-output pairs generated from a program (Graves et al., 2014; Kurach et al., 2015; Riedel et al., 2016; Grefenstette et al., 2015; Neelakantan et al., 2015).

However, the recursive abstraction that is central to procedural knowledge is perhaps most naturally represented not by abstract models of computation, as in that work, but by symbolic representations that have syntactic structure, such as logical expressions and source code. One type of evidence for this claim is the simple fact that people communicate algorithms using mathematical formulas and pseudocode rather than Turing machines. Yet, apart from some notable exceptions (Alemi et al., 2016; Piech et al., 2015; Zaremba et al., 2014) and Chapter 5 of this dissertation, symbolic representations of procedures have received relatively little attention within the machine learning literature as a source of information for representing procedural knowledge.

In this chapter, we address the problem of learning continuous semantic representations (SEMVECs) of arbitrary symbolic expressions. Our goal is to assign continuous vectors to symbolic expressions in such a way that semantically equivalent, but syntactically diverse expressions are assigned to identical (or highly similar) continuous

vectors, when given access to a training set of pairs for which semantic equivalence is known. This is an important but hard problem; learning composable SEMVECs of symbolic expressions requires that we learn about the semantics of symbolic elements and operators and how they map to the continuous representation space, thus encapsulating implicit knowledge about symbolic semantics and its recursive abstractive nature. This is a first step towards learning about “semantic naturalness” of code that will allow machine learning systems to reason about the conventional semantic operations performed within a snippet of code, devoid of any syntactic “distractions”.

However, we are not solely interested in semantic equivalence of symbolic expressions — which can be efficiently computed with existing symbolic methods — although it is a core property that our model needs to capture. The aim of this work is to learn continuous representations of code that abstract syntactic differences but retain semantic information. Continuous semantic representations are differentiable and capture the notion of similarity (as the distance between continuous representations). These properties make continuous representations suitable for learning and reasoning about semantics with machine learning.

This work is similar in spirit to the work of Zaremba et al. (2014), who focus on learning expression representations to aid the search for computationally efficient identities. They use recursive neural networks (TRENN)¹ (Socher et al., 2012) for modeling *homogeneous, single-variable* polynomial expressions. While they present impressive results, we find that the TRENN model fails when applied to more complex symbolic polynomial and boolean expressions. In particular, in our experiments we find that TRENNs tend to assign similar representations to syntactically similar expressions, even when they are semantically very different. The underlying conceptual problem is how to develop a continuous representation that follows syntax but *not too much*, that respects compositionality while also representing the fact that a small syntactic change can be a large semantic one.

To tackle this problem, we propose a new architecture, called *neural equivalence networks* (EQNETs). EQNETs learn how syntactic composition recursively composes SEMVECs, like a TRENN, but are also designed to model large changes in semantics as the network progresses up the syntax tree. As equivalence is transitive, we formulate an objective function for training based on equivalence classes rather than pairwise decisions. The network architecture is based on composing residual-like multi-layer

¹To avoid confusion, we use TRENN for *recursive* neural networks and retain RNN for *recurrent* neural networks.

networks, which allows more flexibility in modeling the semantic mapping up the syntax tree. To encourage representations within an equivalence class to be tightly clustered, we also introduce a training method that we call *subexpression forcing*, which uses an autoencoder to map representations on a low-dimensional space and unifies the representations of semantically identical but syntactically different representations by requiring reversible symbolic operation to produce reversible continuous representations. Experimental evaluation on a highly diverse class of symbolic algebraic and boolean expression types shows that EQNETs dramatically outperform existing architectures like TRENNs and RNNs.

To summarize, the main contributions of our work are:

- We formulate the problem of learning continuous semantic representations of symbolic expressions (SEMVECs) and develop benchmarks for this task.
- We present neural equivalence networks (EQNETs), a neural network architecture that learns to represent expression semantics onto a continuous semantic representation space and how to perform symbolic operations in this space.
- We provide an extensive evaluation on boolean and polynomial expressions, showing that EQNETs perform dramatically better than state-of-the-art alternatives.

Code and data are available at groups.inf.ed.ac.uk/cup/semvec.

6.1 Neural Equivalence Networks

In this work, we are interested in learning semantic, composable representations of mathematical expressions (SEMVEC) and learn to generate identical representations for expressions that are *semantically* equivalent, *i.e.* they belong to the same equivalence class. Equivalence is a stronger property than similarity that is habitually learned by neural networks, since equivalence is additionally a transitive relationship.

Problem Hardness. Finding the equivalence of arbitrary symbolic expressions is a NP-hard problem or worse. For example, if we focus on boolean expressions, reducing an expression to the representation of the `false` equivalence class amounts to proving its non-satisfiability — an NP-complete problem. Of course, we do *not* expect to circumvent an NP-complete problem with neural networks. A network for solving boolean equivalence would require an exponential number of nodes in the size of the

formula if $P \neq NP$. Instead, our goal is to develop architectures whose inductive biases allow them to efficiently learn to solve the equivalence problems for expressions that are similar to a smaller number of expressions in a given training set. This requires that the network learn identical representations for expressions that may be syntactically different but semantically equivalent and also discriminate between expressions that may be syntactically very similar but are non-equivalent. Table 6.1 shows a sample of such expressions that illustrate the hardness of this problem.

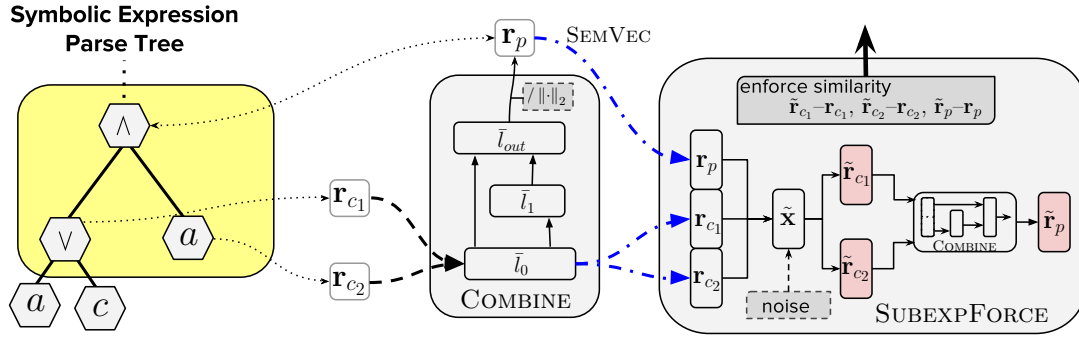
Notation and Framework. We employ the general framework of recursive neural networks (TRENN) (Socher et al., 2012, 2013) to learn to compose subtree representations into a single representation. The TRENNs we consider operate on tree structures of the syntactic parse of a formula. Given a tree T , TRENNs learn distributed representations by recursively computing the representations of its subtrees. We denote the children of a node n as $\text{ch}(n)$ which is a (possibly empty) ordered tuple of nodes. We also use $\text{par}(n)$ to refer to the parent node of n . Each node in our tree has a type, e.g. a terminal node could be of type “a” referring to the variable a or of type “and” referring to a node of the logical AND (\wedge) operation. We refer to the type of a node n as τ_n . At a high level, TRENNs retrieve the representation of a tree T rooted at node ρ , by invoking $\text{TREENET}(\rho)$ that returns a vector representation $\mathbf{r}_\rho \in \mathbb{R}^D$, i.e., a SEMVEC, using the function

```

TREENET(current node  $n$ )
  if  $n$  is not a leaf then
     $\mathbf{r}_n \leftarrow \text{COMBINE}(\text{TREENET}(c_0), \dots, \text{TREENET}(c_k), \tau_n)$ , where  $(c_0, \dots, c_k) = \text{ch}(n)$ 
  else
     $\mathbf{r}_n \leftarrow \text{LOOKUPLEAFEMBEDDING}(\tau_n)$ 
  end if
  return  $\mathbf{r}_n$ 

```

The general framework of TREENET allows two points of variation, the implementation of LOOKUPLEAFEMBEDDING and COMBINE. The traditional TRENNs (Socher et al., 2013) define LOOKUPLEAFEMBEDDING as a simple lookup operation within a matrix of embeddings and COMBINE as a single-layer neural network. As discussed next, these will both prove to be serious limitations in our setting.



(a) Architectural diagram of EQNETs. Example parse tree shown is of the boolean expression $(a \vee c) \wedge a$.

$$\begin{aligned}
 & \text{COMBINE}(\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}, \tau_n) \\
 & \quad \bar{l}_0 \leftarrow [\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}] \\
 & \quad \bar{l}_1 \leftarrow \sigma(W_{i, \tau_n} \cdot \bar{l}_0) \\
 & \quad \bar{l}_{out} \leftarrow W_{o0, \tau_n} \cdot \bar{l}_0 + W_{o1, \tau_n} \cdot \bar{l}_1 \\
 & \quad \text{return } \bar{l}_{out} / \|\bar{l}_{out}\|_2 \\
 & \text{SUBEXPFORCE}(\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}, \mathbf{r}_n, \tau_n) \\
 & \quad \mathbf{x} \leftarrow [\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}] \\
 & \quad \tilde{\mathbf{x}} \leftarrow \tanh(W_d \cdot \tanh(W_{e, \tau_n} \cdot [\mathbf{r}_n, \mathbf{x}] \cdot \mathbf{n})) \\
 & \quad \tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} \cdot \|\mathbf{x}\|_2 / \|\tilde{\mathbf{x}}\|_2 \\
 & \quad \tilde{\mathbf{r}}_n \leftarrow \text{COMBINE}(\tilde{\mathbf{x}}, \tau_n) \\
 & \quad \text{return } -(\tilde{\mathbf{x}}^\top \mathbf{x} + \tilde{\mathbf{r}}_n^\top \mathbf{r}_n)
 \end{aligned}$$

(b) COMBINE of EQNET.

(c) Loss function used for subexpression forcing

Figure 6.1: EQNET architecture.

6.1.1 Neural Equivalence Networks

We now define the *neural equivalence networks* (EQNET) that learn to compose representations of equivalence classes into new equivalence classes (Figure 6.1a). Our network follows the TREENN architecture, that is, our EQNETs are implemented using the TREENET, so as to model the compositional nature of symbolic expressions. However, the traditional TREENNs (Socher et al., 2013) use a single-layer neural network at each tree node. During our preliminary investigations and in Section 6.2, we found that single layer networks are *not* expressive enough to capture many operations, even a simple XOR boolean operator, because representing these operations required high-curvature operations in the continuous semantic representation space. Instead, we turn to multi-layer neural networks. In particular, we define the COMBINE in Figure 6.1b. This uses a two-layer MLP with a residual-like connection to compute the SEMVEC of each parent node in that syntax tree given that of its children. Each node type τ_n , *e.g.* each logical operator, has a different set of weights. We experimented with deeper networks but this did not yield any improvements. However, as TREENN become deeper, they suffer from optimization issues, such as diminishing and exploding gradients. This is essentially because of the highly compositional nature of tree structures, where the same network (*i.e.* the COMBINE non-linear function) is used recursively, causing it to “echo” its own errors and producing unstable feedback loops. We observe this problem even with only two-layer MLPs, as the overall network can become quite deep when using two layers for each node in the syntax tree.

We resolve this issues in a few different ways. First, we constrain each SEMVEC to have unit norm. That is, we set $\text{LOOKUPLEAFEMBEDDING}(\tau_n) = C_{\tau_n} / \|C_{\tau_n}\|_2$, and we normalize the output of the final layer of COMBINE in Figure 6.1b. The normalization step of \bar{l}_{out} and C_{τ_n} is somewhat similar to layer normalization (Ba et al., 2016), although applying layer normalization directly did not work for our problem. Normalizing the SEMVECs partially resolves issues with diminishing and exploding gradients, and removes a spurious degree of freedom in the semantic representation. As simple as this modification may seem, we found that it was vital to obtaining effective performance, and all of our multi-layer TREENNs converged to low-performing parameters without it.

However this modification is not sufficient, since the network may learn to map expressions from the same equivalence class to multiple SEMVECs in the continuous space. We alleviate this problem using a method that we call *subexpression forcing*.

Subexpression forcing guides EQNET to cluster its output to one location per equivalence class, enforces the reversibility of the representations and unifies the representations of semantically identical but syntactically distinct expressions. The high-level idea will be to use an autoencoder with a bottleneck, to remove irrelevant information from the representations, while minimizing the reconstruction error of parent and child representations together, to encourage dependence in the representations of parents and children. More specifically, we encode each parent-children tuple $[\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}, \mathbf{r}_p]$ containing the (computed) SEMVECs of the children and parent nodes into a low-dimensional space using a denoising autoencoder. We then seek to minimize the reconstruction error of the child representations $(\tilde{\mathbf{r}}_{c_0}, \dots, \tilde{\mathbf{r}}_{c_k})$ as well as the reconstructed parent representation $\tilde{\mathbf{r}}_p$ that can be computed from the reconstructed children. More formally, we minimize the return value of subexpression forcing in Figure 6.1c where \mathbf{n} is a binary noise vector with κ percent of its elements set to zero. Note that the encoder is specific to the parent node type τ_p . Although our subexpression forcing may seem similar to the recursive autoencoders of Socher et al. (2011) it differs significantly in form and purpose, since it acts as an autoencoder on the whole parent-children representation tuple and the encoding is *not* used within the computation of the parent representation.

Subexpression forcing has several desired effects. First, it forces each parent-children tuple to lie in a low-dimensional space, requiring the network to compress information from the individual subexpressions. Second, because the denoising autoencoder is reconstructing parent and child representations together, this encourages child representations to be predictable from parents and siblings. Putting these two together, the goal is that the information discarded by the autoencoder bottleneck will be more syntactic than semantic, assuming that the semantics of child node is more predictable from its parent and sibling than its syntactic realization. The goal is to nudge the network to learn consistent, reversible semantics. Additionally, subexpression forcing has the potential to gradually unify distant representations that belong to the same equivalence class. To illustrate this point, imagine two semantically equivalent c'_0 and c''_0 child nodes of different expressions that have distant SEMVECs, *i.e.* $\|\mathbf{r}_{c'_0} - \mathbf{r}_{c''_0}\|_2 \gg \epsilon$ although $\text{COMBINE}(\mathbf{r}_{c'_0}, \dots) \approx \text{COMBINE}(\mathbf{r}_{c''_0}, \dots)$. In some cases due to the autoencoder noise, the differences between the input tuple $\mathbf{x}', \mathbf{x}''$ that contain $\mathbf{r}_{c'_0}$ and $\mathbf{r}_{c''_0}$ will be non-existent and the decoder will predict a single location $\tilde{\mathbf{r}}_{c_0}$ (possibly different from $\mathbf{r}_{c'_0}$ and $\mathbf{r}_{c''_0}$). Then, when minimizing the reconstruction error, both $\mathbf{r}_{c'_0}$ and $\mathbf{r}_{c''_0}$ will be attracted to $\tilde{\mathbf{r}}_{c_0}$ and eventually should merge.

6.1.2 Training

We train EQNETs from a dataset of expressions whose semantic equivalence is known. Given a training set $\mathcal{T} = \{T_1 \dots T_N\}$ of parse trees of expressions, we assume that the training set is partitioned into equivalence classes $\mathcal{E} = \{e_1 \dots e_J\}$. We use a supervised objective similar to classification; the difference between classification and our setting is that whereas standard classification problems consider a fixed set of class labels, in our setting the number of equivalence classes in the training set will vary with N . Given an expression tree T that belongs to the equivalence class $e_i \in \mathcal{E}$, we compute the probability

$$P(e_i|T) = \frac{\exp(\text{TREENN}(T)^\top \mathbf{q}_{e_i} + b_i)}{\sum_j \exp(\text{TREENN}(T)^\top \mathbf{q}_{e_j} + b_j)} \quad (6.1)$$

where \mathbf{q}_{e_i} are model parameters that we can interpret as representations of each equivalence classes that appears in the training class, and b_i are bias terms. Note that in this work, we only use information about the equivalence class of the whole expression T , ignoring available information about subexpressions. This is without loss of generality, because if we do know the equivalence class of a subexpression of T , we can simply add that subexpression to the training set. Directly maximizing $P(e_i|T)$ would be bad for EQNET since its unit-normalized outputs cannot achieve high probabilities within the softmax. Instead, we train a max-margin objective that maximizes classification accuracy, *i.e.*

$$\mathcal{L}_{\text{ACC}}(T, e_i) = \max \left(0, \arg \max_{e_j \neq e_i, e_j \in \mathcal{E}} \log P(e_j|T) - \log P(e_i|T) + m \right) \quad (6.2)$$

where $m > 0$ is a scalar margin. And therefore the optimized loss function for a single expression tree T that belongs to equivalence class $e_i \in \mathcal{E}$ is

$$\mathcal{L}(T, e_i) = \mathcal{L}_{\text{ACC}}(T, e_i) + \frac{\mu}{|Q|} \sum_{n \in Q} \text{SUBEXPFORCE}(\text{ch}(n), n) \quad (6.3)$$

where $Q = \{n \in T : |\text{ch}(n)| > 0\}$, *i.e.* contains the non-leaf nodes of T and $\mu \in (0, 1]$ a scalar weight. We found that subexpression forcing is counterproductive early in training, before the SEMVECs begin to represent aspects of semantics. So, for each epoch t , we set $\mu = 1 - 10^{-vt}$ with $v \geq 0$.

Instead of the supervised objective that we propose, an alternative option for training EQNET would be a Siamese objective (Chopra et al., 2005) that learns about similarities (rather than equivalence) between expressions. In practice, we found the optimization

to be very unstable, yielding suboptimal performance. We believe that this has to do with the compositional and recursive nature of the task that creates unstable dynamics and the fact that equivalence is a stronger property than similarity.

6.2 Evaluation

Datasets We generate datasets of expressions grouped into equivalence classes from two domains. The datasets from the **BOOL** domain contain boolean expressions and the **POLY** datasets contain polynomial expressions. In both domains, an expression is either a variable, a binary operator that combines two expressions, or a unary operator applied to a single expression. When defining equivalence, we interpret distinct variables as referring to different entities in the domain, so that, *e.g.* the polynomials $c \cdot (a \cdot a + b)$ and $f \cdot (d \cdot d + e)$ are not equivalent. For each domain, we generate “simple” datasets which use a smaller set of possible operators and “standard” datasets which use a larger set of more complex operators. We generate each dataset by exhaustively generating *all* parse trees up to a maximum tree size. All expressions are then simplified into a canonical form in order to determine their equivalence class and are grouped accordingly. Table 6.3 shows the datasets we generated. We also present in Table 6.1 some sample expressions. For the polynomial domain, we also generated **ONEV-POLY** datasets, which are polynomials over a single variable, since they are similar to the setting considered by Zaremba et al. (2014) — although **ONEV-POLY** is still a little more general because it is not restricted to homogeneous polynomials. Learning **SEMVECS** for boolean expressions is already a hard problem; with n boolean variables, there are 2^{2^n} equivalence classes (*i.e.* one for each possible truth table). We split the datasets into training, validation and test sets. We create two test sets, one to measure generalization performance on equivalence classes that were seen in the training data (**SEENEQCLASS**), and one to measure generalization to unseen equivalence classes (**UNSEENEQCLASS**). It is easiest to describe **UNSEENEQCLASS** first. To create the **UNSEENEQCLASS**, we randomly select 20% of all the equivalence classes, and place all of their expressions in the test set. We select equivalence classes only if they contain at least two expressions but less than three times the average number of expressions per equivalence class. We thus avoid selecting very common (and hence trivial to learn) equivalence classes in the testset. Then, to create **SEENEQCLASS**, we take the remaining 80% of the equivalence classes, and randomly split the expressions in each class into training, validation, **SEENEQCLASS** test in the proportions 60%–15%–25%. We

provide the datasets online.

Baselines To compare the performance of our model, we train the following baselines. **TF-IDF**: learns a representation given the tokens of each expression (variables, operators and parentheses). This can capture topical/declarative knowledge but is unable to capture procedural knowledge. **GRU** refers to the token-level gated recurrent unit encoder of Bahdanau et al. (2015) that encodes the token-sequence of an expression into a distributed representation. **Stack-augmented RNN** refers to the work of Joulin & Mikolov (2015) which was used to learn algorithmic patterns and uses a stack as a memory and operates on the expression tokens. We also include two recursive neural network (TRENN) architectures. The **1-layer TRENN** which is the original TRENN also used by Zaremba et al. (2014). We also include a **2-layer TRENN**, where COMBINE is a classic two-layer MLP without residual connections. This shows the effect of SEMVEC normalization and subexpression forcing.

Hyperparameters We tune the hyperparameters of the baselines and EQNET using Bayesian optimization (Snoek et al., 2012), optimizing on a boolean dataset with 5 variables and maximum tree size of 7 (not shown in Table 6.3). We use the average k -NN ($k = 1, \dots, 15$) statistics (described next) as an optimization metric. The selected hyperparameters are detailed in Table 6.2.

6.2.1 Quantitative Evaluation

Metric To evaluate the quality of the learned representations we count the proportion of k nearest neighbors of each expression (using cosine similarity) that belong to the same equivalence class. More formally, given a test query expression q in an equivalence class c we find the k nearest neighbors $\mathbb{N}_k(q)$ of q across all expressions, and define the score as

$$score_k(q) = \frac{|\mathbb{N}_k(q) \cap c|}{\min(k, |c| - 1)}. \quad (6.4)$$

To report results for a given testset, we simply average $score_k(q)$ for all expressions q in the testset.

Evaluation Figure 6.2 presents the average $score_k$ across the datasets for each model. Table 6.3 shows $score_5$ of UNSEENEQCLASS for each dataset. It can be clearly seen that EQNET performs better for all datasets, by a large margin. The only exception is POLY5, where the two-layer TRENN performs better. However, this may have to do with the small size of the dataset. The reader may observe that the simple datasets (containing

BOOL8		
$(\neg a) \wedge (\neg b)$	$(\neg a \wedge \neg c) \vee (\neg b \wedge a \wedge c) \vee (\neg c \wedge b)$	$(\neg a) \wedge b \wedge c$
$a \neg((\neg a) \Rightarrow ((\neg a) \wedge b))$	$c \oplus (((\neg a) \Rightarrow a) \Rightarrow b)$	$\neg((\neg b) \vee ((\neg c) \vee a))$
$\neg((b \vee (\neg(\neg a))) \vee b)$	$\neg((b \oplus (b \vee a)) \oplus c)$	$((a \vee b) \wedge c) \wedge (\neg a)$
$(\neg a) \oplus ((a \vee b) \oplus a)$	$\neg((\neg(b \vee (\neg a))) \oplus c)$	$(\neg((\neg(\neg b)) \Rightarrow a)) \wedge c$
$(b \Rightarrow (b \Rightarrow a)) \wedge (\neg a)$	$((b \vee a) \oplus (\neg b)) \oplus c$	$(c \wedge (c \Rightarrow (\neg a))) \wedge b$
$((\neg a) \Rightarrow b) \Rightarrow (a \oplus a)$	$(\neg((b \oplus a) \wedge a)) \oplus c$	$b \wedge (\neg(b \wedge (c \Rightarrow a)))$
False	$(\neg a) \wedge (\neg b) \vee (\wedge c)$	$\neg a \vee b$
$(a \oplus a) \wedge (c \Rightarrow c)$	$(a \Rightarrow (\neg c)) \oplus (a \vee b)$	$a \Rightarrow ((b \wedge (\neg c)) \vee b)$
$(\neg b) \wedge (\neg(b \Rightarrow a))$	$(a \Rightarrow (c \oplus b)) \oplus b$	$\neg(\neg((b \vee a) \Rightarrow b))$
$b \wedge ((a \vee a) \oplus a)$	$b \oplus (a \Rightarrow (b \oplus c))$	$(\neg a) \oplus (\neg(b \Rightarrow (\neg a)))$
$((\neg b) \wedge b) \oplus (a \oplus a)$	$(b \vee a) \oplus (x \Rightarrow (\neg a))$	$b \vee (\neg((\neg b) \wedge a))$
$c \wedge ((\neg(a \Rightarrow a)) \wedge c)$	$b \oplus ((\neg a) \vee (c \oplus b))$	$\neg((a \Rightarrow (a \oplus b)) \wedge a)$

POLY8		
$-a - c$	c^2	$b^2 c^2$
$(b - a) - (c + b)$	$(c \cdot c) + (b - b)$	$(b \cdot b) \cdot (c \cdot c)$
$b - (c + (b + a))$	$((c \cdot c) - c) + c$	$c \cdot (c \cdot (b \cdot b))$
$a - ((a + a) + c)$	$((b + c) - b) \cdot c$	$(c \cdot b) \cdot (b \cdot c)$
$(a - (a + a)) - c$	$c \cdot (c - (a - a))$	$((c \cdot b) \cdot c) \cdot b$
$(b - b) - (a + c)$	$c \cdot c$	$((c \cdot c) \cdot b) \cdot b$
c	$b \cdot c$	$b - c$
$c - ((c - c) \cdot a)$	$(c - (b - b)) \cdot b$	$(a - (a + c)) + b$
$c - ((a - a) \cdot c)$	$(b - (c - c)) \cdot c$	$(a - c) - (a - b)$
$((a - a) \cdot b) + c$	$(b - b) + (b \cdot c)$	$(b - (c + c)) + c$
$(c + a) - a$	$c \cdot ((b - c) + c)$	$(b - (c - a)) - a$
$(a \cdot (c - c)) + c$	$(b \cdot c) + (c - c)$	$b - ((a - a) + c)$

Table 6.1: Sample expressions for the datasets used.

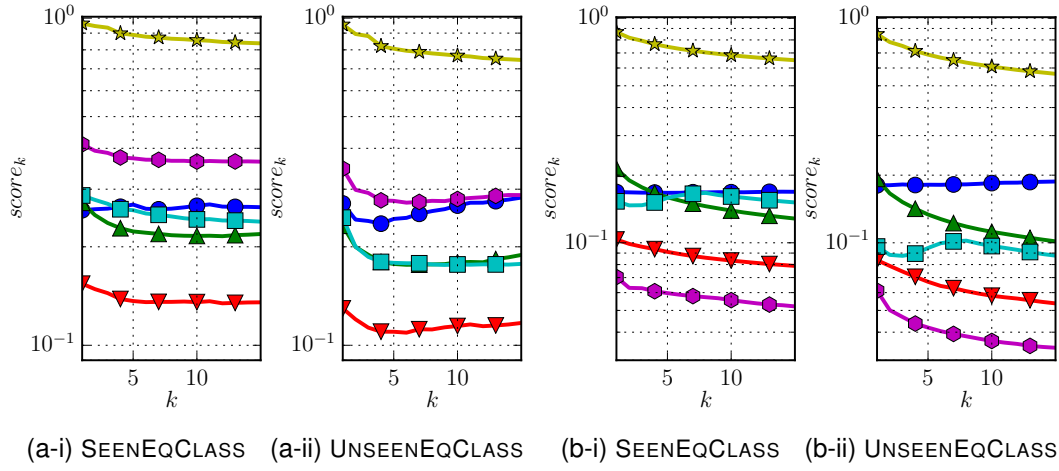
Table 6.2: Hyperparameters used in this work.

Model	Hyperparameters
EQNET	learning rate $10^{-2.1}$, RMSProp $\rho = 0.88$, momentum 0.88, minibatch size 900, representation size $D = 64$, autoencoder size $M = 8$, autoencoder noise $\kappa = 0.61$, gradient clipping 1.82, initial parameter standard deviation $10^{-2.05}$, dropout rate .11, hidden layer size 8, $v = 4$, curriculum initial tree size 6.96, curriculum step per epoch 2.72, objective margin $m = 0.5$
1-layer-TREENN	learning rate $10^{-3.5}$, RMSProp $\rho = 0.6$, momentum 0.01, minibatch size 650, representation size $D = 64$, gradient clipping 3.6, initial parameter standard deviation $10^{-1.28}$, dropout 0.0, curriculum initial tree size 2.8, curriculum step per epoch 2.4, objective margin $m = 2.41$
2-layer-TREENN	learning rate $10^{-3.5}$, RMSProp $\rho = 0.9$, momentum 0.95, minibatch size 1000, representation size $D = 64$, gradient clipping 5, initial parameter standard deviation 10^{-4} , dropout 0.0, hidden layer size 16, curriculum initial tree size 6.5, curriculum step per epoch 2.25, objective margin $m = 0.62$
GRU	learning rate $10^{-2.31}$, RMSProp $\rho = 0.90$, momentum 0.66, minibatch size 100, representation size $D = 64$, gradient clipping 0.87, token embedding size 128, initial parameter standard deviation 10^{-1} , dropout rate 0.26
StackRNN	learning rate $10^{-2.9}$, RMSProp $\rho = 0.99$, momentum 0.85, minibatch size 500, representation size $D = 64$, gradient clipping 0.70, token embedding size 64, RNN parameter weights initialization standard deviation 10^{-4} , embedding weight initialization standard deviation 10^{-3} , dropout 0.0, stack count 40

Table 6.3: Dataset statistics and results. SIMP datasets contain simple operators (“ \wedge , \vee , \neg ” for BOOL and “ $+$, $-$ ” for POLY) while the rest contain all operators (*i.e.* “ \wedge , \vee , \neg , \oplus , \Rightarrow ” for BOOL and “ $+$, $-$, \cdot ” for POLY). \oplus is the XOR operator. The number in the dataset name is the maximum tree size of the parsed expressions within that dataset. L refers to a “larger” number of 10 variables. H refers to the entropy of equivalence classes.

Dataset	# Vars	# Equiv Classes	# Exprs	H	$score_5$ (%) in UNSEEN EQ CLASS					
					tf-idf	GRU	Stack RNN	TREENN 1-L	2-L	EQ NET
SIMPBOOL8	3	120	39,048	5.6	17.4	30.9	26.7	27.4	25.5	97.4
SIMPBOOL10 ^S	3	191	26,304	7.2	6.2	11.0	7.6	25.0	93.4	99.1
BOOL5	3	95	1,239	5.6	34.9	35.8	12.4	16.4	26.0	65.8
BOOL8	3	232	257,784	6.2	10.7	17.2	16.0	15.7	15.4	58.1
BOOL10 ^S	10	256	51,299	8.0	5.0	5.1	3.9	10.8	20.2	71.4
SIMPBOOLL5	10	1,342	10,050	9.9	53.1	40.2	50.5	3.48	19.9	85.0
BOOLL5	10	7,312	36,050	11.8	31.1	20.7	11.5	0.1	0.5	75.2
SIMPPOLY5	3	47	237	5.0	21.9	6.3	1.0	40.6	27.1	65.6
SIMPPOLY8	3	104	3,477	5.8	36.1	14.6	5.8	12.5	13.1	98.9
SIMPPOLY10	3	195	57,909	6.3	25.9	11.0	6.6	19.9	7.1	99.3
ONEV-POLY10	1	83	1,291	5.4	43.5	10.9	5.3	10.9	8.5	81.3
ONEV-POLY13	1	677	107,725	7.1	3.2	4.7	2.2	10.0	56.2	90.4
POLY5	3	150	516	6.7	37.8	34.1	2.2	46.8	59.1	55.3
POLY8	3	1,102	11,451	9.0	13.9	5.7	2.4	10.4	14.8	86.2

^SDatasets are sampled at uniform from all possible expressions, and include all equivalence classes but sampling 200 expressions per equivalence class if more expressions can be formed.



(a) Evaluation of generalization: Models trained and tested on same type of data. Averaged over all datasets in Table 6.3. (b) Evaluation of compositionality; training set simpler than test set. Averaged over all pairs in Figure 6.3c.

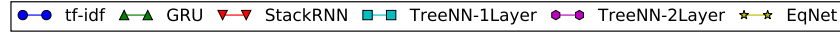
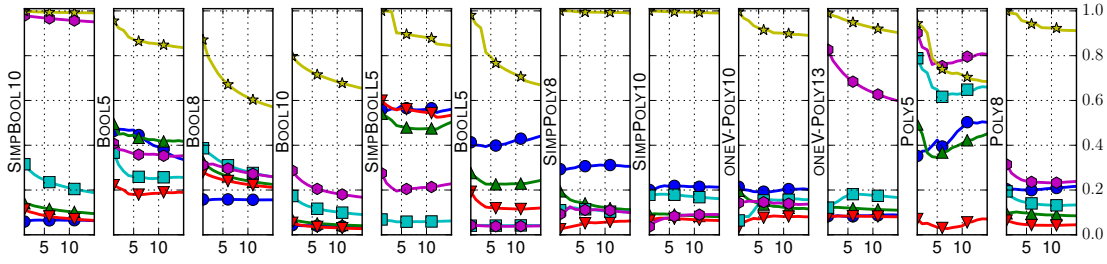


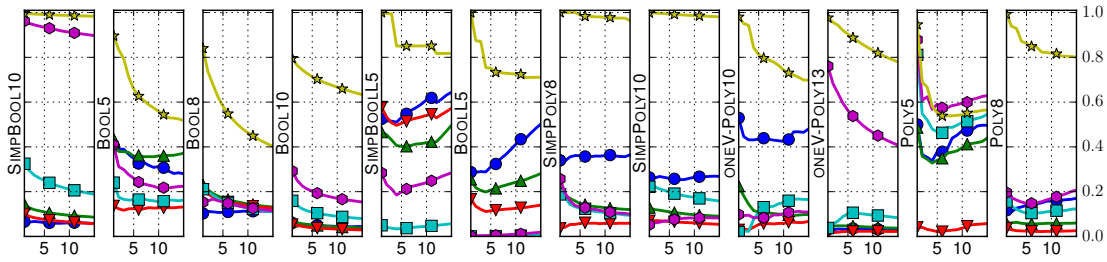
Figure 6.2: Average $score_k$ (y-axis in log-scale). Markers are shown every three ticks for clarity. TRENN refers to [Socher et al. \(2012\)](#). Detailed, per-dataset, plots can be found in Figure 6.3.

fewer operations and variables) are easier to learn. Understandably, introducing more variables increases the size of the represented space reducing performance. The tf-idf method performs better in settings where more variables are included, because it captures well the variables and operations used. Similar observations can be made for sequence models. The one and two layer TRENNs have mixed performance; we believe that this has to do with exploding and diminishing gradients due to the deep and highly compositional nature of TRENNs. Although [Zaremba et al. \(2014\)](#) consider a different problem to us, they use data similar to the ONEV-POLY datasets with a traditional TRENN architecture. Our evaluation suggests that EQNETs perform much better within the ONEV-POLY setting.

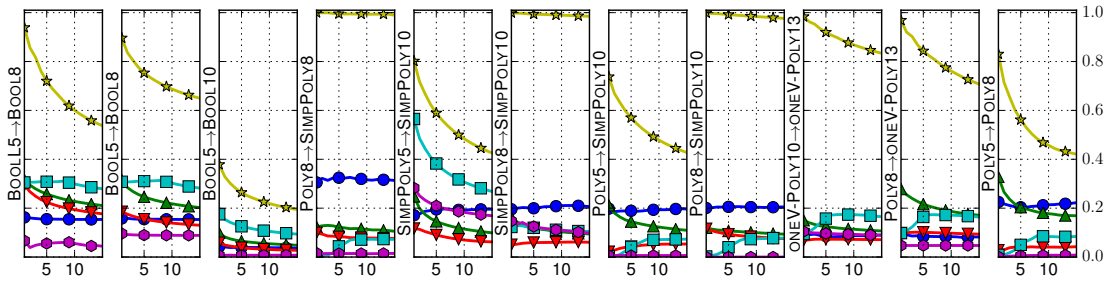
Figure 6.3 presents a detailed evaluation for our k -NN metric for each dataset. Figure 6.3c and Figure 6.3d shows the detailed evaluation when using models trained on simpler datasets but tested on more complex ones, essentially evaluating the learned compositionality of the models. Figure 6.4 show how the performance varies across the datasets based on their characteristics. As expected as the number of variables increase, the performance worsens (Figure 6.4a) and expressions with more complex operators tend to have worse performance (Figure 6.4b). In contrast, Figure 6.4c suggests no



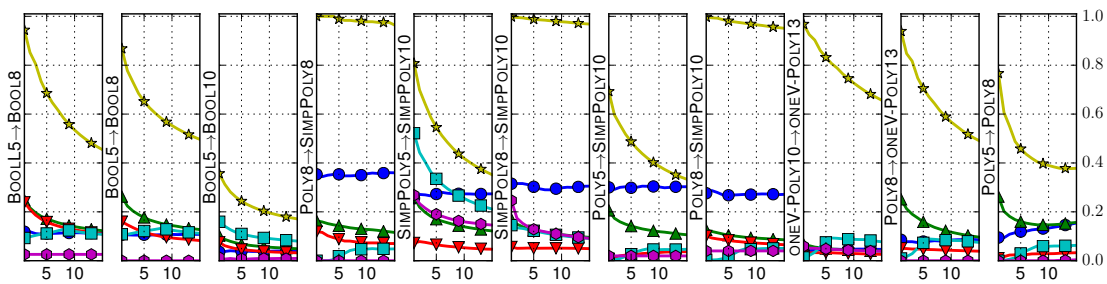
(a) SEENEQCLASS evaluation using model trained on the respective training set.



(b) UNSEENEQCLASS evaluation using model trained on the respective training set.



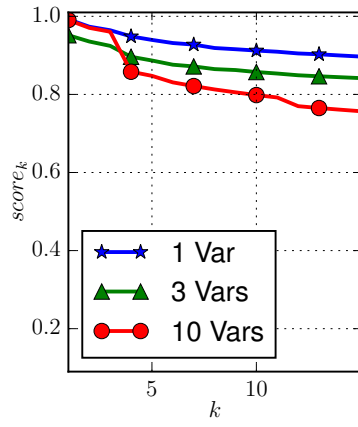
(c) SEENEQCLASS evaluation using model trained on simpler datasets. Caption is “model trained on”→“Test dataset”.



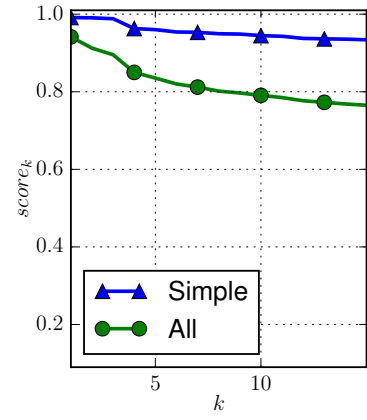
(d) Evaluation of compositionality. UNSEENEQCLASS evaluation using model trained on simpler datasets. Caption is “model trained on”→“Test dataset”.



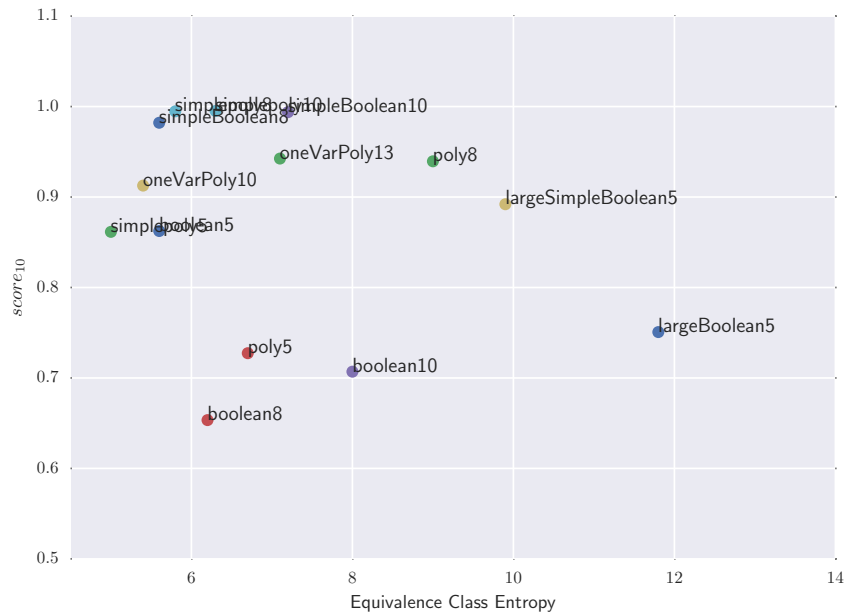
Figure 6.3: Evaluation of $score_x$ (y axis) for $x = 1, \dots, 15$. on the respective SEENEQCLASS and UNSEENEQCLASS. The markers are shown every five ticks of the x -axis to make the graph more clear. TRENN refers to the model of Socher et al. (2012).



(a) Performance vs. Number of Variables.



(b) Performance vs. Operator Complexity.



(c) Entropy H vs. $score_{10}$ for all datasets.

Figure 6.4: Performance of EQNET on SEENEQCLASS for the datasets grouped by their characteristics.

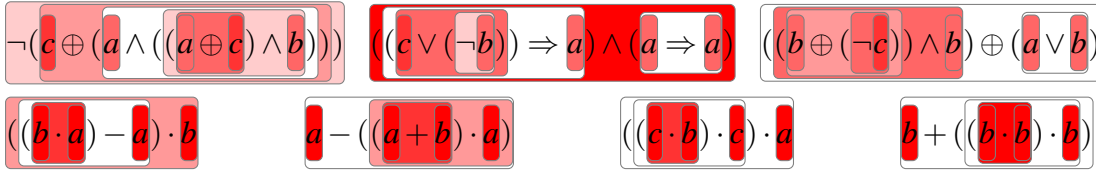


Figure 6.5: Visualization of $score_5$ for all expression nodes for three BOOL10 and four POLY8 test sample expressions using EQNET. The darker the color, the lower the score, *i.e.* white implies a score of 1 and dark red a score of 0.

obvious correlation between performance and the entropy of the equivalence classes within the datasets. The results for UNSEENEQCLASS look very similar and are not plotted.

Evaluation of Compositionality We evaluate whether the EQNETs have successfully learned to compute compositional representations, rather than overfitting to expression trees of a small size. We evaluate this by considering a type of transfer setting, in which we train on simpler datasets, but tested on more complex ones; for example, training on the training set of BOOL5 but testing on the testset of BOOL8. We average over 11 different train-test pairs (full list in Figure 6.3c) and present the results in Figure 6.2b-i and Figure 6.2b-ii (note the differences in scale to the two figures on the left). These graphs again show that EQNETs are dramatically better than any of the other methods, and indeed, performance is only a bit worse than in the non-transfer setting.

Impact of EQNET Components EQNETs differ from traditional TREENNs in two major components, which we analyze here. First, SUBEXPFORCE has a positive impact on performance. When training the network with and without subexpression forcing, on average, the area under the curve (AUC) of the $score_k$ decreases by 16.8% on the SEENEQCLASS and 19.7% on the UNSEENEQCLASS. This difference is smaller in the transfer setting of Figure 6.2b-i and Figure 6.2b-ii, where AUC decreases by 8.8% on average. However, even in this setting we observe that SUBEXPFORCE helps more in large and diverse datasets. The second key difference to traditional TREENNs is the output normalization at each layer. Comparing our model to the one-layer and two-layer TREENNs again, we find that output normalization results in important improvements (the two-layer TREENNs have on average 60.9% smaller AUC).

Table 6.4: *Non* semantically equivalent first nearest-neighbors from BOOL8. A checkmark indicates that the method correctly results in the nearest neighbor being from the same equivalence class.

Expression	$a \wedge (a \wedge (a \wedge (\neg c)))$	$a \wedge (a \wedge (c \Rightarrow (\neg c)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg c))$
tfidf	$c \wedge ((a \wedge a) \wedge (\neg a))$	$c \Rightarrow (\neg((c \wedge a) \wedge a))$	$c \Rightarrow (\neg((c \wedge a) \wedge a))$
GRU	✓	$a \wedge (a \wedge (c \wedge (\neg c)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg c))$
1L-TREENN	$a \wedge (a \wedge (a \wedge (\neg b)))$	$a \wedge (a \wedge (c \Rightarrow (\neg b)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg b))$
EQNET	✓	✓	$(\neg(b \Rightarrow (b \vee c))) \wedge a$

Table 6.5: *Non* semantically equivalent first nearest-neighbors from POLY8. A checkmark indicates that the method correctly results in the nearest neighbor being from the same equivalence class.

Expression	$a + (c \cdot (a + c))$	$((a + c) \cdot c) + a$	$(b \cdot b) - b$
tf-idf	$a + (c + a) \cdot c$	$(c \cdot a) + (a + c)$	$b \cdot (b - b)$
GRU	$b + (c \cdot (a + c))$	$((b + c) \cdot c) + a$	$(b + b) \cdot b - b$
1L-TREENN	$a + (c \cdot (b + c))$	$((b + c) \cdot c) + a$	$(a - c) \cdot b - b$
EQNET	✓	✓	$(b \cdot b) \cdot b - b$

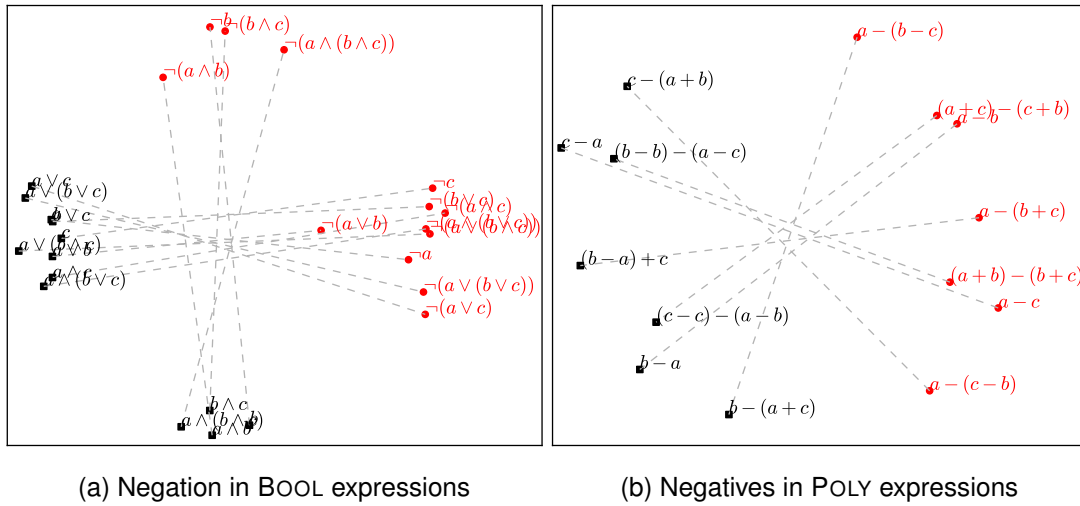


Figure 6.6: A PCA visualization of some simple *non-equivalent* boolean and polynomial expressions (black-square) and their negations (red-circle). The lines connect the negated expressions.

6.2.2 Qualitative Evaluation

Table 6.4 and Table 6.5 shows expressions whose SEMVEC nearest neighbor is of an expression of another equivalence class. Manually inspecting boolean expressions, we find that EQNET confusions happen more when a XOR or implication operator is involved. In fact, we fail to find any confused expressions for EQNET *not* involving these operations in BOOL5 and in the top 100 expressions in BOOL10. As expected, tf-idf confuses expressions with others that contain the same operators and variables ignoring order. In contrast, GRU and TRENN tend to confuse expressions with very similar symbolic representation differing in one or two deeply nested variables or operators. In contrast, EQNET tends to confuse fewer expressions (as we previously showed) and the confused expressions tend to be more syntactically diverse and semantically related.

Figure 6.5 shows a visualization of $score_5$ for each node in the expression tree. One may see that as EQNET knows how to compose expressions that achieve good score, even if the subexpressions achieve a worse score. This suggests that for common expressions, (e.g. single variables and monomials) the network tends to select a unique location, without merging the equivalence classes or affecting the upstream performance of the network. Larger scale interactive t-SNE visualizations can be found online.

Figure 6.6 presents two PCA visualizations of the learned embeddings of simple expressions and their negations/negatives. It can be easily discerned that the black dots

and their negations (in red) are easily discriminated in the semantic representation space. Figure 6.6b shows this property in a very clear manner: left-right discriminates between polynomials with a and $-a$, top-bottom between polynomials that contain b and $-b$ and the diagonal $y = x$ between c and $-c$. We observe a similar behavior in Figure 6.6a for boolean expressions.

6.3 Related Work in Machine Learning

Researchers have proposed compilation schemes that can transform any given program or expression to an equivalent neural network (Gruau et al., 1995; Neto et al., 2003; Siegelmann, 1994). One can consider a serialized version of the resulting neural network as a representation of the expression. However, it is not clear how we could compare the serialized representations corresponding to two expressions and whether this mapping preserves semantic distances.

Recursive neural networks (TREE NN) (Socher et al., 2012, 2013) have been successfully used in NLP with multiple applications. Socher et al. (2012) show that TREE NNs can learn to compute the *values* of some simple propositional statements. EQNET’s SUBEXPFORCE may resemble recursive autoencoders (Socher et al., 2011) but differs in form and function, encoding the whole parent-children tuple to force a clustering behavior. In addition, when encoding each expression our architecture does *not* use a pooling layer but directly produces a single representation for the expression.

This work is related to other probabilistic models of source code discussed in Chapter 2, Mou et al. (2016) use tree convolutional neural networks to classify code into 106 student submissions tasks. Although their model learns intermediate representations of the student tasks, it is a way of learning task-specific features in the code, rather than of learning semantic representations of programs. Piech et al. (2015) also learn distributed matrix representations of programs from student submissions. However, to learn the representations, they use input and output program states and do not test over program equivalence. Additionally, these representations do not necessarily represent program equivalence, since they do not learn the representations over the exhaustive set of all possible input-output states. Allamanis et al. (2016d) learn variable-sized representations of source code snippets to summarize them with a short function-like name. This method aims to learn summarization features in code rather than to learn representations of symbolic expression equivalence.

More closely related is the work of Zaremba et al. (2014) who use a recursive neural

network (TREENN) to guide the tree search for more efficient mathematical identities, limited to homogeneous single-variable polynomial expressions. In contrast, EQNETs consider at a much wider set of expressions, employ subexpression forcing to guide the learned SEMVECs to better represent equivalence, and do *not* use search when looking for equivalent expressions. Alemi et al. (2016) use RNNs and convolutional neural networks to detect features within mathematical expressions and speed the search for premise selection during automated theorem proving but do not explicitly account for semantic equivalence. In the future, SEMVECs may find useful applications within this work.

Our work is also related to recent work on neural network architectures that learn controllers/programs (Gruau et al., 1995; Graves et al., 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Dyer et al., 2015; Reed & de Freitas, 2016; Neelakantan et al., 2015; Kaiser & Sutskever, 2016). In contrast to this work, we do not aim to learn how to evaluate expressions or execute programs with neural network architectures but to learn continuous semantic representations (SEMVECs) of expression semantics irrespective of how they are syntactically expressed or evaluated.

6.4 Conclusions

In this chapter, we presented EQNETs, a first step in learning continuous semantic representations (SEMVECs) of procedural knowledge. SEMVECs have the potential of bridging continuous representations with symbolic representations and their semantics. This will be useful for creating machine learning methods that learn to reason about code semantics and the associated conventions. In the future, such methods may have applications in finding bugs in source code but may also find use in artificial intelligence and existing program analysis methods.

We showed that EQNETs perform significantly better than state-of-the-art alternatives. But further improvements are needed, especially for more robust training of compositional models in machine learning. In addition, even for relatively small symbolic expressions, we have an exponential explosion of the semantic space to be represented. Fixed-sized SEMVECs, like the ones used in EQNET, eventually limit the capacity that is available to represent procedural knowledge. In the future, to represent more complex procedures, variable-sized representations would most probably be necessary.

Chapter 7

Mining Idiomatic Source Code

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

– Martin Fowler. Refactoring: Improving the Design of Existing Code, 1999

Programming language text is a means of human communication. Programmers write code not simply to be executed by a computer, but also to communicate the precise details of the code’s operation to later developers who will adapt, update, test and maintain the code. Programmers themselves use the term *idiomatic* to refer to code that is written in a style that other experienced developers find natural and conventional.¹ Programmers believe that it is important to write idiomatic code, as evidenced by the amount of relevant resources available: For example, Wikibooks has a book devoted to C++ idioms ([Wikibooks, 2013](#)), and similar guides are available for Java ([Java Idioms Editors, 2014](#)) and JavaScript ([Chuan, 2014](#); [Waldron, 2014](#)). A guide on GitHub for idiomatic JavaScript ([Waldron, 2014](#)) has more than 11,949 stars and 1,589 forks. A search for the keyword “idiomatic” on StackOverflow yields over 49,000 hits; all but one of the first 100 hits are questions about what the idiomatic method is for performing a given task.

The notion of *code idiom* is one that is commonly used but seldom defined. We take the view that an idiom is a code fragment that represents a “mental chunk” that humans use when reasoning about some code which describes a single coherent operation. In this chapter, we discuss syntactic and semantic idioms. *Syntactic idioms* represent

¹This use of the word “idiom” refers to notion of style similar to art or music, rather than to linguistic idioms that are groups of words whose meaning is *not* deducible from the meaning of the individual words.

common syntactic structures. For example, the loop `for(int i=0;i<n;i++) { ... }` is common for iterating over an array and although it would be possible to express this operation in many other ways, such as with a `do-while` loop or using recursion, we would find those alternatives alien and more difficult to understand. Similarly, *semantic idioms* represent common semantic concepts which developers use as units of reasoning. For example, a map-reduce operation over a collection is a common semantic idiom, which includes applying a mapping function to each element of the collection and then reducing the mapped values to a single result. Such an idiomatic semantic operation is so common that the simple “map-reduce” term is used to explain the concept among software engineers.

Idioms differ significantly from previous notions of patterns in software, such as code clones (Roy & Cordy, 2007) and API patterns (Zhong et al., 2009). Unlike clones, idioms commonly recur across projects, even ones from different domains, and unlike API patterns, idioms commonly involve syntactic constructs, such as iteration and exception handling. A large number of example syntactic idioms, all of which are automatically identified by our system, are shown in Figures 7.7, 7.8 and 7.9 while Figure 7.15 shows sample semantic idioms. As the reader may notice, all idioms may have metavariables that abstract over identifiers, code blocks and other elements.

Idioms are important to software engineering. First, syntactic idioms are widely useful as a form of documentation of some code construct or API. Already, major IDEs support syntactic idioms by including features that allow programmers to define idioms and easily reuse them. Eclipse’s SnipMatch (Recommenders, 2014) and IntelliJ IDEA’s live templates (JetBrains, 2014) allow the user to define custom snippets of code that can be inserted on demand. NetBeans includes a similar “Code Templates” feature in its editor and Microsoft created Bing Developer Assistant (Microsoft Research, 2014; Zhang et al., 2016) that allows users to search and add snippets to their code, by retrieving code from popular coding websites, such as StackOverflow and other documentation. The fact that all major IDEs include features that allow programmers to manually define and use idioms attests to their importance.

Semantic idioms on the other hand represent patterns that software engineering tool designers need to know to achieve good coverage of their tool. For example, when designing a refactoring tool, toolsmiths implicitly aim to capture common semantic patterns so that the implemented code rewritings can achieve good coverage on real-life code. Similarly, software architects define new APIs that simplify common usage idioms of library or project-internal APIs. Semantic idioms are also the driving force

behind the introduction of new programming language features that cover common language use cases, such as the introduction of the `foreach` construct in Java and C# or the multi-catch statement in Java 7.

We are unaware, however, of methods for *automatically* identifying code idioms. This is a major gap in tooling for software development. Software developers cannot use manual IDE tools for syntactic idioms without significant effort to organize the idioms of interest and then manually add them to the tool. This is especially an obstacle for less experienced programmers who do not know which idioms they should be using. Indeed, as we demonstrate later, many syntactic idioms are library-specific, so even an experienced programmer will not be familiar with the code idioms for a library that is new to them. Although in theory this cost could be amortized if the users of each library were to manually create an idiom guide, in practice even expert developers will have difficulty listing the most “valuable” idioms that they use daily, just as a native speaker of English would have difficulty exhaustively listing all of the words that they know. Perhaps for this reason, although IDEs have included features for manually specifying idioms for many years,² we are unaware of large-scale efforts by developers to list and categorize library-specific idioms. The ability to automatically identify syntactic idioms is needed.

Similarly, designers and tool developers who work to manipulate code need help with automatically identifying common and meaningful patterns, *i.e.* patterns that are easy to manipulate and reason about and implement a cohesive functionality. Tools such as `grep` and manual inspection currently dominate the search for useful patterns and, unfortunately, tend to return frequent but trivial or redundant patterns. These tools also require that the tool developer already has some intuition about existing pattern, something that may not be true especially for project or domain-specific patterns. It is crucial therefore, to assist tool developers to find and rank useful and meaningful patterns. Mining semantic idioms solves this issue by providing data-based evidence mining and ranking code patterns within a codebase.

In this chapter, we present a novel method for automatically mining semantic and syntactic code idioms from an existing corpus of idiomatic code. At first, this might seem to be a simple proposition: simply search for subtrees that occur often in a parsed corpus. However, this naïve method does not work well, for the simple reason that frequent trees are not necessarily interesting trees. To return to our previous example, `for` loops occur more commonly than `for(int i=0;i<n;i++) {...}`, but one would be

²See *e.g.* <http://bit.ly/1nN4hz6>

hard pressed to argue that `for(...) {...}` on its own (that is, with no expressions or body) is an interesting pattern.

Instead, we rely on a different principle: interesting patterns are those that help to explain the code that programmers write. As a measure of “explanation quality”, we use a probabilistic model of the source code, and retain those idioms that make the training corpus more likely under the model. However, a naïve implementation of this idea would also create problems—if we simply create one giant idiom for each file that contains its entire abstract syntax tree (AST), then this will explain the training corpus with high probability, despite being a lousy model. So we also need a method that controls the number of idioms induced, adding a new idiom only if it “explains enough” about the corpus. These ideas can be formalized in a single, theoretically principled framework using a *nonparametric Bayesian* analysis. Nonparametric Bayesian methods have become enormously popular in statistics, machine learning, and natural language processing because they provide a flexible and principled way of automatically inducing a “sweet spot” of model complexity based on the amount of data that is available (Orbanz & Teh, 2010; Gershman & Blei, 2012; Teh & Jordan, 2010). In particular, we employ a *nonparametric Bayesian tree substitution grammar*, which has recently been developed in the field of natural language processing (Cohn et al., 2010; Post & Gildea, 2009), but which has not been applied to source code.

Because our method is primarily statistical in nature, it is language agnostic, and can be applied to any programming language for which one can collect a corpus of previously-written code. Our major contributions are:

- We introduce the idiom mining problem (Section 7.1);
- We present HAGGIS, a method for automatically mining syntactic and semantic code idioms based on nonparametric Bayesian tree substitution grammars (Section 7.2);
- We demonstrate that HAGGIS successfully identifies cross-project syntactic idioms (Section 7.4), for example, 67% of idioms that we identify from one set of open-source projects also appear in an independent set of snippets of example code from the popular Q&A site StackOverflow;
- Examining the syntactic idioms that HAGGIS identifies (Figure 7.7 and Figure 7.8), we find that they describe important program concepts, including object creation, exception handling, and resource management;
- To further demonstrate that the syntactic idioms identified by HAGGIS are semanti-

cally meaningful, we examine the relationship between idioms and code libraries (Subsection 7.4.4), finding that many idioms are strongly connected to package imports in a way that can support suggestion.

- We present a method for extending syntactic idiom mining to mine semantic idioms by abstracting ASTs and embedding additional semantic information in the mining process. We specialize this framework for mining semantic idioms of loop constructs (Section 7.5).
- We demonstrate how semantic loop idioms can be used for API and language design via two case studies: we find that adding `Enumerate` to C# would simplify 12% of loops or adding a certain API in `lucenenet` can reduce the complexity of its code (Section 7.6).
- Finally, we demonstrate how semantic loop idioms can be used by software tool designers via a case study on refactoring. We show that the top 25 loop idioms cover 45% of concrete loops and that semantic idioms can be simply used to suggest loop-to-LINQ refactorings achieving 89% accuracy (Section 7.6).

We submitted a small set of syntactic idioms from HAGGIS to the Eclipse SnipMatch project (Subsection 7.4.3) for inclusion into its pre-supplied library of snippets. Several of these snippets were accepted.

7.1 Problem Definition

A *code idiom* is a code fragment that represents some common concept and acts a “mental chunk” of code that programmers tend to use frequently and assign a coherent meaning. An example of a syntactic idiom is shown in Figure 7.1(b) This is an idiom which is used for manipulating objects of type `android.database.Cursor`, which ensures that the cursor is closed after use (this idiom is indeed discovered by our method). Figure 7.2a shows a semantic idiom discovered by our method that corresponds to a simple reduce operation with a `foreach` in C#. As in these example, idioms have parameters, which we will call *metavariables*, such as the name of the `Cursor` variable, and a code block describing what should be done if the `moveToFirst` operation is successful. An Android programmer who is unfamiliar with this idiom might make mistakes, like not calling the `close` method or not using a `finally` block, causing subtle memory leaks.

Many syntactic idioms, like the `close` example or those in Figure 7.8, are specific

```

1 ...
2 if (c != null) {
3   try {
4     if (c.moveToFirst()) {
5       number = c.getString(
6         c.getColumnIndex(
7           phoneColumn));
8     }
9   } finally {
10    c.close();
11  }
12 }
13 ...

```

(a) A snippet from PhoneNumberUtils in android.telephony.

```

1 try {
2   if ($(Cursor).moveToFirst()) {
3     $BODY$
4   }
5 } finally {
6   $(Cursor).close();
7 }

```

(b) A common idiom when handling android.database.Cursor objects, successfully mined by HAGGIS.



(c) Eclipse JDT's AST for the code in (a). Shaded nodes are those included in the idiom.

Figure 7.1: Example of code idiom extraction.

```
foreach (var $0 in $EXPR$) {
    $BLOCK[UR($0, $1); URW($2);]
}
```

(a) A semantic loop idiom capturing a reduce idiom, which reads the unitary variables \$0 and \$1 and reduces them by writing into the unitary variable \$2.

```
foreach(var refMap$0 in mapping.ReferenceMaps)
    this$2.AddProperties(properties$1, refMap$0.Data.Mapping);
```

(b) Concrete loop from csvhelper that match the semantic loop idiom in Figure 7.2a.

Figure 7.2: A sample semantic idiom and a matching loop.

to particular software libraries. Other syntactic idioms are general across projects of the same programming language, such as those in Figure 7.9, including an idiom for looping over an array or an idiom defining a `String` constant. (All of the idioms in these figures are discovered by our method). As these examples show, idioms are usually *parameterized* and the parameters often have syntactic structure, such as expressions and code blocks. We also make similar observations for semantic idioms, semantic idioms tend to be either general or domain-specific.

We define syntactic idioms as fragments of abstract syntax trees, which allows us to naturally represent the syntactic structure of an idiom. For semantic idioms, we choose to further abstract the AST by removing information not relevant to the semantics of interest to our application and annotating the tree with additional semantic information. Again, we define semantic idioms as fragments of the processed AST structure. Since mining both syntactic and semantic idioms follows the same process on trees, henceforth we will use the term *idiom* to refer to both types of idioms and we will refer to both the abstracted and the traditional ASTs simply as ASTs.

More formally, an idiom is a fragment $\mathcal{T} = (V, E)$ of an (possibly further abstracted) abstract syntax tree (AST). By *fragment*, we mean the following. Let G be the context-free grammar³ of the programming language in question. Then a fragment \mathcal{T} is a tree of terminals and nonterminal from G that is a subgraph of some valid parse tree from G .

³Programming language grammars typically describe parse trees rather than ASTs, but since there is a 1:1 mapping between the two, we assume a context free grammar (CFG) that directly describes ASTs is available.

An idiom \mathcal{T} can have as leaves both terminals and non-terminals. Non-terminals correspond to metavariables which must be filled in when instantiating the idiom. For example, in Figure 7.1(c), the shaded lines represent the fragment for an example idiom; notice how the `Block` node of the AST, which is a non-terminal, corresponds to a `$BODY$` metavariable in the idiom.

Mining Idioms We introduce the *idiom mining problem*, namely, to identify a set of idioms automatically given only a corpus of previously-written idiomatic code. More formally, given a training set of source files or snippets $\{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_N\}$, we first convert them into (possibly abstracted) ASTs using a function

$$f : \mathbb{C} \rightarrow T \quad (7.1)$$

that converts the code into an AST-like structure. For syntactic idioms, f is a traditional AST parser with some small modifications (Section 7.3). For semantic idioms f takes a more complex form, as described in Section 7.5. Thus given a dataset $\mathcal{D} = \{f(\mathbb{C}_1), f(\mathbb{C}_2), \dots, f(\mathbb{C}_N)\} = \{T_1, T_2, \dots, T_N\}$, the idiom mining problem is to identify a set of idioms $I = \{\mathcal{T}_i\}$ that occur in the training set. This is an *unsupervised* learning problem, as we do not assume that we are provided with any example idioms that are explicitly identified. Each fragment \mathcal{T}_i should occur as a subgraph of every tree in some subset $\mathcal{D}(\mathcal{T}_i) \subseteq \mathcal{D}$ of the training corpus.

What Idioms are Not Idioms are not clones. Code clones (Roy & Cordy, 2007; Roy et al., 2009; Baker, 1993; Basit & Jarzabek, 2009; Jiang et al., 2007; Kontogiannis et al., 1996) are pieces of code that are used verbatim (or nearly so) in different code locations usually due to copy paste operations. For example, a large identical code fragment that appears only twice in a codebase *is* a code clone but *not* an idiom. By contrast, idioms are used verbatim (or nearly so) in different code locations because programmers find them natural for performing a particular task. Essentially, idioms have a semantic purpose that developers are consciously aware of. Indeed, unlike clones we suggest that idioms are not typically entered by copy-paste — speaking for ourselves, we do not need copy-paste to enter something as simple as `for(int i=0; i<n; i++)`. Rather, we suggest that programmers treat idioms as mental chunks, which they often type directly by hand when needed, although we leave this conjecture to future work.

Because methods for clone detection work by finding repeated regions of code, existing clone detection methods could also be applied to find idioms. However, in our experiments (Subsection 7.4.2), this does not prove to be an effective approach. We argue that this highlights a conceptual difference between clone detection and idiom

detection: Clone detection methods attempt to find the largest fragment that is copied (even if only once), whereas methods for idiom detection need to search for fragments that seem “natural” to programmers, which requires a trade off between the size of the fragment and the frequency with which programmers use it.

Also, idiom mining is not API mining. API mining (Nguyen et al., 2009; Wang et al., 2013; Zhong et al., 2009) is an active research area that focuses on mining groups of library functions from the same API that are commonly used together. These types of patterns that are inferred are essentially sequences, or sometimes finite state machines, of method invocations. Although API patterns are valuable, idiom mining is markedly different, because idioms have syntactic structure and can even contain no API calls. For example, current API mining approaches cannot find patterns such as a library with a `Tree` class that requires special iteration logic, or a Java library that requires the developer to free resources within a `finally` block. This is exactly the type of pattern that HAGGIS identifies.

7.2 Mining Idioms

In this section, we introduce the technical framework that is required for HAGGIS,⁴ our proposed method for the idiom mining problem. At a high level, we approach the problem of mining source code idioms as that of inferring of commonly reoccurring fragments in (abstracted) ASTs. We apply recent advanced techniques from statistical NLP (Cohn et al., 2010; Post & Gildea, 2009), but we need to explain them in some detail to justify why they are appropriate for this software engineering task, and why simpler methods would not be effective.

We will build up step by step. First, we will describe our representation of idioms. In particular, we describe a family of probability distributions over ASTs which are called probabilistic tree substitution grammars (pTSGs). A pTSG is essentially a probabilistic context free grammar (PCFG) with the addition of special rules that insert a tree fragment all at once.

Second, we describe how we *discover* idioms. We do this by learning a pTSG that best explains a large quantity of existing source code. We consider as idioms the tree fragments that appear in the learned pTSG. We learn the pTSG using a powerful general framework called *nonparametric Bayesian methods*. Nonparametric Bayes provides a principled theoretical framework for automatically inferring how complex a model

⁴Holistic, Automatic Gathering of Grammatical Idioms from Software.

should be from data. Every time we add a new fragment rule to the pTSG, we are adding a new parameter to the model (the rule's probability of appearing), and the number of potential fragments that we could add is infinite. This creates a risk that by adding a large number of fragments we could construct a model with too many parameters, which would be likely to overfit the training data. Nonparametric Bayesian methods provide a way to tradeoff the model's fit to the training set with the model's size when the maximum size of the model is unbounded.

It is also worth explaining why we employ *probabilistic* models here, rather than a standard deterministic CFG. Probabilities provide a natural quantitative measure of the quality of a proposed idiom: A proposed idiom is worthwhile only if, when we include it into a pTSG, it increases the probability that the pTSG assigns to the training corpus. This encourages the method to avoid identifying idioms that are frequent but boring.

At first, it may seem odd that we apply grammar learning methods here, when of course the grammar of the programming language is already known. We clarify that our aim is *not* to re-learn the known grammar, but rather to learn probability distributions over trees from the known grammar. These distributions will represent which rules from the grammar are used more often, and, crucially, which sets of rules tend to be used contiguously.

7.2.1 Probabilistic Grammars

A *probabilistic context free grammar* (PCFG) is a simple way to define a distribution over the strings of a context-free language. A PCFG is defined as $G = (\Sigma, N, S, R, \Pi)$, where Σ is a set of terminal symbols, N a set of nonterminals, $S \in N$ is the root nonterminal symbol and R is a set of productions. Each production in R has the form $X \rightarrow Y$, where $X \in N$ and $Y \in (\Sigma \cup N)^*$, where $*$ denotes the Kleene star of the set. The set Π is a set of distributions $P(r|c)$, where $c \in N$ is a non-terminal, and $r \in R$ is a rule with c on its left-hand side. To sample a tree from a PCFG, we recursively expand the tree, beginning at S , and each time we add a non-terminal c to the tree, we expand c using a production r that is sampled from the corresponding distribution $P(r|c)$. The probability of generating a particular tree T from this procedure is the product over all rules that are required to generate T . The probability $P(x)$ of a string $x \in \Sigma^*$ is the sum of the probabilities of the trees T that yield x , that is, we simply consider $P(x)$ as a marginal distribution of $P(T)$.

Tree Substitution Grammars A tree substitution grammar (TSG) (Joshi & Schabes,

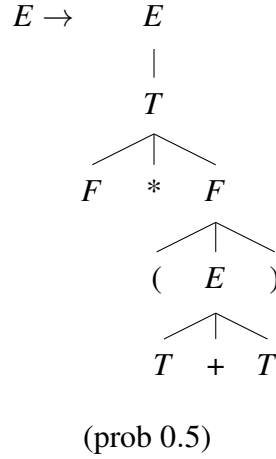


Figure 7.3: An example of a pTSG rule for a simple expression grammar. See text for more details.

1997; Bod et al.) is a simple extension to a CFG, in which productions expand into tree fragments rather than simply into a list of symbols. Formally, a TSG is also a tuple $G = (\Sigma, N, S, R)$, where Σ, N, S are exactly as in a CFG, but now each production $r \in R$ takes the form $X \rightarrow \mathcal{T}_X$, where \mathcal{T}_X is a fragment. To produce a string from a TSG, we begin with a tree containing only S , and recursively expand the tree in a manner exactly analogous to a CFG — the only difference is that some rules can increase the height of the tree by more than 1. A probabilistic tree substitution grammar (pTSG) G (Cohn et al., 2010; Post & Gildea, 2009) augments a TSG with probabilities, in an analogous way to a PCFG. A pTSG is defined as $G = (\Sigma, N, S, R, \Pi)$ where Σ is a set of terminal symbols, N a set of non terminal symbols, $S \in N$ is the root non-terminal symbol, R is a set of tree fragment productions. Finally, Π is a set of distributions $P_{TSG}(\mathcal{T}_X|X)$, for all $X \in N$, each of which is a distribution over the set of all rules $X \rightarrow \mathcal{T}_X$ in R that have left-hand side X . Note that TSGs and PCFGs are similar with the difference that the productions in R expand to *tree fragments* that may or may not contain terminal symbols. Thus, a well-formed TSG contains frequently reoccurring tree fragments. Although TSGs are a full language model, in this chapter we will only take advantage of their ability to learn in a probabilistically principled way common tree fragments.

The key reason that we use pTSGs for idiom mining is that each tree fragment \mathcal{T}_X can be thought of as describing a set of context-free rules that are typically used in sequence. This is exactly what we are trying to discover in the idiom mining problem. In other words, *our goal will be to induce a pTSG in which every tree fragment represents*

a *code idiom* if the fragment has depth greater than 1, or a rule from the language's original grammar if the depth equals 1. As a simple example, consider the PCFG

$$\begin{array}{ll} E \rightarrow T + T & (\text{prob } 0.7) \\ E \rightarrow T & (\text{prob } 0.3) \\ F \rightarrow (E) & (\text{prob } 0.1) \end{array} \quad \begin{array}{ll} T \rightarrow F * F & (\text{prob } 0.6) \\ T \rightarrow F & (\text{prob } 0.4) \\ F \rightarrow id & (\text{prob } 0.9), \end{array}$$

where E , T , and F are non-terminals, and E the start symbol. Now, suppose that we are presented with a corpus of strings from this language that include many instances of expressions like $id * (id + id)$ and $id * (id + (id + id))$ (perhaps generated by a group of students who are practicing the distributive law). Then, we might choose to add a single pTSG rule to this grammar, displayed in Figure 7.3, adjusting the probabilities for that rule and the $E \rightarrow T + T$ and $E \rightarrow T$ rules so that the three probabilities sum to 1. Essentially, this allows us to represent a correlation between the rules $E \rightarrow T + T$ and $T \rightarrow F * F$.

Finally, note that every CFG can be written as a TSG where all productions expand to trees of depth 1. Conversely, every TSG can be converted into an equivalent CFG by adding extra non-terminals (one for each TSG rule $X \rightarrow \mathcal{T}_X$). So TSGs are, in some sense, fancy notation for CFGs. This notation will prove very useful, however, when we describe the learning problem next.

7.2.2 Learning TSGs

Now we define the learning problem for pTSGs that we will consider. First, we say that a pTSG $G_1 = (\Sigma_1, N_1, S_1, R_1, P_1)$ *extends* a CFG G_0 if every tree with positive probability under G_1 is grammatically valid according to G_0 . Given any set \mathcal{T} of tree fragments from G_0 , we can define a pTSG G_1 that extends G_0 as follows. First, set $(\Sigma_1, N_1, S_1) = (\Sigma_0, N_0, S_0)$. Then, set $R_1 = R_{CFG} \cup R_{FRAG}$, where R_{CFG} is the set of all rules from R_0 , expressed in the TSG form, *i.e.* with right-hand sides as trees of depth 1, and R_{FRAG} is a set of *fragment rules* $X_i \rightarrow \mathcal{T}_i$, for all $\mathcal{T}_i \in \mathcal{T}$ and where X_i is the root of \mathcal{T}_i .

The grammar learning problem that we consider can be called the *CFG extension problem*. The input is a set of trees $T_1 \dots T_N$ from a context-free grammar $G_0 = (\Sigma_0, N_0, S_0, R_0)$. The CFG extension problem is to learn a pTSG G_1 that extends G_0 and is good at explaining the training set $T_1 \dots T_N$. The notion of “good” is deliberately vague; formalizing it is part of the problem. It should also be clear that we *are not* trying

to learn the CFG for the original programming language — instead, we are trying to identify sequences of CFG rules that commonly co-occur contiguously.

7.2.2.1 Why Not Just Count Common Trees?

A natural first approach to the CFG extension problem is to mine frequent patterns, for example, to return the set of all AST fragments that occur more than a user-specified parameter M times in the training set. This task is called frequent tree mining, and has been the subject of some work in the data mining literature (Jiménez et al., 2010; Termier et al., 2002; Zaki, 2002, 2005). Unfortunately, preliminary investigation by Kuzborskij (2011) found that these approaches do not yield good idioms, suffering from known problems observed with frequent pattern mining in the data mining literature (Aggarwal & Han, 2014, Chapter 4). Instead, the fragments that are returned tend to be small and generic, omitting many details that, to a human eye, are central to the idiom. For example, given the idiom in Figure 7.1(c), it would be typical for tree mining methods to return a fragment containing the `try`, `if`, and `finally` nodes but not the crucial method call to `Cursor.close()`.

The reason for this is simple: Given a fragment \mathcal{T} that represents a true idiom, it can always be made more frequent by removing one of the leaves, even if that leaf co-occurs often with the rest of the tree. So tree mining algorithms tend to return these shorter trees, resulting in incomplete idioms. This is a general problem with frequent pattern mining: *frequent patterns can be “boring” patterns* (Aggarwal & Han, 2014, Chapter 5). To avoid this problem, we need to penalize the method when it chooses *not* to extend a pattern to include a node that co-occurs frequently. This is what is provided by our probabilistic approach.

A different idea is to use the *maximum likelihood principle*, that is, to find the pTSG G_1 that extends G_0 and maximizes the probability that G_1 assigns to $T_1 \dots T_N$. This also does not work. The reason is that a trivial solution is simply to add a fragment rule $E \rightarrow T_i$ for every training tree T_i . This will assign a probability of $1/N$ to each training tree, which in practice will often be optimal. What is going on here is that the maximum likelihood grammar is overfitting. It is not surprising that this happens: there are an infinite number of potential trees that could be used to extend G_0 , so if a model is given such a large amount of flexibility, overfitting becomes inevitable. What we need is a strong method of controlling overfitting, which the next section provides.

7.2.2.2 Nonparametric Bayesian Methods

At the heart of any machine learning application is the need to control the complexity of the model. For example, in a clustering task — an unsupervised task in which the object is to partition a dataset into a set of groups, called *clusters*, that are intuitively similar — many clustering methods, such as K -means, require the user to specify the number of clusters K in advance. If K is too small, then each cluster will be very large and not contain useful information about the data. If K is too large, then each cluster will only contain a few data points, so again, the cluster centroid will not tell us much about the dataset. For the *CFG extension problem*, the key factor that determines model complexity is the number of fragment rules that we allow for each non-terminal. If we allow the model to assign too many fragments or fragments that are too large to each non-terminal, then it can simply memorize the training set. But if we allow too few, then the model will be unable to find useful patterns. Nonparametric Bayesian methods provide a powerful and theoretically principled method for managing this trade-off. Although powerful, these methods can be difficult to understand at first. We will not give a detailed tutorial due to space; for a gentle introduction, see [Gershman & Blei \(2012\)](#).

To begin, we must first explain Bayesian statistics. Bayesian statistics ([Gelman et al., 2013](#); [Murphy, 2012](#)) is an alternative general framework to classical frequentist statistical methods, such as confidence intervals and hypothesis testing, that allows the analyst to encode prior knowledge about the quantity of interest. The idea behind Bayesian statistics is that whenever one wants to estimate an unknown parameter θ from a dataset x_1, x_2, \dots, x_N , the analyst should not only treat the data $x_1 \dots x_N$ as random variables — as in classical statistics — but also θ as well. To do this, the analyst must choose a prior distribution $P(\theta)$ that encodes any prior knowledge about θ (if little is known, this distribution can be vague, *e.g.* uniform), and then a likelihood $P(x_1 \dots x_N | \theta)$ that describes a model of how the data is generated given θ . To be clear, the prior and the likelihood are mathematical models of the data, that is, they are mathematical approximations to reality that are designed by the data analyst. Some models are better approximations than others, and more accurate models will yield more accurate inferences about θ .

Once we define a prior and a likelihood, the laws of probability provide only one choice for how to infer θ , namely, via the conditional distribution $P(\theta | x_1 \dots x_N)$ which is uniquely defined by Bayes' rule. This distribution is called the *posterior distribution* and encapsulates all of the information that we have about θ from the data. We can

compute summaries of the posterior to make inferences about θ , for example, if we want to estimate θ by a single vector, we might compute the mean of $P(\theta|x_1 \dots x_N)$. Although mathematically the posterior distribution is a simple function of the prior and likelihood, in practice it can be very difficult to compute, and approximations are often necessary. To summarize, applications of Bayesian statistics have three steps: first, choose a prior $p(\theta)$; second, choose a likelihood $p(x_1 \dots x_N | \theta)$, finally, compute $p(\theta|x_1 \dots x_N)$ using Bayes's rule.

As a simple example, suppose the data $x_1 \dots x_N$ are real numbers, which we believe to be distributed independently according a Gaussian distribution with variance 1 but unknown mean θ . Then we might choose a prior $p(\theta)$ to be Gaussian with mean 0 and a large variance, to represent the fact that we do not know much about θ before we see the data. Our beliefs about the data indicate that $p(x_i|\theta)$ is Gaussian with mean θ and variance 1. By applying Bayes's rule, it is easy to show that $P(\theta|x_1 \dots x_N)$ is also Gaussian, whose mean is approximately⁵ equal to $N^{-1} \sum_i x_i$ and whose variance is approximately $1/N$. This distribution represents a Bayesian's belief about the unknown mean θ , after seeing the data.

Nonparametric Bayesian methods handle the more complex case where the *number* of parameters is unknown as well. For example, consider a clustering model where, conditioned on the cluster identity, the data is Gaussian, but the number of clusters is unknown. In this case, θ is a vector containing the centroid for each cluster, but then, because before we see the data the number of clusters could be arbitrarily large, θ has unbounded dimension. Nonparametric Bayesian methods focus on developing prior distributions over such infinite dimensional objects, which are then used within Bayesian statistical inference. Bayesian nonparametrics have been the subject of intense research in statistics and machine learning, with popular models including the Dirichlet process (Hjort, 2010) and the Gaussian process (Williams & Rasmussen, 2006).

Applying this discussion to the CFG extension problem, what we are trying to infer is a pTSG T . So, to apply Bayesian inference, our prior distribution must be a *probability distribution over probabilistic grammars*. In order to define this distribution, we will need to take a brief digression and define first a distribution $P_0(T)$ over *fragments* from a CFG. Let G_0 be the known CFG for the programming language in question. We will assume that we have available a PCFG for G_0 , because this can be easily estimated by maximum likelihood from a training corpus; call this distribution P_{ML} . Now, P_{ML} gives us a distribution over full trees. To get a distribution over fragments, we include

⁵The exact value depends on precisely what variance we choose in $p(\theta)$, but the formula is simple.

a distribution over tree sizes, yielding

$$P_0(T) = P_{\text{geom}}(|T|, p_{\$}) \prod_{r \in T} P_{\text{ML}}(r), \quad (7.2)$$

where $|T|$ is the size of the fragment T , P_{geom} is a geometric distribution with parameter $p_{\$}$, and r ranges over the multiset of productions that are used within T .

Now we can define a prior distribution over pTSGs. Recall that we can define a pTSG G_1 that extends G_0 by specifying a set of tree fragments \mathcal{F}_X for each non-terminal X . So, to define a distribution over pTSGs, we will define a distribution $P(\mathcal{F}_X)$ over the set of tree fragments rooted at X . We need $P(\mathcal{F}_X)$ to have several important properties. First, we need $P(\mathcal{F}_X)$ to have infinite support, that is, it must assign positive probability to *all possible fragments*. This is because if we do not assign a fragment positive probability in the prior distribution, we will never be able to infer it as an idiom, no matter how often it appears. Second, we want $P(\mathcal{F}_X)$ to exhibit a “rich-get-richer” effect, namely, once we have observed that a fragment \mathcal{T}_X occurs many times, we want to be able to predict that it will occur more often in the future.

A natural distribution with these properties is the Dirichlet process (DP). The Dirichlet process has two parameters: a *base measure*,⁶ in our case, the fragment distribution P_0 , and a concentration parameter $\alpha \in \mathbb{R}^+$, which controls the strength of the rich-get-richer effect. Following the *stick-breaking* representation (Sethuraman, 1991), a Dirichlet process defines a prior distribution over \mathcal{F}_X as

$$P(\mathcal{T} \in \mathcal{F}_X) = \sum_{k=1}^{\infty} \pi_k \delta_{\{\mathcal{T}=\mathcal{T}_k\}} \quad \mathcal{T}_k \sim P_0 \quad (7.3)$$

$$\pi_k = u_k \prod_{j=1}^{k-1} (1 - u_j) \quad u_k \sim \text{Beta}(1, \alpha). \quad (7.4)$$

To interpret this, recall that the symbol \sim is read “is distributed as,” the Beta distribution is a standard distribution over the set $[0, 1]$, and $\delta_{\{\mathcal{T}=\mathcal{T}_k\}}$ is a delta function, *i.e.*, a probability distribution over \mathcal{T} that generates \mathcal{T}_k with probability 1. Note, that as we will discuss next, mining does not directly implement this equation, but only its posterior and avoids the infinite summation. However, these equations are illustrative of how a DP works: Intuitively, what is going on in a DP is that a sample from the DP is a distribution over a countably infinite number of fragments $\mathcal{T}_1, \mathcal{T}_2, \dots$. Each one of these fragments is sampled independently from the fragment distribution P_0 . To assign a probability to each fragment, we recursively split the interval $[0, 1]$ into a countable number of

⁶The base measure will be a probability measure, so for our purposes, we can think of this as a fancy word for “base distribution”.

sticks π_1, π_2, \dots . The value $(1 - u_k)$ defines what proportion of the remaining stick is assigned to the current sample \mathcal{T}_k , and the remainder is assigned to the infinite number of remaining trees $\mathcal{T}_{k+1}, \mathcal{T}_{k+2}, \dots$. This process defines a distribution over fragments \mathcal{F}_X for each non-terminal X , and hence a distribution $P(G_1)$ over the set of all pTSGs that extend G_0 . We will refer to this distribution as a *Dirichlet process probabilistic tree substitution grammar* (DPpTSG) (Post & Gildea, 2009; Cohn et al., 2010).

This process may seem odd for two reasons: (a) each sample from $P(G_1)$ is infinitely large, so we cannot store it exactly on a computer, (b) the fragments from G_1 are sampled randomly from a PCFG, so there is no reason to think that they should match real idioms. Fortunately, the answer to both these concerns is simple. We are *not* interested in the fragments that exist in the prior distribution, but rather of those in the posterior distribution. More formally, the DP provides us with a prior distribution G_1 over pTSGs. But G_1 itself, like any pTSG, defines a distribution $P(T_1, T_2, \dots, T_N | G_1)$ over the training set. So, just as in the parametric case, we can apply Bayes's rule to obtain a posterior distribution $P(G_1 | T_1, T_2, \dots, T_N)$. It can be shown that this distribution is also a DPpTSG, and, amazingly, that this posterior DPpTSG can be characterized by a *finite* set of fragments \mathcal{F}'_X for each non-terminal. It is these fragments that we will identify as code idioms (Section 7.3).

7.2.2.3 Inference

Now that we have defined a posterior distribution over probabilistic grammars, we need to describe how to *compute* this distribution. Unfortunately, the posterior distribution cannot be computed exactly, so we resort to approximations. The most commonly used approximations in the literature are based on Markov chain Monte Carlo (MCMC), which we explain below. The high-level idea behind MCMC is to define a Markov chain that has the property that if we run the Markov chain for long enough, samples from the chain will eventually be approximately distributed according to the intractable distribution that we care about. But first, we make one more observation about pTSGs. All of the pTSGs that we consider are extensions of an unambiguous base CFG G_0 . This means that given a source code file or snippet \mathbb{c} , we can separate the pTSG parsing task into two steps: first, parse \mathbb{c} using f (which is defined by G_0), resulting in a CFG tree T ; second, group the nodes in T according to which fragment rule in the pTSG was used to generate them. We can represent this second task as a tree of binary variables z_s for each node s . These variables indicate whether s is the root of a new fragment ($z_s = 1$), or if s is part of the same fragment as its parent ($z_s = 0$). Essentially, the variables z_s show

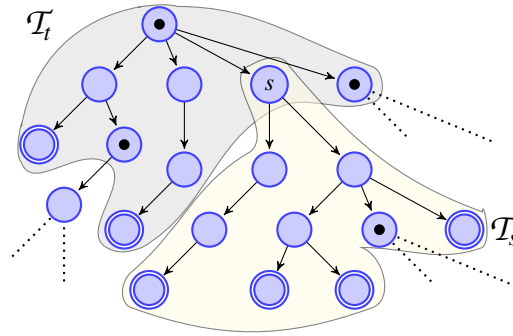


Figure 7.4: Sampling a pTSG from a tree T . Dots within the node circles show the points where the tree is split (*i.e.* $z_t = 1$). Terminal nodes have double-line border. The figure shows two tree fragments, each shaded with a different color (\mathcal{T}_t is the top blue-shaded tree fragment and \mathcal{T}_s is the lower yellow-shaded tree fragment). At some point during inference node s is considered. At that point, z_s is set to 0 or 1 with probability $P(z_s = 0)$ using Equation 7.5. If z_s is sampled to have a value of 1, then \mathcal{T}_t and \mathcal{T}_s remain separate and exist as two separate rules in the pTSG. Otherwise ($z_s = 0$) \mathcal{T}_t and \mathcal{T}_s are joined into a single elementary tree that spans the whole blue and yellow-shaded regions. This process is performed iteratively on all nodes multiple times. After a few burn-in iterations, at each iteration, we retrieve a sample from the posterior pTSG (each sample is a pTSG) that explains how each tree could have been generated from the real posterior pTSG.

the boundaries of the inferred tree patterns; see Figure 7.4 for an example. Conversely, even if we don't know what fragments are in the grammar, given a training corpus that has been parsed in this way, we can use the z_s variables to read off what fragments must have been in the pTSG.

With this representation in hand, we are now ready to present an MCMC method for sampling from the posterior distribution over grammars, using a particular method called Gibbs sampling. Gibbs sampling is an iterative method, which starts with an initial value for all of the z variables, and then updates them one at a time. At each iteration, the sampler visits every tree node t of every tree in the training corpus, and samples a new value for z_t . Let s be the root of the fragment where t belongs. If we choose $z_t = 1$, we can examine the current values of the z variables to determine the tree fragment \mathcal{T}_t that contains t and the fragment \mathcal{T}_s for s , which must be disjoint. On the other hand, if we set $z_t = 0$, then s and t will belong to the same fragment, which will be exactly $\mathcal{T}_{\text{join}} = \mathcal{T}_s \cup \mathcal{T}_t$. Now, we set z_t to 0 with probability

$$P_{\text{post}}(z_t = 0) = \frac{P_{\text{post}}(\mathcal{T}_{\text{join}})}{P_{\text{post}}(\mathcal{T}_{\text{join}}) + P_{\text{post}}(\mathcal{T}_s)P_{\text{post}}(\mathcal{T}_t)}. \quad (7.5)$$

where

$$P_{\text{post}}(\mathcal{T}) = \frac{\text{count}(\mathcal{T}) + \alpha P_0(\mathcal{T})}{\text{count}(h(\mathcal{T})) + \alpha}, \quad (7.6)$$

h returns the root of the fragment, and count returns the number of times that a tree occurs as a fragment in the corpus, as determined by the current values of z . Intuitively, what is happening here is that if the fragments \mathcal{T}_s and \mathcal{T}_t occur very often together in the corpus, relative to the number of times that they occur independently, then we are more likely to join them into a single fragment.

It can be shown that if we repeat this process for a large number of iterations, eventually the resulting distribution over fragments at the end of each iteration will converge to the posterior distribution over fragments defined by the DPpTSG. It is these fragments that we return as idioms.

We present the Gibbs sampler because it is a useful illustration of MCMC, but in practice we find that it converges too slowly to scale to large codebases. Instead we use the type-based MCMC sampler of [Liang et al. \(2010b\)](#), which samples multiple node states at each time step. In specific, type-based MCMC is a form of block Gibbs sampling where each block is the set of all nodes whose upper and lower trees are identical. For example, in Figure 7.4 z_s of node s will be considered jointly with all other nodes whose upper and lower tree fragments are identical to \mathcal{T}_t and \mathcal{T}_s , *i.e.* the

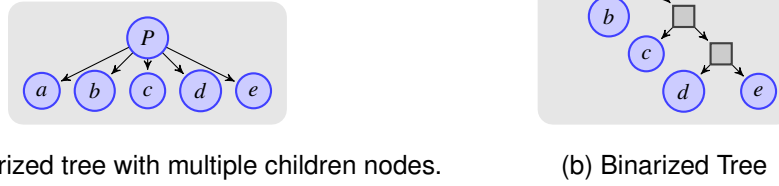


Figure 7.5: Tree binarization (Markovization) for nodes with multiple children. Square nodes represent the dummy nodes added. The goal of this process is to transform nodes with arbitrary number of children to trees with a fixed number of children. This is a common trick for reducing data sparsity in NLP. For code idiom mining, this is useful for AST nodes that represent multiple sequential statements and method arguments.

trees have exactly the same structure and node types. This process tends to yield faster mixing of the MCMC chain and avoids getting stuck in local optima during sampling (Liang et al., 2010b).

7.3 Mining Syntactic Idioms

In this section, we describe a set of necessary transformations to ASTs and pTSGs to adapt these general methods specifically to the task of inferring syntactic code idioms.

AST Transformation As discussed in Section 7.1, to mine syntactic idioms from a code snippet \mathbb{c} we transform it into a tree T using a $f : \mathbb{c} \rightarrow T$. This paragraph describes f_{syn} that is specific to syntactic code idiom mining. For each .java file we use the Eclipse JDT (Eclipse-Contributors, 2014) to extract its AST — a tree structure of `ASTNode` objects. Each `ASTNode` object contains two sets of properties: *simple properties* — e.g. the type of the operator for an infix expression `ASTNode` — and *structural properties* that contain zero or more child `ASTNode` objects. First, we construct the grammar symbols by mapping each `ASTNode`’s type and simple properties into a single (terminal or non-terminal) symbol. The transformed tree is then constructed by mapping the original AST into a tree whose nodes are annotated with the symbols. Each node’s children are grouped by property.

The transformed trees may contain nodes that have more than two children for a single property (e.g. `Block`). This induces unnecessary sparsity in the CFG and TSG rules. For example, `Block` node’s potential children include arbitrary permutations of one or more node types (e.g. assignment nodes, declaration nodes, `if` nodes, etc.), which

results in a very large and sparse set of combinations, even for relatively large amounts of data. This sparsity makes the learning problem harder. Since all such permutations are sparse, even simple PCFGs do not generalize well in this setting. To alleviate this issue, we perform *tree binarization*. This process — common in NLP for exactly the same reasons we outlined — transforms the original tree into a binary tree by adding dummy nodes (non-terminals), making the data less sparse (see Figure 7.5). It also helps us capture idioms in subsequences of sequential statements. Note that binarization is performed per structural property *only* when it contains more than two children, while a node will generally have more than two children across all its structural properties.

One final hurdle for learning meaningful code idioms are variable names. Since variable names are mostly project or class specific we abstract them introducing an intermediate `MetaVariable` node between the `SimpleName` node containing the string representation of the variable name and its parent node. `MetaVariable` nodes are also annotated with the type of the variable they are abstracting. This provides the pTSG with the flexibility to either exclude or include variable names as appropriate. For example, in the snippet of Figure 7.1(a) by using metavariables, we are able to learn the idiom in Figure 7.1(b) without specifying the name of the `Cursor` object by excluding the `SimpleName` nodes from the fragment. Alternatively, if a specific variable name is common and idiomatic, such as the `i` in a `for` loop, the pTSG can choose to include `SimpleName` in the extracted idiom, by merging it with its parent `MetaVariable` node.

Training TSGs and Extracting Code Idioms Training a pTSG happens offline, during a separate training phase. After training the pTSG, we then extract the mined syntactic code idioms which then can be used for any later visualization. In other words, a user of a HAGGIS IDE tool would never need to wait for an MCMC method to finish. The output of an MCMC method is a series of samples from the posterior distribution, each of which in our case, is a single pTSG. These sampled pTSGs need to be post-processed to extract a single, meaningful set of code idioms. First, we aggregate the MCMC samples after removing the first few samples as *burn-in*, which is standard methodology for applying MCMC. Then, to extract idioms from the remaining samples, we merge all samples' tree fragments into a single set that also keeps a count of the number of times each element is added (multiset). We prune this multiset by removing all tree fragments that have been seen less than c_{min} times. We also prune fragments that have fewer than n_{min} nodes to remove trivial idioms. Finally, we convert the remaining fragments back to Java code. The leaf nodes of the fragments that contain non-terminal

symbols represent metavariables and are converted to the appropriate symbol that is denoted by a \$ prefix.

Additionally, to assist the sampler in inducing meaningful idioms, we prune any `import` statements from the corpus, so that they cannot be mined as idioms. We also exclude some nodes from sampling, fixing $z_i = 0$ and thus forcing some nodes to be un-splittable. Such nodes include method invocation arguments, qualified and parameterized type node children, non-block children of `while`, `for` and `if` statement nodes, parenthesized, postfix and infix expressions and variable declaration statements.

7.4 Syntactic Idioms Evaluation

We take advantage of the omnipresence of idioms in source code to evaluate HAGGIS on popular open-source projects. We restrict ourselves to the Java programming language, due to the high availability of tools and source code. We emphasize, however, that HAGGIS is language agnostic. Before we get started, an interesting way to get an intuitive feel for any probabilistic model is simply to draw samples from it. Figure 7.6 shows a code snippet that we synthetically generated by sampling from the posterior distribution over code defined by the pTSG. One can observe that the pTSG — as a language model — is learning to produce idiomatic and syntactically correct code, although — as expected — the code is semantically inconsistent.

Methodology We use two evaluation datasets comprised of Java open-source code available on GitHub. The PROJECTS dataset (Table 7.1) contains the top 13 Java GitHub projects whose repository is at least 100MB in size according to the GitHub Archive⁷. To determine popularity, we computed the z-score of forks and watchers for each project. The normalized scores were then averaged to retrieve each project’s popularity ranking. The second evaluation dataset, LIBRARY (Table 7.2), consists of Java classes that import (*i.e.* use) 15 popular Java libraries. For each selected library, we retrieved from the Java GitHub Corpus (Allamanis & Sutton, 2013b) all files that import that library but do not implement it. We split both datasets into a train and a test set, splitting each project in PROJECTS and each library file set in LIBRARY into a train (70%) and a test (30%) set. The PROJECTS will be used to mine project-specific idioms, while the LIBRARY will be used to mine idioms that occur across libraries.

To extract idioms we run MCMC for 100 iterations for each of the projects in PROJECTS and each of the library file sets in LIBRARY, using the first 75 iterations as

⁷<https://www.githubarchive.org/>

```

1 try {
2   regions=computeProjections(owner);
3 } catch (RuntimeException e) {
4   e.printStackTrace();
5   throw e;
6 }
7 if (elem instanceof IParent) {
8   IJavaElement[] children=((IParent)owner).getChildren();
9   for (int fromPosition=0; i < children.length; i++) {
10    IJavaElement aChild=children[i];
11    Set childRegions=findAnnotations(aChild,result);
12    removeCollisions(regions,childRegions);
13  }
14 }

```

Figure 7.6: Synthetic code randomly generated from a posterior pTSG. The pTSG produces syntactically correct and locally consistent code. This effect allows us to infer code idioms. As expected, the pTSG cannot capture higher level information, such as variable binding.

burn-in. For the last 25 iterations, we aggregate a sample posterior pTSG and extract idioms as detailed in Section 7.3. As a final step, we remove any idioms that do *not* contain identifiers. A threat to the validity of the evaluation using the aforementioned datasets is the possibility that the datasets are not representative of Java development practices, containing solely open-source projects from GitHub. However, the selected datasets span a wide variety of domains, including databases, messaging systems and code parsers, diminishing any such possibility. Furthermore, we perform an extrinsic evaluation on source code found on a popular online Q&A website, StackOverflow.

Evaluation Metrics We compute two metrics on the test corpora. These metrics resemble precision and recall in information retrieval but are adjusted to the code idiom domain. We define *idiom coverage* as the percent of source code AST nodes that matches any of the mined idioms. Coverage is thus a number between 0 and 1 indicating the extent to which the mined idioms exist in a piece of code. We define *idiom set precision* as the percentage of the mined idioms that also appear in the test corpus. Using these two metrics, we tune the concentration parameter of the DPpTSG model by using `android.net.wifi` as a validation set, yielding $\alpha = 1$.

<pre> 1 channel=connection. 2 createChannel(); </pre>	<pre> 1 Elements \$name=\$(Element). 2 select(\$StringLit); </pre>
(a) An com.rabbitmq idiom	(b) An org.jsoup idiom
<pre> 1 Transaction tx=ConnectionFactory. 2 getDatabase().beginTx(); </pre>	<pre> 1 catch (Exception e){ 2 \$(Transaction).failure(); 3 } </pre>
(c) An org.neo4j idiom	(d) An org.neo4j idiom
<pre> 1 SearchSourceBuilder builder= 2 getQueryTranslator().build(3 \$(ContentIndexQuery)); </pre>	<pre> 1 LocationManager \$name = 2 (LocationManager)getSystemService(3 Context.LOCATION_SERVICE); </pre>
(e) An org.elasticsearch idiom	(f) An android.location idiom
<pre> 1 Location.distanceBetween(2 \$(Location).getLatitude(), 3 \$(Location).getLongitude(), 4 \$....); </pre>	<pre> 1 try{ 2 \$BODY\$ 3 }finally{ 4 \$(RevWalk).release(); 5 } </pre>
(g) An android.location idiom	(h) An org.eclipse.jgit idiom
<pre> 1 try{ 2 Node \$name=\$methodInvoc(); 3 \$BODY\$ 4 }finally{ 5 \$(Transaction).finish(); 6 } </pre>	<pre> 1 ConnectionFactory factory = 2 new ConnectionFactory(); 3 \$methodInvoc(); 4 Connection connection = 5 factory.newConnection(); </pre>
(i) An org.neo4j idiom	(j) An io.netty idiom

Figure 7.7: Top cross-project syntactic idioms for LIBRARY projects (Table 7.1). Here we include idioms that appear in the test set files. We rank them by the number of distinct files they appear in and restrict into presenting idioms that contain at least one library-specific (*i.e.* API-specific) identifier. The special notation `$(TypeName)` denotes the presence of a variable of type `TypeName` whose name is undefined. `$BODY$` denotes a user-defined code block of one or more statements, `$name` a freely defined (variable) name, `$methodInvoc` a single method invocation statement and `$ifstatement` a single `if` statement. All the idioms have been automatically identified by HAGGIS. More samples in Figure 7.8.

```

1 while ($(ModelNode) != null){
2   if ($(ModelNode) == limit)
3     break;
4   $ifstatement
5   $(ModelNode)=$(ModelNode)
6     .getParentModelNode();
7 }

```

(a) An org.mozilla.javascript idiom

```

1 if ($(Connection) != null){
2   try{
3     $(Connection).close();
4   }catch (Exception ignore){}
5 }

```

(c) An io.netty idiom

```

1 try{
2   Session session
3     =HibernateUtil
4       .currentSession();
5   $BODY$
6 }catch (HibernateException e){
7   throw new DaoException(e);
8 }

```

(e) An org.hibernate idiom

```

1 FileSystem $name
2   =FileSystem.get(
3     $(Path).toUri(),conf);

```

(g) An org.apache.hadoop idiom

```

1 Toast.makeText(this,
2   $stringLiteral,Toast.LENGTH_SHORT)
3   .show()

```

(i) An android idiom

```

1 Document doc=Jsoup.connect(URL).
2   userAgent("Mozilla").
3   header("Accept","text/html").
4   get();

```

(b) An org.jsoup idiom

```

1 Traverser traverser
2   =$(Node).traverse();
3 for (Node $name : traverser){
4   $BODY$
5 }

```

(d) An org.neo4j idiom

```

1 catch (HibernateException e) {
2   if ($(Transaction) != null) {
3     $(Transaction).rollback();
4   }
5   e.printStackTrace();
6 }

```

(f) An org.hibernate idiom

```

1 (token=$(XContentParser)
2   .nextToken())
3   != XContentParser
4     .Token.END_OBJECT

```

(h) An org.apache.lucene idiom

Figure 7.8: Top cross-project syntactic idioms for LIBRARY projects (Table 7.1). Continued from Figure 7.7.

Table 7.1: PROJECTS dataset used for in-project idiom evaluation. Projects in alphabetical order.

Name	Forks	Stars	Files	Commit	Description
arduino	2633	1533	180	2757691	Electronics Prototyping
atmosphere	1606	370	328	a0262bf	WebSocket Framework
bigbluebutton	1018	1761	760	e3b6172	Web Conferencing
elasticsearch	5972	1534	3525	ad547eb	REST Search Engine
grails-core	936	492	831	15f9114	Web App Framework
hadoop	756	742	4985	f68ca74	Map-Reduce Framework
hibernate	870	643	6273	d28447e	ORM Framework
libgdx	2903	2342	1985	0c6a387	Game Dev Framework
netty	2639	1090	1031	3f53ba2	Net App Framework
storm	1534	7928	448	cdb116e	Distributed Computation
vert.x	2739	527	383	9f79416	Application platform
voldemort	347	1230	936	9ea2e95	NoSQL Database
wildfly	1060	1040	8157	043d7d5	Application Server

7.4.1 Top Idioms

Figure 7.7 and Figure 7.8 shows the top idioms mined in the LIBRARY dataset, ranked by the number of files in the test sets where each idiom has appeared in. The reader will observe their immediate usefulness. Some idioms capture how to retrieve or instantiate an object. For example, in Figure 7.7, the idiom 7.7a captures the instantiation of a message channel in RabbitMQ, 7.8g retrieves a handle for the Hadoop file system, 7.7e builds a SearchSourceBuilder in Elasticsearch and 7.8b retrieves a document at a given address using JSoup. Other idioms capture important transactional properties of code: idiom 7.7h demonstrates proper use of the memory-hungry RevWalk object in JGit and 7.7i is a transaction idiom in Neo4J. Other idioms capture common error handling, such as 7.7d for Neo4J and 7.8e for a Hibernate transaction. Finally, some idioms capture common operations, such as closing a connection in Netty (7.8c), traversing through the database nodes (7.8d), visiting all AST nodes in a JavaScript file in Rhino (7.8a) and computing the distance between two locations (7.7g) in Android. The reader may observe that these idioms provide a meaningful set of coding patterns for each library,

Table 7.2: LIBRARY dataset for cross-project idiom evaluation. Each API file set contains all class files that `import` a class belonging to the respective package or one of its subpackages.

Package Name	Files	Description
<code>android.location</code>	1262	Android location API
<code>android.net.wifi</code>	373	Android WiFi API
<code>com.rabbitmq</code>	242	Messaging system
<code>com.spatial4j</code>	65	Geospatial library
<code>io.netty</code>	65	Network app framework
<code>opennlp</code>	202	NLP tools
<code>org.apache.hadoop</code>	8467	Map-Reduce framework
<code>org.apache.lucene</code>	4595	Search Server
<code>org.elasticsearch</code>	338	REST Search Engine
<code>org.eclipse.jgit</code>	1350	Git implementation
<code>org.hibernate</code>	7822	Persistence framework
<code>org.jsoup</code>	335	HTML parser
<code>org.mozilla.javascript</code>	1002	JavaScript implementation
<code>org.neo4j</code>	1294	Graph database
<code>twitter4j</code>	454	Twitter API

capturing semantically consistent actions that a developer is likely to need when using these libraries. However, as we will see in Section 7.5 these idioms are too detailed to represent common semantic operations and, thus, useful to software tool designers.

In Figure 7.9 we present a small set of general Java syntactic idioms mined across all datasets by HAGGIS. These idioms represent frequently used patterns that could be included by default in tools such as Eclipse’s SnipMatch ([Recommenders, 2014](#)) and IntelliJ’s live templates ([JetBrains, 2014](#)). These include idioms for defining constants (Figure 7.9c), creating loggers (Figure 7.9b) and iterating through an iterable (Figure 7.9a).

We now quantitatively evaluate the mined idiom sets. Table 7.3 shows idiom coverage, idiom set precision and the average size of the matched idioms in the test sets of each dataset. We observe that HAGGIS achieves better precision and coverage in PROJECTS than LIBRARY. This is expected since code idioms recur more often within

Table 7.4: Extrinsic evaluation of mined idioms from LIBRARY.

Test Corpus	Coverage	Precision
StackOverflow	31%	67%
PROJECTS	22%	50%

a project than across disparate projects. This effect may be partially attributed to the small number of people working in a project and partially to project-specific idioms. Table 7.3 also gives an indication of the trade-offs we can achieve for different c_{min} and n_{min} .

7.4.2 Code Cloning vs. Code Idioms

Previously, we argued that syntactic code idioms differ significantly from code clones. We now show this by using a cutting-edge clone detection tool: DECKARD (Jiang et al., 2007) is a state-of-the-art tree-based clone-detection tool that uses an intermediate vector representation to detect similarities. To extract code idioms from the code clone clusters that DECKARD computes, we retrieve the maximal common subtree of each cluster, ignoring patterns that are less than 50% of the original size of the tree.

We run DECKARD on the validation set with multiple parameters ($\text{stride} \in \{0, 2\}$, $\text{similarity} \in \{0.95, 1.0\}$, $\text{minToks} \in \{10, 20\}$) and picked those that achieve the best combination of precision and coverage. These parameters would be plausible choices if one would try to mine idioms with a clone detection tool. Table 7.3 shows precision, coverage and average idiom size (in number of nodes) of the patterns found through DECKARD and HAGGIS. HAGGIS found larger and higher coverage idioms, since clones seldom recur across projects. The differences in precision and coverage are statistically significant (paired t -test; $p < 0.001$). We note that the overlap in the patterns extracted by DECKARD and HAGGIS is small ($< 0.5\%$).

These results are not a criticism of DECKARD — which is a high-quality, state-of-the-art code clone detection tool — but rather show that *the task of code clone detection is different from code idiom mining*. Code clone detection — even when searching for gapped clones — is concerned with finding pieces of code that are not necessarily frequent but are maximally identical. In contrast, idiom mining is concerned with finding very common tree fragments that trade off between pattern size and frequency.

7.4.3 Extrinsic Evaluation of Mined Syntactic Idioms

Now, we evaluate HAGGIS extrinsically on a dataset of StackOverflow questions (Bacchelli, 2013). StackOverflow is a popular Q&A site for programming-related questions. The questions and answers often contain code snippets, which are representative of general development practice and are usually short, concise and idiomatic, containing only essential pieces of code. Our hypothesis is that snippets from StackOverflow are more idiomatic than typical code, so if HAGGIS idioms are meaningful, they will occur more commonly in code snippets from StackOverflow than in typical code.

To test this, we first extract all code fragments in questions and answers tagged as java or android, filtering only those that can be fully parsed by Eclipse JDT (Eclipse-Contributors, 2014). We further remove snippets that contain less than 5 tokens. After this process, we have 108,407 partial Java snippets. Then, we create a single set of idioms, merging all those found in LIBRARY and removing any idioms that have been seen in less than five files in the LIBRARY test set. We end up with small but high precision set of idioms across all APIs in LIBRARY.

Table 7.4 shows precision and coverage of HAGGIS’s idioms comparing StackOverflow, LIBRARY and PROJECTS. Using the LIBRARY idioms, we achieve a coverage of 31% and a precision of 67% on StackOverflow, compared to a much smaller precision and coverage in PROJECTS. This shows that the mined idioms are more frequent in StackOverflow than in a “random” set of projects. Since we expected StackOverflow snippets to be more highly idiomatic than average projects’ source code, this provides strong indication that HAGGIS has mined a set of meaningful syntactic idioms. We note that precision depends highly on the popularity of LIBRARY’s libraries. For example, because Android is one of the most popular topics in StackOverflow, when we limit the mined idioms to those found in the two Android libraries, HAGGIS achieves a precision of 96.6% at a coverage of 21% in StackOverflow. This indicates that HAGGIS idioms are widely used in practice.

Eclipse SnipMatch To further evaluate HAGGIS, we submitted a set of idioms to Eclipse SnipMatch (Recommenders, 2014). SnipMatch currently contains about 100 human-created code snippets. Currently only JRE, SWT and Eclipse specific snippets are being accepted. Upon discussion with the community, we mined a set of idioms specifically for SWT, JRE and Eclipse. Some of the HAGGIS mined idioms already existed in SnipMatch. Of the remaining idioms, we manually translated 27 idioms into JFace templates, added a description and submitted them for consideration. Five of

these were merged as is, four were rejected because of unsupported features/libraries in SnipMatch (but might be added in the future), one was discarded as a bad practice that nevertheless appeared often in our data, and one more was discarded because it already existed in SnipMatch. Finally, another snippet was rejected to allow SnipMatch “to keep the snippets balanced, *i.e.* cover more APIs equally well”. The remaining fifteen were still under consideration at the time of writing. This provides informal evidence that HAGGIS mines useful idioms that other developers find useful. Nevertheless, this experience also highlights that, as with any data-driven method, the idioms mined will also reflect any old or deprecated coding practices in the data.

7.4.4 Syntactic Idioms and Code Libraries

As a final evaluation of the mined syntactic code idioms’ semantic consistency, we now show that code idioms are highly correlated with the imported packages of a Java file. We merge the idioms across our LIBRARY projects and visualize the *lift* among code idioms and `import` statements. Lift, commonly used in association rule mining, measures how dependent the co-appearance of two elements is. For each imported package p , we compute lift l of the code idiom t as

$$l(p, t) = \frac{P(p, t)}{P(p)P(t)} \quad (7.7)$$

where $P(p)$ is the probability of importing package p , $P(t)$ is the probability of the appearance of code idiom t and $P(p, t)$ is the probability that package p and idiom t appear together. $l(p, t)$ is higher as package p and idiom t are more correlated, *i.e.*, their appearance is not independent.

Figure 7.10 shows a matrix of the lift of the top idioms and packages. We show the top 300 most frequently used packages in the training set and their highest correlating code idioms, along with the top 100 most frequent idioms in LIBRARY. Each row represents a single code idiom and each column a single package. At the top, one can see idioms that do not depend strongly on the package imports. These are generic syntactic idioms (*e.g.* Figure 7.9c) that do not correlate significantly with any package. We can also observe dark blocks of packages and idioms. Those represent library or project-specific idioms that co-appear frequently. This provides additional evidence that HAGGIS finds meaningful idioms since, as expected, some idioms are common throughout Java, while others are API or project-specific.

Suggesting idioms To further demonstrate the semantic consistency of the HAGGIS idioms, we present a preliminary approach to suggesting idioms based on package

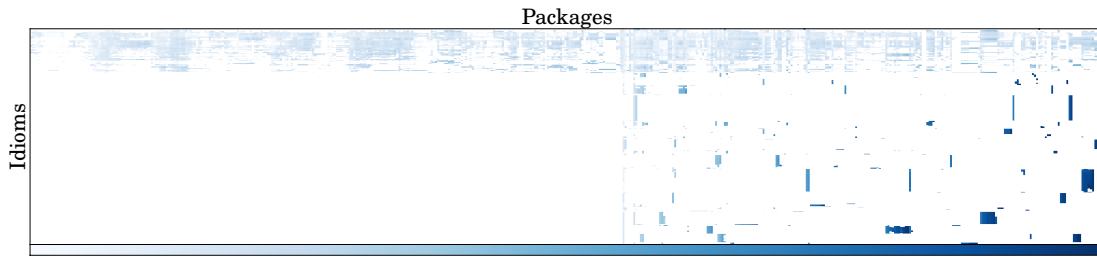


Figure 7.10: Lift (Equation 7.7) between imported packages (columns) and syntactic idioms (rows). For each package-idiom pair, we have one pixel whose color indicates the lift value. Darker blue color implies higher lift. The rows/columns were sorted in such a way that makes the clusters visible. Generic/language idioms are seen on the top since they are used across all code irrespectively of the package. The idioms on the bottom are package-specific since they are related only to specific packages as seen from the dark blocks on the right of the figure. We ordered the idioms/packages using visualization techniques that order the elements based on the top eigenvectors that make the patterns visible. Idioms and packages shown only for `android.location`, `android.net.wifi` and `org.hibernate` for brevity.

imports. We caution that our goal here is to develop an initial proof of concept, not the best possible suggestion method. First, we score each idiom \mathcal{T}_i by computing

$$s(\mathcal{T}_i|\mathbb{I}) = \max_{p \in \mathbb{I}} l(p, \mathcal{T}_i) \quad (7.8)$$

where \mathbb{I} is the set of all imported packages. We then return a ranked list $\mathbb{T}_{\mathbb{I}} = \{\mathcal{T}_1, \mathcal{T}_2, \dots\}$ such that for all $i < j$, $s(\mathcal{T}_i, \mathbb{I}) > s(\mathcal{T}_j, \mathbb{I})$. Additionally, we use a threshold s_{th} to control the precision of the returned suggestions, showing only those idioms t_i that have $s(\mathcal{T}_i, \mathbb{I}) > s_{th}$. Thus, we are only suggesting idioms where the level of confidence is higher than s_{th} . This parameter controls suggestion frequency, *i.e.* the percent of the times where we present at least one code idiom.

To evaluate HAGGIS’s idiom suggestions, we use the LIBRARY idioms mined from the train set and compute the recall-at-rank- k on the LIBRARY’s test set. Recall-at-rank- k evaluates HAGGIS’s ability to return at least one code idiom for each test file. Figure 7.11 shows that for suggestion frequency of 20% we achieve a recall of 76% at rank $k = 5$, meaning that in the top 5 results we return at least one relevant idiom 76% of the time. This result shows the quality of the mined idioms, suggesting that HAGGIS can provide a set of meaningful suggestions to developers by solely using the code’s imports. Further improvements in performance can be achieved by using advanced classification methods, which we leave to future work, and will enable an

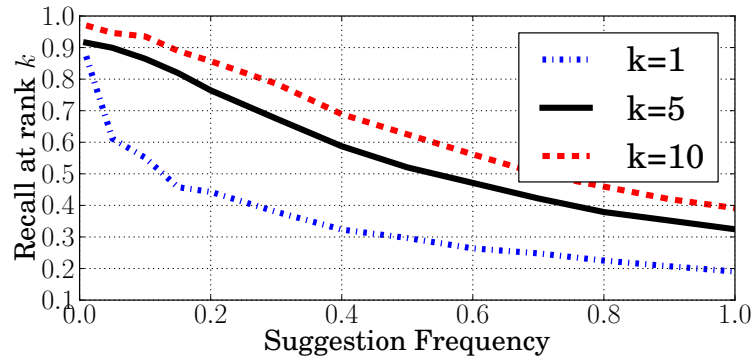


Figure 7.11: Recall at rank k for syntactic code idiom suggestion.

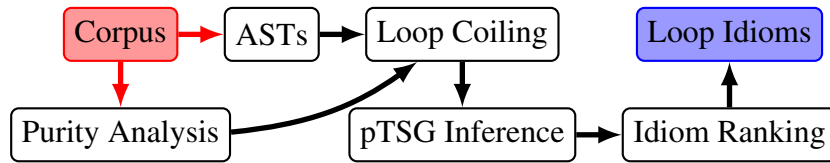


Figure 7.12: The architecture of our semantic idiom mining system, specialized to loop idioms. As Section 7.6 demonstrates, loop idioms enable code transformation toolsmiths and language designers to make data-driven decisions about which rewritings or constructs to add.

IDE side-pane with suggested code idioms.

7.5 Mining Semantic Idioms

We previously discussed syntactic idioms that captured conventional syntactic patterns of code. In this section, we are interested in mining *semantic idioms* of code, that instead of capturing the syntactic structure of a language construct or an API, capture conventional semantic operations. Syntactic idiom mining captures the syntactic form in great detail, obscuring semantic similarities, which makes it impossible to capture conventional semantic operations. Mining semantic conventions can be useful to developers of software engineering tools (toolsmiths) that employ rewriting rules within the tools that transform conventional code. For example, such tools include refactoring, program analysis tools and compiler optimizations. Semantic idioms are also useful to language and API designers that want to introduce new language and API constructs that simplify conventional semantic operations. Therefore, we view semantic idiom mining as a data-driven method for extracting *semantic* code conventions within a single project or across large corpora.

What are semantic idioms? Source code semantics denote the underlying operations performed by code. Semantics exclude various aspects of code that are not used for execution. For example, exact variable names are ignored and any syntactic sugar (*e.g.* multiple types of looping constructs) is unified. This process allows better understanding and analysis of code semantics.

When designing semantic code idioms, we aimed to abstract needless syntactic diversity, such as variable names, but retain as much of the semantics necessary. The need for semantic idioms is evident from Figure 7.7. Although syntactic idioms capture conventional use of code constructs in conjunction with API usages, they fail to capture common *semantic* operations such as the well-known conventional “map-reduce” operation of mapping one collection to another or reducing a collection to a single element (*e.g.* summation of all the elements of an array). This is because of the detailed information within the AST that is presented to the mining algorithm, inducing sparsity. Thus, mining semantic idioms with the methods presented in this chapter necessitates the abstraction of information from the ASTs and the addition of other semantic properties of the code.

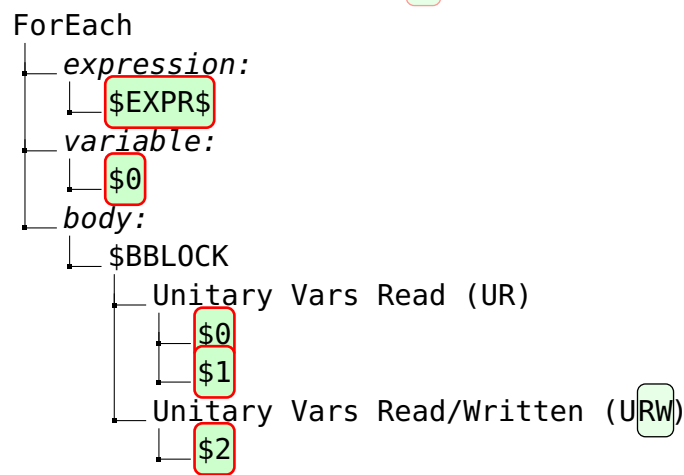
We design semantic idioms as partial semantic-augmented views of the code that allows us to reason about semantic conventions. For example, such patterns may include conventional error handling (*e.g.* exceptions), common ways of allocating and deallocating resources (*e.g.* files, mutexes) or handling of asynchronous events. In this work, we focus on semantic operations performed by loop structures (*e.g.* Figure 7.13). Loop-based semantic conventional operations are commonly used (*e.g.* “map-reduce”) to the extent that specialized libraries (*e.g.* Java’s `Streams`, C#’s `LINQ`) exist to aid the developers when writing such operations.

To mine semantic idioms, we follow a process similar to the one presented in the previous sections. First, we convert each source code snippet of interest into a “semantic” AST-like tree structure. Figure 7.13 contains an example of a concrete loop and its abstraction. In this example, the whole tree (Figure 7.13b) consist of a single idiom. When abstracting source code, described next, we remove exact variable names and maintain only semantic information about the operations within each basic block. Then, using a corpus of abstracted trees, we mine semantic idioms. In contrast to syntactic idioms, where the tree used was the full AST, this pre-processing removes syntactic information and introduces annotations for any relevant semantic information we wish to include.

Loop Semantic Idioms In this chapter, we focus on semantic idioms of loops because

```
foreach (var $0 in $EXPR$) {
    $BLOCK[UR($0, $1); URW($2);]
}
```

(a) A semantic loop idiom capturing a conventional reduce semantic operation. The `foreach`'s body is a single basic block. Within the block two unitary variables `$0` and `$1` are only read. Within the basic block a single unitary variable `$2` is read and written. Therefore, this semantic idiom is a reduce operation into the unitary variable `$2`.



(b) The abstracted AST of the semantic idiom shown above.

```
foreach(var refMap$0 in mapping.ReferenceMaps)
    this$2.AddProperties(properties$1, refMap$0.Data.Mapping);
```

(c) Concrete loop from `csvhelper` that matches the semantic loop idiom in Figure 7.13a. The numbers next to each variable shows the correspondence of each variable to the metavariables within the semantic idiom.

Figure 7.13: A sample semantic idiom, a matching loop and the modified tree structure (CAST) used for idiom mining. Example from Figure 7.2.

loops are vital to programming and program analysis. If we were to mine (semantic) loop idioms as syntactic idioms, we would not be able to find meaningful semantic patterns. The syntactic details of the code, such as variable names and exact types of operations, introduce sparsity when mining code which makes syntactic idioms very detailed and unable to find common semantic operations.

To tackle this problem we turn to abstraction. Our “coiling” abstraction removes syntactic information, such as variable and method names, while retaining loop-relevant properties like loop control variables, collections and introducing additional semantic information (*e.g.* variable purity information). Coiling transforms traditional ASTs (like those used in Section 7.3) into another tree structure which we call coiled ASTs (CAST). Our coiling process is essential for mining loop semantic idioms, since it abstracts irrelevant — for our application — variability but maintains and embeds all relevant semantic information and allows loop idiom mining to find meaningful patterns. Although in this section we focus coiling on loops, this abstraction process can be easily generalized to other code constructs. Figure 7.12 depicts the workflow of semantic loop idiom mining.

In Section 7.6, we discuss how the mined semantic loop idioms look like and how refactoring, language, and API designers can use them to identify candidate refactorings, new language constructs, or API features. To mine semantic loop idioms, we first manipulate the ASTs into coiled ASTs (CAST). The CASTs are then passed to the pTSG inference (described in Section 7.2) to mine the idioms. This section describes $f_{sem} : \mathbb{C} \rightarrow T$, *i.e.* the function in Equation 7.1 that converts source code snippets \mathbb{C} (snippets of loops in our case) to tree structures that can be mined by HAGGIS.

7.5.1 Purity Analysis

Purity information is important for loop semantics and we embed this information in the CASTs as additional semantic information that did not exist in the original AST. We call, a function *pure* in a variable (or global), when it is does not write (change the value) of that variable during its execution. *Impurity*, its complement, is a strong property of code. Using purity analysis, we tag each variable in a CAST with information about its purity. This information is significant for code semantics and thus should be directly embedded within semantic idioms.

Semantic idiom mining is agnostic to the exact method used to infer purity information. For the purposes of this work, we chose an approximate method that infers purity

information using dynamic information. Usually only a few runs of a code fragment are necessary to reveal impurity, because impure code must be carefully written to disguise its impurity and there is rarely any reason to do so. Thus, exercising a code snippet against its program's test suite is likely to detect its impurity. Armed with this intuition, we implemented an approximate dynamic purity detection technique based on testing. Given a method and a test suite that invokes that method, we run the test suite and snapshot memory before and after each invocation of the method. If the memory is unchanged across all its invocations, the method is *pure modulo the test suite*; otherwise, it is impure.

To snapshot the heap, we traverse the heap starting at the input method's reference arguments and globals. The heap is an arbitrary graph, but we traverse it breadth first as a tree and avoid loops by stopping further traversal at objects that have already been visited. Given the sequence of the traversed values we compute a hash. We compare the hashes of the before and after invocation snapshots. If the test suite does *not* execute the method, its purity, and that of its variables and globals, is unknown. Otherwise, the input method's arguments and globals are pure until marked impure. When it executes, our technique may report false negatives (incorrectly reporting a variable as pure, when it is impure) but not false positives.

Our use of a dynamic purity analysis avoids the imprecision issues common to static analyses (Xu et al., 2007) and is sufficient for mining semantic loop idioms. Other applications may require soundness; for this reason, we designed our mining procedure to encapsulate our dynamic purity analysis so that we can easily replace it with any sound static analysis (Sălcianu & Rinard, 2005; Marron et al., 2008; Cherem & Rugina, 2007), without otherwise affecting our idiom mining. An important aspect of inferring the useful idioms is to annotate their syntactic structure with semantic information, as we do here with purity information. It would be easy to further augment idioms with other semantically rich properties, such as heap/aliasing information (Barr et al., 2013; Raychev et al., 2014).

We instrument every method to realize our technique. First, we wrap its body in a `try` block, so that we can capture all the ways the function might exit in a `finally` block. At entry and in the `finally` block, we then inject snapshot calls that take the method's arguments and globals and computes their hash. In the `finally` block after the snapshot, we compare the hashes and mark any variables that point to memory that changed as impure.

To speed up our purity inference and avoid the costly memory traversals, we use

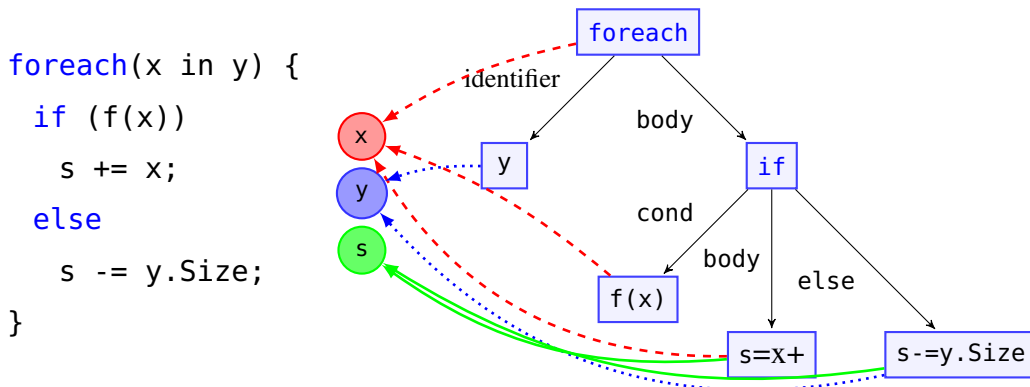


Figure 7.14: Abstract Syntax Tree with References. A reference (depicted red, blue and green circles) is a set of AST nodes that refer to the same program variable. We label each AST node with zero or more references.

exponential backoff: if a method has been tested n times and has been found pure with respect to some argument or global, then we test purity only with probability p^n . We used $p = 0.9$. As a further optimization and to avoid stack overflows, we assume that by convention the overridden methods `GetHashCode()` and `Equals(object)` methods to be pure and ignore them. These methods may be invoked very frequently and therefore instrumentation is costly. Our method cannot detect when a variable is overwritten with the same value. This is a potential source of false negatives to the extent to which such identity rewritings are correlated with impurity.

Since we cannot easily rewrite and rebuild libraries, our technique cannot assess the purity of calls into them. However, they are frequent in code, so we manually annotated the purity of about 1,200 methods and interfaces in core libraries, including CoreCLR. These annotations encompass most operations on common data structures such as dictionaries, sets, lists, strings *etc.*

7.5.2 Coiling Loops

In Section 7.3 we showed how to extract syntactic idioms from code. Here, we mine loop idioms to capture universal semantic properties of loops. Thus, we keep only AST subtrees rooted at loop headers and abstract nodes that obscure structural similarities. We call this process *coiling* and detail its abstractions next.

Program Variables In conventional ASTs, a node refers to a single variable. Coiling breaks this rule, creating nodes that potentially refer to many variables. Because we want to infer loop idioms that incorporate and therefore match patterns of variable

usage, we need to re-encode variable usage into the AST. To this end, we introduce the notion of a reference. A *reference* is a set of nodes that refer to the same program variable. We label nodes with zero or more references as depicted in Figure 7.14. To combat sparsity, our pTSG inference merges two references that share the same node set. Thus, an idiom can match a concrete loop that contains more variables than the number of references in the idiom.

Expressions Expressions are quite diverse in their concrete syntax, due to the diversity of variable names, but are usually structurally quite similar. Since our goal is to discover universal loop properties, we abstract loop expressions to a single `EXPR` node, labeled with the variables that it uses. There are three exceptions: increment, decrement, and loop termination expressions. The pre and post increment and decrement operators from C introduce spurious diversity in increment expressions. Thus, we abstract all increment and decrement operations to the single `INC/DEC` node. We preserve the top-level operator of a termination expression and rewrite its operands to `Expr` nodes, with the exception of bounding expressions that compute a size or length, which we rewrite to a `SizeOf` node and label it with the reference to the measured variable.

Basic Blocks A basic block (`BLOCK`) is sequence of lines of code lacking control statements. Basic blocks tend to be quite diverse, so we collapse them into a single node labeled with references to the variables they use. These basic blocks are equivalent to uninterpreted functions. To make our pTSG inference aware of purity, we encode the purity of each of a basic block's variables as children of the basic block's node. We label each child node with its variable's reference and give it a node type that indicates its purity in the basic block. The purity node types are `R`, `W`, and `(RW)` (we circumscribe `RW` for notational purposes).

Loops usually traverse collections, so we distinguish them from unitary (primitive or non-collection) variables. The annotation U denotes a unitary variable. For collection variables (denoted by C), we separate them into their *spine*, the references that interconnect the elements, and the elements it contains. Our purity analysis separately tracks the mutability of a collection's spine C^S and its elements C^E . This notation allows us to detect that a collection has changed when the same number of elements have been added and removed, without comparing its elements to the elements in a snapshot. In practice, the spine and the elements change together most often and only 9 idioms of the top 200 idioms (with total coverage 1.2%) have loops that change the elements of a collection, but leave the spine intact.

Blocks For us, a block is a graph of basic blocks; it is the code that appears between {

and `}` delimiters in a C-style language. Blocks can have multiple exits, including those, like `break` or `continue` statements, that exit the loop. Coiling assigns different node types to single and multi-exit blocks. This allows our pTSG inference to infer loop idioms that distinguish single exit blocks, whose enclosing loops are often easier to refactor or replace.

Example Figure 7.15 shows example idioms and concrete loops they match annotated with the binding of the idiom’s references to the program variables. In Figure 7.15-8, the idiom contains `var $0=0`, because Roslyn⁸ defines the initializer as a statement. The idiom contains the `<` operator, because our expression abstraction preserves the top-level operator in termination expressions. `INC` denotes the special node for increment expression. It contains a single block that, in turn, contains a single basic block that references at least (since we merge references with identical sets of nodes) four variables: The first two are read-only unitary variables (denoted by `UR`); `$2` is a collection with a read-only spine and elements (denoted by `CSR` for the spine and `CER` for the elements); and `$4` is a read-write unitary variable (denoted as `U(RW)`).

7.5.3 Mining Semantic Idioms

So far we described f_{sem} , *i.e.* the function in Equation 7.1 that deterministically converts the code into tree structures, that can be used within the pTSG inference framework. The CASTs returned from f_{sem} are passed the inference process described in Section 7.2. For mining semantic idioms we make two small modifications to the inference framework. First, we simplify the prior (Equation 7.2) by removing the geometric distribution P_{geom} over the size of the idioms, because we found that it heavily penalizes large but otherwise highly probable idioms, *i.e.*

$$P_0(T) = \prod_{r \in T} P_{ML}(r). \quad (7.9)$$

Thus, our prior pTSG is simply the PCFG defined over our dataset. In addition, instead of the Dirichlet process (Equation 7.3), we use the more general Pitman-Yor process. Regarding the hyperparameters, we observe that in practice — and for our data — different values do not make any difference.

⁸<http://roslyn.io>

7.5.4 Semantic Idiom Ranking

After mining the idioms, we rank them in order of their utility in characterizing the target constructs — loops in our case. The ranked list provides data-based evidence to interested parties (*e.g.* API designers, refactoring toolsmiths) augmenting their intuition when identifying the most important code constructs. To mine idioms, we use a score that computes a trade-off between coverage and idiom expressivity. If we solely ranked idioms by coverage, we would end up picking very general and uninformative loop idioms, as would happen with frequent tree mining. We want idioms that have as much information content as possible *and* the greatest possible coverage. We score each idiom by multiplying the idiom’s coverage with its cross-entropy gain. Cross-entropy gain measures the expressivity of an idiom and is the average (over the number of CFG productions) log-ratio of the posterior pTSG probability of the idiom over its PCFG probability. This ratio measures how much the idiom improves upon the base PCFG. To pick the top idiom we use the following simple iterative procedure. First, we rank all idioms by their score and pick the top idiom. Then, we remove all loops that were covered by that idiom and repeat the process. We repeat this until there are no more loops covered by the remaining idioms. This greedy knapsack-like selection yields idioms that achieve both high coverage and are highly informative. Since purity information is explicitly encoded within the CASTs (as special nodes, as discussed in Subsection 7.5.2), the ranking takes into consideration both the purity information as well as the other information about each loop.

7.6 Semantic Idioms Evaluation

This work rests on the claim that we can mine semantic idioms of code to provide data-driven knowledge to refactoring toolsmiths and API designers. The goal of mining loop idioms is to reduce the cost of identifying loop rewritings (*e.g.* for loop-to-LINQ refactoring) by working on loop idioms, instead of concrete loops. A necessary condition for this is an effective procedure for mining loop idioms that cover, or match, a substantial proportion of real world loops.

Coverage of Idiomatic Loops Our semantic loop idioms are mined from a large set of projects consisting of 577kLOCs (Table 7.5), which form our training corpus. Figure 7.16 shows the percent coverage achieved by the *ranked* list of idioms. With the first 10 idioms, 30% of the loops are covered, while with 100 idioms 62% of the loops

Semantic Idiom	Sample Matching Concrete Loop
<pre>for(int \$0=\$EXPR; \$0<\$EXPR; INC(\$0)) \$BLOCK[UR(\$0);C^{S,E}R(\$1); URW(\$2)]</pre>	<pre>for (int i \$0 = 0; i \$0 < length; i \$0++) charsNeeded \$2 += components \$1[i \$0].Length;</pre>
(1) Semantic Operation: Reduce with for Coverage: 14%	
<pre>foreach(var \$0 in \$EXPR) \$BLOCK[UR(\$0, \$1); URW(\$2);]</pre>	<pre>foreach(Term term \$0 in pq.GetTerms()) rootMap \$2.AddTerm(term \$0.Text, query \$1.Boost);</pre>
(2) Semantic Operation: Reduce with foreach Coverage: 2%	
<pre>foreach(var \$0 in \$EXPR) \$BLOCK[UR(\$0, \$1);C^{S,E}RW(\$2)]</pre>	<pre>foreach(DictionaryEntry entry \$0 in dict) hash \$2[entry \$0.Key]=entry \$0.Value;</pre>
(3) Semantic Operation: Map with foreach Coverage: 2%	
<pre>foreach(var \$0 in \$EXPR) \$BLOCK[UR(\$1); URW(\$0, \$2);]</pre>	<pre>foreach(var exp \$0 in args) exp \$0.Emit(member \$1, gen \$2);</pre>
(4) Semantic Operation: Map overwrite and reduce with foreach Coverage: 2%	
<pre>for(int \$0=\$EXPR; \$0<\$EXPR; INC(\$0)) \$BLOCK[UR(\$0, \$1);C^{S,E}R(\$2);C^{S,E}RW(\$3)]</pre>	<pre>for (int k \$0=a; k \$0<b; k \$0++) ranks \$3[index \$2[k \$0]] = rank \$1;</pre>
(5) Semantic Operation: Map collection-to-collection with for . Coverage: 5%	
<pre>for(int \$0=\$EXPR; \$0<\$EXPR; INC(\$0)) \$BLOCK[UR(\$0, \$1);URW(\$2);C^{S,E}RW(\$3)]</pre>	<pre>for (var k \$0= 0; k \$0<i; k \$0++){ d \$3[k \$0] /= scale \$1; h \$2 += d \$3[k \$0] * d \$3[k \$0]; }</pre>
(6) Semantic Operation: Map and reduce with for . Coverage: 5%	
<pre>foreach(var \$0 in \$EXPR) \$BLOCK[UR(\$1);URW(\$0)]</pre>	<pre>foreach(LoggingEvent event \$0 in loggingEvents) event \$0.Fix = m_fixFlags \$1;</pre>
(7) Semantic Operation: Map and overwrite foreach . Coverage: 1%	
<pre>for(var \$0=0; \$0 < \$EXPR(\$1,\$2,\$3); INC(\$0)){ if(\$EXPR(\$0, \$1, \$2, \$4, \$5)) \$BLOCK[UR(\$0, \$1);URW(\$4);C^{S,E}R(\$2)] }</pre>	<pre>for(int i \$0=0; i \$0<data \$2.Length \$3; i \$0++){ if(data \$2[i \$0]>max \$4 && !float \$5.IsNaN(data \$2[i \$0])) max \$4 = data \$2[i \$0]; }</pre>
(8) Semantic Operation: Reduce with for and conditional Coverage: 1%	

Figure 7.15: Sample semantic idioms and a concrete loops they match.

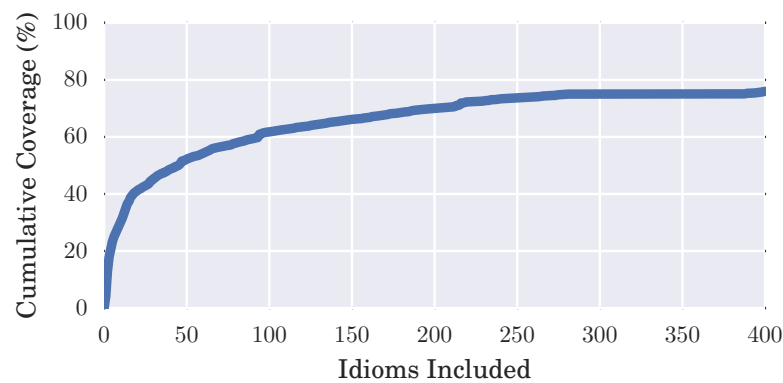


Figure 7.16: Cumulative loop coverage vs. the number of (top) semantic idioms used. Given the diminishing returns the distributions fits well into a Pareto distribution. The Gini coefficient is $G = 0.785$ indicating a high coverage inequality among idioms. When using 50 idioms, 50% of the loops can be covered and with 200 idioms 70% of the loops are covered. 22% of the loops in our corpus are non-idiomatic (*i.e.* are not covered by an idiom).

are covered. This shows that idioms have a Pareto distribution — a core property of natural code — with a very few common idioms and a long tail of less common ones. This shows a useful property of the idioms. As a toolsmith or a language or API feature designer uses the idioms, she will capture gradually more loops, but with diminishing returns. In our case, the top 50 idioms capture about 50% of the loops, while the next 100 idioms increase the coverage only by another 20%. Therefore, our data-driven approach allows the prioritization of semantic code idioms — such as the design of loop-to-LINQ refactorings — and helps to achieve the highest possible coverage with the minimum possible effort.

Nonidiomatic Loops Figure 7.16 shows that about 22.4% of the loops are not covered by any of the idioms. Here, we perform a case study of these nonidiomatic loops. We sampled uniformly at random 100 loops that were not covered by any of the mined idioms and studied how they differed from idiomatic loops. We found that 41% of these loops were in test suites, while another 8% of the nonidiomatic loops were loops that were either automatically generated or were semi-automatically translated from other languages (*e.g.* Java and C). Another 13% of these loops were extremely domain-specific loops (*e.g.* compression algorithms, advanced math operations), which suggests that some of these operations are distinct and highly specialized semantic operations. The rest of the nonidiomatic loops were seemingly normal. However, we noticed they often contain rare combinations of control statements (*e.g.* a `for` with an `if` and another

Table 7.5: C# Projects (577kLOC total) that were used to mine semantic loop idioms after collecting purity information by running their test suites (containing 34,637 runnable unit tests).

Name	Git SHA	Description
Core	3b9517	Castle Framework Core
csvhelper	7c63dc	Read/write CSV files
dotliquid	9930ea	Template language
libgit2sharp	f4a600	C# Git implementation
log4net	782e82	Logging framework
lucenenet	70ba37	Full-text search engine
mathnet-numerics	f18e96	Math library
metrics-net	9b46ba	Metrics Framework
mongo-csharp-driver	6f237b	Database driver
Nustache	23f9cc	Logic-less templates
Sandra.Snow	c75320	Static Site Generator

loop inside the `else` statement), convoluted control flow in the body of the loops or rare purity properties. Some of these rare combinations, like two consecutive `if-else` statements, are, in isolation, normal or frequent, but rare when enclosed in a loop rather than a method. We speculate these loops look normal to developers because we humans tend to notice the local normality, while neglecting the abnormality of the neighborhood. Unsurprisingly, our method also considers loops with empty bodies nonidiomatic. The analysis of the all the studied loops in this paragraph suggests that our method correctly designates these loops nonidiomatic. Knowing which loops are nonidiomatic and that they are rare is crucial, since it allows toolmakers to avoid wasting time on them.

Project Specificity of Semantic Loop Idioms So far we discussed idiom coverage with regards to the corpus used for inferring the pTSG. Now, we are interested in characterizing the project specificity of the mined loop idioms. For each of the 11 projects, we infer a pTSG on the other 10 projects and compute the coverage of the new top 200 computed idioms on the target project. We find that the average percent of loops that are covered by the top 200 idioms trained on all the projects is 70.1% but when the project is excluded from the training set, it drops to 66.3%. This shows both that the

top ranked loop idioms are general and that there is nontrivial proportion of domain-specific loop idioms. There are two exceptions: the lucenenet text search engine and the mathnet-numerics math library that about 18% of the loops are project-specific and cannot be covered by idioms found in the other projects. By manually investigating the project-specific idioms, we find that mathnet-numerics has a significant number of math-related specialized loop idioms, while lucenenet’s project-specific idioms are mostly in its Snowball stemmer, which is autogenerated code that has been ported from Java and is highly specialized to the text-processing domain.

These results show how mining loop idioms identify not only universal, domain-independent loop idioms that are frequent yet detailed enough for use within applications but also domain, even project, specific loop idioms that may still benefit from custom-defined replacement transformations. Our data-driven approach allows toolsmiths and language designers to abandon the straitjacket of building “one size fits all” tools, forced on them by limited engineering resources, and build bespoke, even adaptable, transformation tools that many more developers can use.

7.6.1 Using Semantic Loop Idioms

Popular semantic idioms identify opportunities for identifying “natural” rewritings of code patterns that are structurally and semantically similar and frequent enough to warrant the cost of abstracting and reusing them. Therefore, highly ranked idioms can serve as a useful basis for language and API designers, representing opportunities for introducing new language constructs or new APIs. Highly ranked idioms can also serve as a useful basis for the lefthand side of a rewriting rule that a refactoring toolsmith or a compiler optimization developer is creating. For each idiom, one has to write the righthand side of the rewriting rule. For example, loop idioms, our focus in this work, are well-suited for identifying opportunities for the task of rewriting loops using new APIs, defining new language constructs that can simplify common operations or even functionalizing them into LINQ statements. Because these rules are mined from actual usage, we refer to this process as *prospecting*. Our semantic idiom ranking allows to prioritize patterns (*e.g.* when creating a rewriting rule) in such a way that the top idioms achieve the maximum codebase coverage possible. In this section, we discuss three case studies on using loop idioms for prospecting. In the first case study, we discuss how loop idioms can be used for prospecting loop-to-LINQ rewritings, then we show some evidence that loop idioms can help with designing better APIs or even provide

data-driven arguments for introducing new language features.

Prospecting Loop-to-LINQ Refactorings Semantic loop idioms can help in an important instance of refactoring: replacing loops with functional operators. Since 2007, C# supports Language Integrated Query (LINQ) (Meijer, 2011; Marguerie et al., 2008), that provides functional-style operations, such as map-reduce, on streams of elements and is widely used in C# code. LINQ is concise and supports lazy operations that are often easy to parallelize. For example, multiplying all elements of the collection `y` by two and removing those less than 1, in parallel, is `y.AsParallel().Where(x=>x<1).Select(x=>2*x)`. We call a loop that can be replaced with LINQ statements “LINQable”. LINQability has important implications for the maintainability and comprehensibility of code. LINQ’s more conceptually abstract syntax 1) manifests intent, making loops easier to understand and more amenable to automated reasoning and 2) saves space, in terms of keystrokes, as a crude measure of effort to compose and read code.

As a testament to the importance of refactoring loops to functional operators, two tools already support such operations: LAMBDAFICATOR targets Java’s Streams and JetBrains’s Resharper (JetBrains, 2015) replaces loops with LINQ statements. Both these tools have followed the classic development model of refactoring tools: they support rewritings that its toolsmiths decided to support from first principles (Gyori et al., 2013). Our approach can complement the intuition of the tool makers and therefore allow these tools to support refactorings that the tool authors would not envision without data, enabling the data-driven, inference-based development of refactorings. Additionally, data-driven inference allows toolmakers to discover project or domain-specific semantic idioms without needing a deep knowledge of a domain or a specific project.

To do this toolsmiths can build a refactoring tool using loop idioms as key elements to the rewritings that map loops to LINQ statements. In other words, we can use our semantic idioms inference to automatically identify loop constructs that could be replaced by a LINQ operator, *i.e.* are LINQ-able.

To evaluate the fitness of our loop idiom mining for prospecting natural loop rewritings, we built an idiom-to-LINQ suggestion engine. This engine is *not* a refactoring tool. Its purpose is to validate the quality of the semantic loop idioms it prospects and to show how a refactoring toolsmith could make use of them. To build our suggestion engine, we manually mapped loop idioms to LINQ operators. These suggestions are *not* sound, since our engine simply matches an idiom to a concrete loop and does *not* check for semantics preservation as a fully automated sound refactoring tool would require. For example, our idiom-to-LINQ suggestion engine maps the idiom in Figure 7.15.8

to a reduce operation. Thus, for the concrete loop in Figure 7.15.8, the suggestion engine outputs the loop and its location, then replaces references with the concrete loop's variable names and outputs the following suggestion:

The loop is a reduce on `max`. Consider replacing it with:

```
data.Where(cond).Aggregate((elt, max)=>accum)
```

Notes

1. `Where(cond)` may not be necessary.
2. Replace `Aggregate` with `Min` or `Max` if possible.

We know that this loop is a reduce because the matching idiom's purity information tells us that there is a read-write only on a unitary variable. When our suggestion engine accurately suggests a loop refactoring, a refactoring toolsmith should find it easy to formalize a rewriting rule (*e.g.* identifying and checking the relevant preconditions) using loop idioms as a basis. In our example, a polished refactoring tool should refactor the loop in Figure 7.15.8 into `data.Where(x=>!float.IsNaN(x)).Max()`. In our corpus, we find that at least 55% of all loops are LINQable.

We used the top 25 idioms that cover 45.4% of the loops in our corpus. We mapped 23 idioms, excluding two of the loop idioms (both `while` idioms, covering 1.5% of the loops) that have no corresponding LINQ expression. To map each idiom to an expression, we found the variables that match the references, along with the purity and type information of each variable. We then wrote C# code to generate a suggestion template, as previously described. The process of manually mapping the top 23 idioms to LINQ took less than 12 hours.

With this map, our engine suggests LINQ replacements for 5,150 loops. Each idiom matches one or more loops and is mapped to a LINQ expression in our idiom-to-LINQ map. To validate the quality of these suggestions, we uniformly sampled 150 loops and their associated suggestions. For each of these loops, two annotators assessed our engine's suggestion accuracy. This should *not* be seen as a part of an effort to develop a batch-refactoring tool, but rather as a means of evaluating our proposed method. Our results show that the suggestions are correct 89% of the time. The inter-rater agreement was $\kappa = 0.81$ (*i.e.* agreed 96% of the time). So not only is our idiom to LINQ map easy to build, it also achieves good precision. This suggests that the mined idioms indeed learn semantic loop patterns that can be used for refactoring. Table 7.6 shows how often common LINQ operators are used when loops are fictionalized to LINQ expressions

Table 7.6: Basic LINQ operators and coverage statistics from the top 100 loop idioms. # Idioms is the number of idioms our suggestion engine maps to a LINQ expression that uses each LINQ operator. Use frequency is the proportion of concrete loops that when converted to LINQ use the given LINQ operator.

Operator	Description	# Idioms	Use Frequency
Range	Returns integer sequence	50	77%
Select	Maps a lambda to each element	42	32%
Aggregate	Reduce elements into a value	43	21%
SelectMany	Flattens collection and maps lambda to each element	5	10%
Where	Filters elements	13	7%
Zip	Combines two enumerables	6	3%
First	Returns the first element	2	1%

using our top 100 loop idioms. This evaluation indicates that a refactoring toolsmith can easily use a loop idiom as the lefthand side of a refactoring rule. She can then write extra code that checks for the correctness of the refactoring. Most importantly, this process allows the prioritized consideration of rewritings that can provide the maximum codebase coverage with the minimal effort.

Prospecting New Library APIs The top mined loop idioms are interesting semantic patterns of the usage of code. However, some of the common patterns may be hard to read and cumbersome to write. Since semantic idioms represent common operations, they implicitly suggest new APIs that can simplify how developers invoke some operation. Thus, the data-driven knowledge that can be extracted from semantic idiom mining can be used to drive changes in libraries, by introducing new API features that make its usage easier. For example, one common set of semantic loop idioms (covering 13.7% of the loops) have the form

```
foreach (var element in collection)
    obj.DoAction(foo(element))
```

where each element in the collection is mapped using foo and then some non-pure action (DoAction in the code snippet above) is performed on obj. The frequent usage of this loop idiom for a given API can provide a strong indication that a new API feature

should be added. For example in lucenenet the following (slightly abstracted) loop appears

```
for (int i = 0; i < numDocs; i++) {
    Document doc = function_to_get_doc(i);
    writer.AddDocument(doc);
}
```

In this example, the method `AddDocument` does not support any operation that adds more than one `Document` object at a time. This forces the developers of the project to consistently write loops that perform this operation. Adding an API method `AddDocuments`, that accepts enumerables would lead to simpler, more readable and more concise code:

```
writer.AddDocuments(collection.Select(doc => foo(doc)))
```

Prospecting New LINQ Operators Mined semantic loop idioms can implicitly help with designing new LINQ operators. For example, while mapping loop idioms to LINQ, we found 5 idioms (total coverage of 5.4%) that map to the rather cumbersome LINQ statement

```
Range(0, M).SelectMany(i => Range(0, N)
    .Select(j => foo(i, j)))
```

These idioms essentially are doubly nested `for` loops that perform some operation for each `i` and `j`. This suggests that a 2-d `Range` LINQ operator, would be useful and would cover about 5.4% of the loops. In contrast, our data suggests that a n -d ($n > 2$) `Range` operator would be used very rarely and therefore no such operator needs to be added. We note that we have found two StackOverflow questions⁹ with 15k views that are looking for a similar functionality. Another example is a set of idioms (coverage 6.6%) that map to

```
Range(M, N).Select(i=>foo(collection[i]))
```

essentially requiring a slice of an ordered collection.¹⁰ The common appearance of this idiom in 6.6% of the loops provides strong data-driven evidence that a new feature would be highly profitable to introduce. For example, to remove these loops or their cumbersome LINQ equivalent, we could introduce a new `Slice` feature that allows the more idiomatic `collection.Slice(M, N).Select(foo)`. Indeed, the data has helped

⁹<http://stackoverflow.com/questions/3150678> and <http://stackoverflow.com/questions/18673822>

¹⁰ This could also be mapped to the equally ugly `collection.Skip(M).Take(N-M).Select(foo)`.

us identify a frequently requested functionality: This operation seems to be common enough that .NET 3.0 introduced the `slice` feature, but only for arrays rather than for arbitrary sequential collections (*e.g.* `Lists`). Additionally, the need of such a feature — that we automatically identified through data — can be verified by the existence of a highly voted StackOverflow question¹¹ with 122k views and 15 answers (with 422 votes in total) asking about slicing with some of the answers suggesting a `Slice` LINQ extension function.

Finally, we observe that some loops perform more than one impure operation (*e.g.* adding elements to two collections), while efficiently reusing intermediate results. To refactor this with LINQ statements an intermediate LINQ expression needs to be converted to an object (*e.g.* by using `ToList()`) to be consequently used in two or more other LINQ expressions, because of the laziness of LINQ operators. This is not memory efficient and may create an unneeded bottleneck when performing parallel LINQ operations. A memoization LINQ operator that can distribute the intermediate value into two or more LINQ streams, could remove such hurdles from refactoring loops into LINQ.

In our dataset, LINQ slicing seems to be a common idiom required across many projects suggesting that an addition to core LINQ API could be reasonable. In contrast, the `2d Range` seems to be most commonly used within the `mathnet-numerics` project, suggesting that a domain-specific helper/extension LINQ operator could be introduced in that project. This discussion shows how mining loop idioms allows toolsmiths and language designers to leverage data to guide the evolution and improvement of their tools and languages, all with the aim of making code more readable and maintainable.

Prospecting New Language Features Semantic loop idioms can provide data-driven evidence for the introduction of new language features. For example, some of the top loop idioms suggest novel language features. For example, 5 top loop idioms with total coverage 12% have the form:

```
for (int i=0; i < collection.Length; i++) {  
    foo(i, collection[i])  
}
```

where they are iterating over a collection but also require the index of the current element. A potential new feature would be the introduction of an `Enumerate` operation that would jointly return the index and the element of a collection. This resembles the

¹¹<http://stackoverflow.com/questions/406485>

enumerate function that Python already has and Ruby’s `each_with_index`. Interestingly, using loop idioms we have identified a common problem faced by C# developers: in StackOverflow there is a related question for C#¹² with about 379k views and a highly voted answers (453 votes) that suggests a helper method for bypassing the lack of such a function. In addition there are two questions about the same feature in Java.¹³

7.7 Conclusions

In this chapter we presented HAGGIS, a system for automatically mining high-quality semantic and syntactic code idioms. The idioms discovered include project, API, and language specific idioms. One interesting direction for future work is the question of why syntactic code idioms arise and their effect on the software engineering process. It may be that there are “good” and “bad” idioms. “Good” idioms could arise as an additional abstraction over programming languages helping developers communicate more clearly their intention. “Bad” idioms may compensate for deficiencies of a programming language or an API. For example, the “multi-catch” statement in Java 7¹⁴ was designed to remove the need for a syntactic idiom that consisted of a sequence of catch statements with identical bodies. However, it may be argued that other idioms, such as the ubiquitous `for(int i=0; i<n; i++)` aid code understanding. An empirical study about the differences between these types of idioms could be of great interest to software engineers and library and language designers.

In this chapter, we also extended our method for unsupervised mining of *semantic* idioms, specifically loop idioms, from a corpus of idiomatic code (Section 7.5). By abstracting the AST and augmenting it with semantic facts like purity, we showed that idiom mining can cope with syntactic diversity to find and prioritize patterns whose replacement might improve a refactoring tool’s coverage or help programming language and API designers introduce new features. Idioms can also benefit other areas of program analysis and transformation, guiding the selection of heuristics and choice of corner cases with data, as in auto-vectorization (Barthe et al., 2013).

¹²<http://stackoverflow.com/questions/43021/>

¹³<http://stackoverflow.com/questions/23817840> and <http://stackoverflow.com/questions/7167253/>

¹⁴<https://docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html>

Chapter 8

Conclusions

“We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance.”

– John Archibald Wheeler (Scientific American
1992, Vol. 267)

This dissertation presented the first — to our knowledge — approach to tackling the problem of learning natural coding conventions including naming, syntactic and semantic coding conventions. This was achieved by carefully designing machine learning models that learn to reason and analyze various aspects of source code.

In Chapter 4, we presented two machine learning models that learn to predict the name of a variable given its context. Variables are the basic building blocks of source code and learning to name them requires an understanding of their role and function. The machine learning models presented in this dissertation model this role and function in a probabilistic way. In the next chapter (Chapter 5), we discussed the more complicated problem of learning method naming conventions. To learn the naming conventions of methods or arbitrary code snippets, machine learning methods need to learn to reason about the procedural knowledge within the code. We achieve this using a sophisticated neural attention-based convolutional network that learns continuous representations of the procedural knowledge within the code and uses this representation to reason about naming each method. We show that this model outperforms other alternative machine learning models.

In Chapter 6, we take the ideas presented in Chapter 5 further and explore a core machine learning question: “*Can we learn semantic continuous representations of procedural knowledge?*”. We explore this ambitious goal using boolean and polynomial

symbolic expressions grouped into various equivalence classes. We design and train a machine learning model to learn to map each equivalent expression — regardless of its syntactic structure — to a single semantic vector representation and show that our model outperforms other alternatives. Learning accurate semantic representations of code can help us reason about procedural knowledge and the conventional semantic operations devoid of the syntax of the language we use to describe it.

In the final section (Chapter 7), we discuss unsupervised methods for mining syntactic and semantic idioms of source code. These idioms represent conventional “mental chunks” of source code that developers use. We show that syntactic idioms can be useful within documentation, whereas semantic idioms can be used by designers of software engineering tools to achieve greater coverage of a tool. In addition, idioms can be useful to API and language designers helping them mine common usage patterns of an API or a language feature to design new APIs and language features that simplify the code.

The Software Engineering Perspective Modeling and understanding source code artifacts with machine learning can have a direct impact in software engineering. The problem of learning to name variables and methods and learning to represent semantics of expressions is a first step towards the more general goal of developing machine learning representations of source code that will allow machine learning methods to reason probabilistically about code resulting in useful software engineering tools that will help code construction and maintenance.

Within the realm of coding conventions in software engineering, this dissertation has the potential to change how conventions are inferred and enforced. Machine learning methods allow us to infer the “emergent” naming conventions within a codebase without the need of writing explicit rules. Enforcing those conventions makes codebases more coherent and therefore maintainable.

The Machine Learning Perspective Source code — and its derivative artifacts — represent a new modality for machine learning with very different characteristics compared to images and natural language. Therefore, models of source code necessitate research into new methods that could have interesting parallels to images and natural language. This work is a step towards this direction: the machine learning models presented in the dissertation attempt to “understand” the highly-structured format of source code. This is not just useful for building smart software engineering tools. Machine learning and artificial intelligence that needs to synthesize or reuse existing code needs may use such methods to understand the code and its properties.

The Programming Languages Perspective Within programming language research, coding conventions may not be as important as other (formal) properties of the code. However, the models presented here can be seen as methods for probabilistically understanding source code and can be useful within methods of code analysis that need to first provide “educated” guesses about code properties, before proving formally that a property holds. Finally, syntactic and semantic code idioms will be widely useful for programming language designers that need to introduce new features to simplify the usage of the languages and let their users write more robust and maintainable code. Automatically mining semantic code idioms may also have implications in compiler optimization and verification where idioms provide hints for selecting relevant proof strategies and proof premises.

8.1 Future Work

There are many possible direction for future work, some of which are discussed here.

Transfer of Models to Industry In this dissertation we presented a series of methods for learning and enforcing coding conventions. Although the models were evaluated in a principled manner new challenges and opportunities will arise when these models are used within an industrial setting. The scalability of the models as well as the usability factors of such tools need to be empirically studied. In addition, incorporating user feedback to improve the recommendation quality of models that suggest new variable and method names presents interesting challenges in the machine learning domain.

Learning to Name Variables for Deobfuscation Recently, [Bichsel et al. \(2016\)](#) introduced a structured prediction model for deobfuscating source code. This work uses machine learning methods on carefully constructed dependency graphs to select simultaneously names for multiple identifiers. This model can be easily combined with the deep learning methods presented in Chapter 4 and Chapter 5 to learn to capture subtoken naming conventions and learn non-hand-crafted dependency features, potentially yielding better performance.

Structured Models of Autocompletion The learned code idioms suggest that larger patterns tend to appear often in code. However, current autocompletion tools usually autocomplete only one token per time. Using the notion of code idioms and new models for statistical autocompletion, we might be able to create new tools that *learn* to autocomplete larger patterns of code.

Research on Coding Conventions The empirical use of coding conventions in almost all software projects suggests that it plays an important role to software engineers. Empirical studies have confirmed some of the benefits of coding conventions, but further investigation is required. Such investigation may have impact on the design of programming languages and the conventions that are used. Furthermore, apart from enforcing coding conventions within an industrial setting, the work presented in this dissertation may be useful for educational purposes, *i.e.* for teaching students (such as in a MOOC setting) not only how to write correct code to achieve a task, but also how to write *conventional* code.

New Machine Learning Models for Other Coding Conventions Novel machine learning methods that look at the code at a significantly higher level of abstraction need to be researched to allow inference of other coding conventions, such as architectural conventions. For example, new machine learning methods are required to model and suggest the sparse nature of software architectures and transfer architectural knowledge across teams and projects.

Composability and Structure in Machine Learning The machine learning models used within this dissertation and the related work are mostly based on simple code representations. However, code is a highly structured object with multiple forms and complex structure. Existing machine learning methods are not able to efficiently fuse multiple representations and work on highly complex graph structures, such as highly-detailed program dependency graphs. This suggests that more research is still required to create structured and compositional models that are able to capture the rich semantics of source code and its derivative artifacts.

Appendix A

List of Published Work

This appendix contains all the published work of the author during the PhD. The order is chronological.

- Mining Source Code Repositories at Massive Scale using Language Modeling ([Allamanis & Sutton, 2013b](#)).
- Why, When and What: Analyzing Stack Overflow Questions by Topic, Type and Code ([Allamanis & Sutton, 2013a](#)).
- Learning Natural Coding Conventions ([Allamanis et al., 2014](#)).
- Autofolding for Source Code Summarization ([Fowkes et al., 2016](#)).
- Mining Idioms from Source Code ([Allamanis & Sutton, 2014](#)).
- Bimodal Modelling of Source Code and Natural Language ([Allamanis et al., 2015b](#)).
- Suggesting Accurate Method and Class Names ([Allamanis et al., 2015a](#)).
- A Convolutional Attention Network for Extreme Summarization of Source Code ([Allamanis et al., 2016d](#)).
- Learning Continuous Semantic Representations of Symbolic Expressions ([Allamanis et al., 2016c](#)).
- Tailored Mutants Fit Bugs Better ([Allamanis et al., 2016b](#)).
- Mining Semantic Loop Idioms from Big Code ([Allamanis et al., 2016a](#)).

Appendix B

GitHub Pull Request Discussions

This appendix contains the discussions that we had during the submission of candidate variable renamings, as discussed in Subsection 4.4.2. These are included here for achieving purposes.

Renaming variables for codebase consistency #834

Merged stefanbirkner merged 3 commits into junit-team:master from mallamanis:master on 26 Feb 2014

Conversation 10

Commits 3

Files changed 20

+64 -64



mallamanis commented on 25 Feb 2014

I've been working on a research machine learning-based tool (link: <http://groups.inf.ed.ac.uk/naturalize/>) tool that analyzes source code identifiers and makes suggestions for renaming them. The goal is to reduce unnecessary diversity in variable naming and improve code readability. This pull request is only a small sample of the suggestions made for JUnit.

No functional changes were made in any of the commits.

Renamed "result" (18.69%) to "suite" (81.31%). The naturalize tool de... d70ca7f

kcooney and 2 others commented on an outdated diff on 25 Feb 2014

```
src/main/java/junit/framework/TestResult.java
@@ -144,8 +144,8 @@ public void runProtected(final Test test, Protectable p)
    addFailure(test, e);
    } catch (ThreadDeath e) { // don't catch ThreadDeath by accident
        throw e;
-    } catch (Throwable e) {
-        addError(test, e);
+    } catch (Throwable t) {
```

kcooney on 25 Feb 2014

IMHO, for things like variable naming, local consistency is arguably more important than global consistency, so naming this `e` is best. If we make a global change, I would prefer to rename variables of type `Throwable` to `e`

mallamanis on 25 Feb 2014

You are probably right. Indeed the tool seems confused about the convention used. It had suggested multiple changes of the form `Throwable t -> Throwable e`, but I thought that this was a false positive (e.g. in `org.junit.rules.TestWatcher` and `org.junit.tests.experimental.rules.TestWatcherTest`).

I understand that making a global change would be daunting and possibly controversial, would like me to submit such a pull request?

stefanbirkner on 25 Feb 2014

Such a pull request would be nice.



stefanbirkner commented on 25 Feb 2014

Your first commit (renaming "result" to "suite") is nice and I would like to merge it. But renaming "oldOut" is not good, because it is a more meaningful name than "oldPrintStream". Renaming "e" to "t" is no improvement, because we should consistently use `e`.

Could you please create a pull request with the first commit only.



mallamanis commented on 25 Feb 2014

@stefanbirkner thanks for the comment. It seems that you are right about the "e" to "t" renaming. Sorry for not noticing that before.

Concerning the `oldPrintStream`, I too like `oldOut` more. The tools suggested this renaming because it was used in a similar context in `org.junit.tests.running.core.MainRunner` (lines 276 and 288). Do you think that renaming that variable (`oldPrintStream` in `org.junit.tests.running.core.MainRunner`) would

Projects

None yet

Labels

None yet

Milestone

4.12

Assignees

No one assigned

3 participants

Notifications

You're receiving notifications because you were mentioned.

☐ Allow edits from maintainers.



pull request comment on 25 Feb 2014

@stefanbirkner thanks for the comment. It seems that you are right about the "e" to "t" renaming. Sorry for not noticing that before.

Concerning the `oldPrintStream`, I too like `oldOut` more. The tools suggested this renaming because it was used in a similar context in `org.junit.tests.running.core.MainRunner` (lines 276 and 288). Do you think that renaming that variable (`oldPrintStream` in `org.junit.tests.running.core.MainRunner`) would make any sense? Obviously, we can always just drop the `oldPrintStream` change.



stefanbirkner commented on 25 Feb 2014

Renaming the variable in `org.junit.tests.running.core.MainRunner` makes sense. 👍



mallamanis added a commit to mallamanis/junit that referenced this pull request on 25 Feb 2014

🔗 Renamed `oldPrintStream` to `oldOut` as discussed in #834

904b18a



mallamanis added a commit to mallamanis/junit that referenced this pull request on 25 Feb 2014

🔗 Renamed all Throwables "t" to "e" as discussed in #834 🗨️

8ca99c9



mallamanis commented on 25 Feb 2014

I've reverted the changes and made new renamings as we discussed. What do you think?



stefanbirkner commented on 26 Feb 2014

LGTM. @mallamanis do you have experience with Git? It would be nice if you rebase your commits to three commits (`Throwable`, `System.out`, `suite`).



mallamanis added some commits on 25 Feb 2014

🔗 Renamed "`oldPrintStream`" to "`oldOut`" to make the naming slightly

2745d01

🔗 Renamed all Throwables "t" to "e" for naming consistency.

b064a27



mallamanis commented on 26 Feb 2014

@stefanbirkner I think I've done that now. Let me know, if I need to make any other changes.



stefanbirkner merged commit `8a63df4` into `junit-team:master` on 26 Feb 2014



stefanbirkner added this to the 4.12 milestone on 26 Feb 2014



stefanbirkner commented on 26 Feb 2014

Everything is fine now. Thank you.

Variable renamings to reduce unnecessary variable naming diversity #5075

 **Closed** mallamanis wants to merge 3 commits into elastic:master from mallamanis:renamingSuggestions

 Conversation 1  Commits 3  Files changed 6 +42 -42









mallamanis commented on 10 Feb 2014

I've been working on a research machine learning-based tool (link: <http://groups.inf.ed.ac.uk/naturalize/>) tool that analyzes source code identifiers and makes suggestions for renaming them. The goal is to reduce unnecessary diversity in variable naming and improve code readability. This pull request is only a small sample of the suggestions made for elasticsearch.

No functional changes were made in any of the commits

 mallamanis added some commits on 10 Feb 2014

-   Renamed the ImmutableBlobContainer container to blobContainer. ✓ af0f713
-   Renamed XContentParser.Token named "t" to "token". ✓ 7d092c2
-   Renamed ClusterBlocks variable named "block" to "blocks". ✓ b847d2e

 javanna added **enhancement** **v2.0.0** **v1.2.0** labels on 7 Apr 2014

 javanna self-assigned this on 7 Apr 2014



javanna commented on 7 Apr 2014

elastic member

Merged, thanks!

 javanna closed this on 7 Apr 2014

 clintongormley added the **:Internal** label on 7 Jun 2015

Projects

None yet

Labels

Milestone

No milestone

Assignees

 javanna

3 participants



Notifications

You're receiving notifications because you were assigned.

☐ Allow edits from maintainers.

Renaming variables for codebase consistency #454

Merged **cketti** merged 5 commits into `k9mail:master` from `mallamanis:master` on 1 Mar 2014

Conversation **3** Commits **5** Files changed **5** +23 -23



mallamanis commented on 26 Feb 2014

I've been working on a research machine learning-based tool (link: <http://groups.inf.ed.ac.uk/naturalize/>) tool that analyzes source code identifiers and makes suggestions for renaming them. The goal is to reduce unnecessary diversity in variable naming and improve code readability. This pull request is only a small sample of the suggestions made for k-9.

No functional changes were made in any of the commits.

- mallamanis** added some commits on 26 Feb 2014
- Renamed "usee" to "uee". `be2b3b1`
 - Renamed "s" to "sizeParam". `2df2058`
 - Renamed "local_folder" to "localFolder". `6075add`
 - Renamed "tokens" to "tokenizer". `cfeed40`
 - Renamed "identityi" to "identity". `c17d032`



obra commented on 26 Feb 2014 K-9 Mail member

Thanks! The concept is very cool. s to sizeParam in the sample seems a little odd..

...



mallamanis commented on 26 Feb 2014

Thanks **@obra** ! Naturalize suggested `s` to `sizeParam` because a `String` object was named like that when used in a similar context in `src/com/fsck/k9/view/AttachmentView.java` (lines 142, 143, 145).

I could rename both instances to `s` or just drop that renaming. What do you think?



cketi commented on 28 Feb 2014 K-9 Mail member

"sizeParam" looks good enough to me.

@mallamanis: Thanks a lot! I'll try to get this merged during the weekend.

cketi merged commit `bf9264d` into `k9mail:master` on 1 Mar 2014

Projects
None yet

Labels
None yet

Milestone
No milestone

Assignees
No one assigned

3 participants

Notifications
You're receiving notifications because you were mentioned.

☐ Allow edits from maintainers.

Renaming variables for codebase consistency #1400

Merged **badlogic** merged 5 commits into `libgdx:master` from `mallamanis:master` on 26 Feb 2014

Conversation **2** Commits **5** Files changed **5**

+14 -14



mallamanis commented on 26 Feb 2014

I've been working on a research machine learning-based tool (link: <http://groups.inf.ed.ac.uk/naturalize/>) tool that analyzes source code identifiers and makes suggestions for renaming them. The goal is to reduce unnecessary diversity in variable naming and improve code readability. This pull request is only a small sample of the suggestions made for libgdx.

No functional changes were made in any of the commits.

- mallamanis** added some commits on 26 Feb 2014
- Renamed "i" (15.46%) to "index" (31.34%). The naturalize tool detected 35301ce
 - Renamed "value" (18.55%) to "scalar" (51.13%). The naturalize tool b522d52
 - Minor changes in JavaDoc 46d4393
 - Renamed "vector" (7.51%) to "point" (64.23%). The naturalize tool 5d6dffc
 - Renamed "scale" (24.02%) to "scaleXY" (75.98%). The naturalize tool 33726e2



sinistersnare commented on 26 Feb 2014

Very interesting project, and thanks for the contribution!

I've always wanted to spend a good amount of time using findbugs with libgdx, there's a ton of recommendations it provides.



badlogic commented on 26 Feb 2014 libgdx member

Wow, that's a pretty cool tool!

badlogic closed this on 26 Feb 2014

badlogic reopened this on 26 Feb 2014

badlogic merged commit `3d5040f` into `libgdx:master` on 26 Feb 2014

Projects
None yet

Labels
None yet

Milestone
No milestone

Assignees
No one assigned

3 participants

Notifications

You're receiving notifications because you authored the thread.

☐ Allow edits from maintainers.

Bibliography

- Abebe, Surafel Lemma, Haiduc, Sonia, Tonella, Paolo, and Marcus, Andrian. The effect of lexicon bad smells on concept location in source code. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2011.
- Abebe, Surafel Lemma, Arnaoudova, Venera, Tonella, Paolo, Antoniol, Giuliano, and Gueheneuc, Y. Can lexicon bad smells improve fault prediction? In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2012.
- Acharya, Mithun, Xie, Tao, Pei, Jian, and Xu, Jun. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.
- Adams, Edward N. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 1984.
- Aggarwal, Charu C and Han, Jiawei. *Frequent pattern mining*. Springer, 2014.
- Aggarwal, Karan, Salameh, Mohammad, and Hindle, Abram. Using machine translation for converting Python 2 to Python 3 code. Technical report, 2015.
- AirBnb. Airbnb JavaScript Style Guide. <https://github.com/airbnb/javascript>, 2015. Visited Sep 2016.
- Alemi, Alex A, Chollet, Francois, Irving, Geoffrey, Szegedy, Christian, and Urban, Josef. DeepMath—Deep sequence models for premise selection. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2016.
- Allamanis, Miltiadis and Sutton, Charles. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013a.
- Allamanis, Miltiadis and Sutton, Charles. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013b.
- Allamanis, Miltiadis and Sutton, Charles. Mining idioms from source code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.

- Allamanis, Miltiadis, Barr, Earl T, Bird, Christian, and Sutton, Charles. Learning natural coding conventions. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.
- Allamanis, Miltiadis, Barr, Earl T, Bird, Christian, and Sutton, Charles. Suggesting accurate method and class names. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015a.
- Allamanis, Miltiadis, Tarlow, Daniel, Gordon, Andrew, and Wei, Yi. Bimodal modelling of source code and natural language. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015b.
- Allamanis, Miltiadis, Barr, Earl T., Bird, Christian, Devanbu, Premkumar, Marron, Mark, and Sutton, Charles. Mining semantic loop idioms from Big Code. Technical report, November 2016a. URL <https://www.microsoft.com/en-us/research/publication/mining-semantic-loop-idioms-big-code/>.
- Allamanis, Miltiadis, Barr, Earl T., Just, René, and Sutton, Charles. Tailored mutants fit bugs better. 2016b. URL <http://arxiv.org/abs/1611.02516>.
- Allamanis, Miltiadis, Chanthirasegaran, Pankajan, Kohli, Pushmeet, and Sutton, Charles. Learning continuous semantic representations of symbolic expressions. 2016c. URL <http://arxiv.org/abs/1611.01423>.
- Allamanis, Miltiadis, Peng, Hao, and Sutton, Charles. A convolutional attention network for extreme summarization of source code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016d.
- Amann, Sven, Proksch, Sebastian, Nadi, Sarah, and Mezini, Mira. A study of Visual Studio usage in practice. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.
- Andereessen, Mark. Why software is eating the world. The Wall Street Journal. Available online: <http://on.wsj.com/o6yIeE>, August 2011. URL <http://online.wsj.com/article/SB10001424053111903480904576512250915629460.html>.
- Anquetil, Nicolas and Lethbridge, Timothy. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 4, 1998.
- Anquetil, Nicolas and Lethbridge, Timothy C. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 1999.
- Arnaoudova, Venera, Di Penta, Massimiliano, Antoniol, Giuliano, and Gueheneuc, Yann-Gael. A new family of software anti-patterns: Linguistic anti-patterns. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2013.

- Arnaoudova, Venera, Eshkevari, Laleh Mousavi, Penta, Massimiliano Di, Oliveto, Rocco, Antoniol, Giuliano, and Guéhéneuc, Yann-Gaël. REPENT: analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering (TSE)*, 2014.
- Arnaoudova, Venera, Penta, Massimiliano Di, and Antoniol, Giuliano. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering (ESEM)*, 2015.
- Arthur, Charles. Apple's SSL iPhone vulnerability: How did it happen, and what next? bit.ly/1bJ7aSa, 2014. Visited Jun 2016.
- Association, Motor Industry Software Reliability et al. MISRA-C 2012: Guidelines for the use of the C language in critical systems. *ISBN 9781906400118*, 2012.
- Ayewah, Nathaniel, Pugh, William, Morgenthaler, J David, Penix, John, and Zhou, YuQian. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007.
- Ba, Jimmy Lei, Kiros, Jamie Ryan, and Hinton, Geoffrey E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- Bacchelli, Alberto. Mining challenge 2013: StackOverflow. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013.
- Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.
- Baker, Brenda S. A program for identifying duplicated code. *Computing Science and Statistics*, 1993.
- Barr, Earl T., Bird, Christian, and Marron, Mark. Collecting a heap of shapes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2013.
- Barr, Earl T, Brun, Yuriy, Devanbu, Premkumar, Harman, Mark, and Sarro, Federica. The plastic surgery hypothesis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.
- Barthe, Gilles, Crespo, Juan Manuel, Gulwani, Sumit, Kunz, Cesar, and Marron, Mark. From relational verification to SIMD loop synthesis. In *PPoPP*, 2013.
- Basit, Hamid Abdul and Jarzabek, Stan. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering (TSE)*, 2009.
- Bass, Len. *Software architecture in practice*. Pearson, 2007.
- Beck, Kent. *Implementation patterns*. Pearson Education, 2007.

- Beck, Kent and Cunningham, Ward. Using pattern languages for object-oriented programs. 1987.
- Bengio, Samy, Vinyals, Oriol, Jaitly, Navdeep, and Shazeer, Noam. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.
- Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Janvin, Christian. A neural probabilistic language model. *Journal of Machine Learning Research (JMLR)*, 2003.
- Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, and Engler, Dawson. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 2010.
- Bhatia, Sahil and Singh, Rishabh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.
- Bhoopchand, Avishkar, Rocktäschel, Tim, Barr, Earl T., and Riedel, Sebastian. Learning Python code suggestion with a sparse pointer network. <https://openreview.net/pdf?id=r1kQkVFgl>, 2016.
- Bichsel, Benjamin, Raychev, Veselin, Tsankov, Petar, and Vechev, Martin. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- Bielik, Pavol, Raychev, Veselin, and Vechev, Martin. Programming with “big code”: Lessons, techniques and applications. In *LIPICs-Leibniz International Proceedings in Informatics*, 2015.
- Bielik, Pavol, Raychev, Veselin, and Vechev, Martin. PHOG: Probabilistic model for code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.
- Biggerstaff, Ted J, Mitbender, Bharat G, and Webster, Dallas. The concept assignment problem in program understanding. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1993.
- Binkley, D., Davis, M., Lawrie, D., and Morrell, C. To CamelCase or Under_score. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2009.
- Binkley, Dave, Hearn, Matthew, and Lawrie, Dawn. Improving identifier informativeness using part of speech information. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2011.
- Bod, Rens, Scha, Remko, and Sima'an, Khalil. *Data-oriented parsing*. Center for the Study of Language and Information — Studies in Computational Linguistics. University of Chicago Press.

- Boogerd, Cathal and Moonen, Leon. Assessing the value of coding standards: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2008.
- Bourque, Pierre, Fairley, Richard E, et al. *Guide to the software engineering body of knowledge: Version 3.0*. IEEE Computer Society Press, 2014.
- Brants, Thorsten, Popat, Ashok C, Xu, Peng, Och, Franz J, and Dean, Jeffrey. Large language models in machine translation. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*. Citeseer, 2007.
- Brooks, Frederick P. *The Mythical Man-Month*. Addison-Wesley Reading, 1975.
- Broy, Manfred, Deußenböck, Florian, and Pizka, Markus. A holistic approach to software quality at work. In *Proc. 3rd World Congress for Software Quality (3WCSQ)*, 2005.
- Bruch, Marcel, Monperrus, Martin, and Mezini, Mira. Learning from examples to improve code completion systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.
- Buschmann, Frank, Henney, Kevin, and Schmidt, Douglas C. *Pattern-oriented software architecture, on patterns and pattern languages*. John Wiley & Sons, 2007.
- Buse, Raymond PL and Weimer, Westley. Synthesizing API usage examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.
- Buse, Raymond PL and Weimer, Westley R. Learning a metric for code readability. *IEEE Transactions on Software Engineering (TSE)*, 2010.
- Butler, S., Wermelinger, M., Yu, Yijun, and Sharp, H. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.
- Butler, Simon, Wermelinger, Michel, Yu, Yijun, and Sharp, Helen. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2009.
- Butler, Simon, Wermelinger, Michel, Yu, Yijun, and Sharp, Helen. Mining Java class naming conventions. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2011.
- Campbell, Joshua Charles, Hindle, Abram, and Amaral, José Nelson. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014.
- Caprile, B. and Tonella, P. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2000.

- Cerulo, Luigi, Di Penta, Massimiliano, Bacchelli, Alberto, Ceccarelli, Michele, and Canfora, Gerardo. Irish: A hidden Markov model to detect coded information islands in free text. *Science of Computer Programming*, 2015.
- Chandola, Varun, Banerjee, Arindam, and Kumar, Vipin. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 2009.
- Chen, Stanley F and Goodman, Joshua. An empirical study of smoothing techniques for language modeling. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 1996.
- Cherem, Sigmund and Rugina, Radu. A practical escape and effect analysis for building lightweight method summaries. In *Proceedings of the 16th International Conference on Compiler Construction*, 2007.
- Cho, Kyunghyun, van Merriënboer, Bart, Bahdanau, Dzmitry, and Bengio, Yoshua. On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, 2014.
- Chopra, Sumit, Hadsell, Raia, and LeCun, Yann. Learning a similarity metric discriminatively, with application to face verification. In *CVPR*, 2005.
- Chuan, Shi. JavaScript Patterns Collection. <http://shichuan.github.io/javascript-patterns/>, 2014. Visited Sep 2016.
- Cohn, Trevor, Blunsom, Phil, and Goldwater, Sharon. Inducing tree-substitution grammars. *Journal of Machine Learning Research (JMLR)*, 2010.
- Collobert, Ronan and Weston, Jason. A unified architecture for natural language processing: deep neural networks with multitask learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.
- Corbi, Thomas A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 1989.
- Corley, Christopher S, Damevski, Kostadin, and Kraft, Nicholas A. Exploring the use of deep learning for feature location. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015.
- Cowan, Nelson. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 2001.
- Dam, Hoa Khanh, Tran, Truyen, and Pham, Trang. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.
- De Lucia, Andrea, Di Penta, Massimiliano, Oliveto, Rocco, Panichella, Annibale, and Panichella, Sebastiano. Using IR methods for labeling source code artifacts: Is it worthwhile? In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2012.

- Deissenboeck, Florian and Pizka, Markus. Deïßenböck, and consistent naming. *Software Quality Journal*, 2006.
- Devanbu, Premkumar. New initiative: the naturalness of software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.
- Dong, Jing, Zhao, Yajing, and Peng, Tu. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 2009.
- Dyer, Chris, Ballesteros, Miguel, Ling, Wang, Matthews, Austin, and Smith, Noah A. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.
- Eclipse-Contributors. Eclipse JDT. eclipse.org/jdt, 2014. Visited Sep 2016.
- Eddy, Brian P, Robinson, Jeffrey A, Kraft, Nicholas A, and Carver, Jeffrey C. Evaluating source code summarization techniques: Replication and expansion. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2013.
- Eshkevari, Laleh M, Arnaoudova, Venera, Di Penta, Massimiliano, Oliveto, Rocco, Guéhéneuc, Yann-Gaël, and Antoniol, Giuliano. An exploratory study of identifier renamings. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2011.
- Fast, Ethan, Steffee, Daniel, Wang, Lucy, Brandt, Joel R, and Bernstein, Michael S. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the Annual ACM Conference on Human Factors in Computing Systems*, 2014.
- Fowkes, Jaroslav and Sutton, Charles. Parameter-free probabilistic API mining at github scale. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.
- Fowkes, Jaroslav, Ranca, Razvan, Allamanis, Miltiadis, Lapata, Mirella, and Sutton, Charles. Autofolding for source code summarization. *arXiv preprint arXiv:1403.4503*, 2014.
- Fowkes, Jaroslav, Chanthirasegaran, Pankajan, Ranca, Razvan, Allamanis, Miltiadis, Lapata, Mirella, and Sutton, Charles. TASSAL: autofolding for source code summarization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.
- Fowler, Martin. *Refactoring: improving the design of existing code*. 2009.
- Franks, Christine, Tu, Zhaopeng, Devanbu, Premkumar, and Hellendoorn, Vincent. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.
- Gabel, Mark and Su, Zhendong. A study of the uniqueness of source code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2010.

- Gabel, Mark Gregory. *Inferring Programmer Intent and Related Errors from Software*. PhD thesis, University of California, 2011.
- Gallardo-Valencia, Rosalva E and Elliott Sim, Susan. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, 2009.
- Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object Oriented Software*. 1995.
- Gelman, Andrew, Carlin, John, Stern, Hal, Dunson, David, Vehtari, Aki, and Rubin, Donald. *Bayesian data analysis*. CRC Press, 2013.
- Gershman, Samuel J and Blei, David M. A tutorial on Bayesian nonparametric models. *Journal of Mathematical Psychology*, 2012.
- Glassman, Elena L, Scott, Jeremy, Singh, Rishabh, Guo, Philip J, and Miller, Robert C. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2015.
- Google. Google C++ style guide. <https://google.github.io/styleguide/cppguide.html>, 2010. Visited Sep 2016.
- Gousios, Georgios and Spinellis, Diomidis. GHTorrent: GitHub’s data from a firehose. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2012.
- Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.
- Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.
- Gruau, Frédéric, Ratajszczak, Jean-Yves, and Wiber, Gilles. A neural compiler. *Theoretical Computer Science*, 1995.
- Gruska, Natalie, Wasylkowski, Andrzej, and Zeller, Andreas. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010.
- Gu, Xiaodong, Zhang, Hongyu, Zhang, Dongmei, and Kim, Sunghun. Deep API learning. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.
- Gulwani, Sumit and Marron, Mark. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.
- Gupta, Rahul, Pal, Soham, Kanade, Aditya, and Shevade, Shirish. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Conference of Artificial Intelligence (AAAI)*, 2017.

- Gupta, Samir, Malik, Sana, Pollock, Lori, and Vijay-Shanker, K. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2013.
- Gutmann, Michael U and Hyvärinen, Aapo. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research (JMLR)*, 2012.
- Gvero, Tihomir and Kuncak, Viktor. Synthesizing java expressions from free-form queries. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2015.
- Gyori, Alex, Franklin, Lyle, Dig, Danny, and Lahoda, Jan. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2013.
- Haiduc, Sonia, Aponte, Jairo, and Marcus, Andrian. Supporting program comprehension with source code summarization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010a.
- Haiduc, Sonia, Aponte, Jairo, Moreno, Laura, and Marcus, Andrian. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2010b.
- Hatton, Les. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 2004.
- He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, 2015.
- Hellendoorn, Vincent J, Devanbu, Premkumar T, and Bacchelli, Alberto. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2015.
- Hendrix, Dean, Cross, JH, and Maghsoodloo, Saeed. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transactions on Software Engineering (TSE)*, 2002.
- Hettinger, Raymond. Transforming code into beautiful, idiomatic Python. <https://www.youtube.com/watch?v=0SGv2VnC0go>, 2013. Visited Sep 2016.
- Hindle, Abram, Godfrey, Michael W., and Holt, Richard C. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 2009.
- Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

- Hinton, Geoffrey, Srivastava, Nitish, and Swersky, Kevin. Neural networks for machine learning. *Online Course at coursera. org, Lecture, 6, 2012.*
- Hinton, Geoffrey E. Distributed representations. 1984.
- Hjort, Nils Lid. *Bayesian Nonparametrics*. Cambridge University Press, 2010.
- Hohpe, Gregor and Woolf, Bobby. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.
- Holmes, Reid, Walker, Robert J, and Murphy, Gail C. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*, 2005.
- Holmes, Reid, Walker, Robert J, and Murphy, Gail C. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering (TSE)*, 2006.
- Høst, Einar W. and Østvold, Bjarte M. Debugging method names. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.
- Hsiao, Chun-Hung, Cafarella, Michael, and Narayanasamy, Satish. Using web corpus statistics for program analysis. In *ACM SIGPLAN Notices*, 2014.
- Iyer, Srinivasan, Konstantinos, Ioannis, Cheung, Alvin, and Zettlemoyer, Luke. Summarizing source code using a neural attention model. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.
- Java Idioms Editors. Java idioms. <http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html>, 2014. Visited Sep 2016.
- JetBrains. High-speed coding with custom live templates. <https://blog.jetbrains.com/webide/2012/10/high-speed-coding-with-custom-live-templates/>, 2014. Visited Sep 2016.
- JetBrains. ReSharper. <http://www.jetbrains.com/resharper/>, 2015. URL <http://www.jetbrains.com/resharper/>.
- Jiang, Lingxiao, Misherghi, Ghassan, Su, Zhendong, and Glondu, Stephane. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007.
- Jiménez, Aída, Berzal, Fernando, and Cubero, Juan-Carlos. Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 2010.
- Joshi, Aravind K and Schabes, Yves. Tree-adjoining grammars. In *Handbook of Formal Languages*. Springer, 1997.
- Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

- Jurafsky, Dan. *Speech & Language Processing*. Pearson Education, 2000.
- Kaiser, Łukasz and Sutskever, Ilya. Neural GPUs learn algorithms. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- Kalchbrenner, Nal, Grefenstette, Edward, and Blunsom, Phil. A convolutional neural network for modelling sentences. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2014.
- Karaivanov, Svetoslav, Raychev, Veselin, and Vechev, Martin. Phrase-based statistical translation of programming languages. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014.
- Karpathy, Andrej, Johnson, Justin, and Li, Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.
- Kiros, Ryan, Zemel, R, and Salakhutdinov, Ruslan. Multimodal neural language models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2013.
- Kontogiannis, Kostas A, DeMori, Renator, Merlo, Ettore, Galler, M, and Bernstein, Morris. Pattern matching for clone and concept detection. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. Springer, 1996.
- Kremenek, Ted, Ng, Andrew Y, and Engler, Dawson R. A factor graph model for software bug finding. In *Proceedings of the International Joint Conference on Artificial intelligence (IJCAI)*, 2007.
- Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.
- Kuhn, Roland and De Mori, Renato. A cache-based natural language model for speech recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 1990.
- Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.
- Kushman, Nate and Barzilay, Regina. Using semantic unification to generate regular expressions from natural language. In *Proceedings of Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2013.
- Kuzborskij, Ilja. Large-scale pattern mining of computer program source code. Master's thesis, University of Edinburgh, 2011.
- Langley, Adam. Apple's SSL/TLS bug. bit.ly/MMvx6b, 2014. Visited Jun 2016.
- Lau, Tessa. *Programming by demonstration: a machine learning approach*. PhD thesis, University of Washington, 2001.

- Lawrie, Dawn, Feild, Henry, and Binkley, David. Syntactic identifier conciseness and consistency. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2006a.
- Lawrie, Dawn, Morrell, Christopher, Feild, Henry, and Binkley, David. What's in a name? A study of identifiers. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2006b.
- Lawrie, Dawn, Feild, Henry, and Binkley, David. An empirical study of rules for well-formed identifiers: Research articles. *Journal of Software Maintenance Evolution: Research and Practice*, 2007.
- Le, Quoc V and Mikolov, Tomas. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.
- LeCun, Y., Boser, B., Denker, J. S., Howard, R. E., Hubbard, W., Jackel, L. D., and Henderson, D. Proceedings of the annual conference on neural information processing systems (nips). chapter Handwritten Digit Recognition with a Back-propagation Network. 1990.
- LeCun, Yann, Bottou, Leon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.
- Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. Gated graph sequence neural networks. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- Liang, Percy, Jordan, Michael I, and Klein, Dan. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010a.
- Liang, Percy, Jordan, Michael I, and Klein, Dan. Type-based MCMC. In *Human Language Technologies: Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL)*, 2010b.
- Liblit, Ben, Naik, Mayur, Zheng, Alice X, Aiken, Alex, and Jordan, Michael I. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, 2005.
- Liblit, Ben, Begel, Andrew, and Sweetser, Eve. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, 2006.
- Ling, Wang, Grefenstette, Edward, Hermann, Karl Moritz, Kocisky, Tomas, Senior, Andrew, Wang, Fumin, and Blunsom, Phil. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.
- Liu, Han. Towards better program obfuscation: optimization via language models. In *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016.

- Livshits, Benjamin and Zimmermann, Thomas. Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, 2005.
- Maas, Andrew L., Hannun, Awni Y., and Ng, Andrew Y. Rectified linear units improve restricted Boltzmann machines. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing*, 2013.
- Maddison, Chris and Tarlow, Daniel. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.
- Mangal, Ravi, Zhang, Xin, Nori, Aditya V, and Naik, Mayur. A user-guided approach to program analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.
- Manning, Christopher D., Raghavan, Prabhakar, and Schütze, Hinrich. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- Marguerie, Fabrice, Eichert, Steve, and Wooley, Jim. *LINQ in Action*. Manning, 2008.
- Marron, Mark, Stefanovic, Darko, Kapur, Deepak, and Hermenegildo, Manuel V. Identification of heap-carried data dependence via explicit store heap models. In *Languages and Compilers for Parallel Computing*, 2008.
- Martin, Robert C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.
- McCabe, Thomas J. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, 1976.
- McConnell, Steve. *Code Complete*. Microsoft Press, 2004.
- Mcmillan, Collin, Poshyvanyk, Denys, Grechanik, Mark, Xie, Qing, and Fu, Chen. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2013.
- Meijer, Erik. The world according to LINQ. *Queue*, 2011.
- Menon, Aditya, Tamuz, Omer, Gulwani, Sumit, Lampson, Butler, and Kalai, Adam. A machine learning framework for programming by example. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013.
- Mens, Kim and Lozano, Angela. Source code-based recommendation systems. In *Recommendation Systems in Software Engineering*. Springer, 2014.
- Microsoft Research. High-speed coding with Custom Live Templates. <http://research.microsoft.com/apps/video/dl.aspx?id=208961>, 2014. Visited Sep 2016.
- Mikolov, Tomas, Chen, Kai, Corrado, Greg, and Dean, Jeffrey. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

- Mikolov, Tomas, Yih, Wen-tau, and Zweig, Geoffrey. Linguistic regularities in continuous space word representations. In *Proceedings of Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2013b.
- Miller, George A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 1956.
- Mishne, Alon, Shoham, Sharon, and Yahav, Eran. Typestate-based semantic code search over partial programs. In *ACM SIGPLAN Notices*, 2012.
- Mnih, Andriy and Hinton, Geoffrey. Three new graphical models for statistical language modelling. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2007.
- Mnih, Andriy and Teh, Yee W. A fast and simple algorithm for training neural probabilistic language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2012.
- Mnih, Volodymyr, Heess, Nicolas, Graves, Alex, et al. Recurrent models of visual attention. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2014.
- Mou, Lili, Li, Ge, Zhang, Lu, Wang, Tao, and Jin, Zhi. Convolutional neural networks over tree structures for programming language processing. In *Conference on Artificial Intelligence*, 2016.
- Movshovitz-Attias, Dana and Cohen, William W. KB-LDA: Jointly learning a knowledge base of hierarchy, relations, and facts. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015.
- Movshovitz-Attias, Dana, Cohen, WW, and W. Cohen, William. Natural language models for predicting programming comments. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2013.
- Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
- Murphy-Hill, Emerson, Parnin, Chris, and Black, Andrew P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering (TSE)*, 2012.
- Nagappan, Nachiappan and Ball, Thomas. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering (ESEM)*, 2007.
- Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.
- Nazara, Najam, Hua, Yan, and Jianga, He. Summarizing software artifacts: Classifications, methods, and applications. *Submitted to Knowledge-Based Systems*, 2015.

- Neelakantan, Arvind, Le, Quoc V, and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.
- Neto, João Pedro, Siegelmann, Hava T, and Costa, J Félix. Symbolic processing in neural networks. *Journal of the Brazilian Computer Society*, 2003.
- Neubig, Graham. Survey of methods to generate natural language from source code. <http://www.languageandcode.org/nlse2015/neubig15nlse-survey.pdf>, 2016.
- Nguyen, Anh Tuan and Nguyen, Tien N. Graph-based statistical language model for code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.
- Nguyen, Anh Tuan, Nguyen, Tung Thanh, and Nguyen, Tien N. Lexical statistical machine translation for language migration. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2013a.
- Nguyen, Anh Tuan, Nguyen, Tung Thanh, and Nguyen, Tien N. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015.
- Nguyen, Anh Tuan, Nguyen, Hoan Anh, and Nguyen, Tien N. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2016a.
- Nguyen, Trong Duc, Nguyen, Anh Tuan, and Nguyen, Tien N. Mapping API elements for code migration with vector representations. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016b.
- Nguyen, Tung Thanh, Nguyen, Hoan Anh, Pham, Nam H, Al-Kofahi, Jafar M, and Nguyen, Tien N. Graph-based mining of multiple object usage patterns. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.
- Nguyen, Tung Thanh, Nguyen, Anh Tuan, Nguyen, Hoan Anh, and Nguyen, Tien N. A statistical semantic language model for source code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013b.
- Niu, Haoran, Keivanloo, Iman, and Zou, Ying. Learning to rank code examples for code search engines. *Empirical Software Engineering (ESEM)*, 2016.
- Oda, Yusuke, Fudaba, Hiroyuki, Neubig, Graham, Hata, Hideaki, Sakti, Sakriani, Toda, Tomoki, and Nakamura, Satoshi. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015.

- Oh, Hakjoo, Yang, Hongseok, and Yi, Kwangkeun. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2015.
- Ohba, Masaru and Gondow, Katsuhiko. Toward mining concept keywords from identifiers in large software projects. In *ACM SIGSOFT Software Engineering Notes*, 2005.
- Omar, Cyrus. Structured statistical syntax tree prediction. In *Proceedings of the Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, 2013.
- Oracle. Code conventions for the Java programming language. <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>, 1999. Visited June, 2016.
- Orbanz, P. and Teh, Y. W. Bayesian nonparametric models. In *Encyclopedia of Machine Learning*. Springer, 2010.
- Papineni, Kishore, Roukos, Salim, Ward, Todd, and Zhu, Wei-Jing. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2002.
- Parr, Terence and Vinju, Jurgin. Technical report: Towards a universal code formatter through machine learning. *arXiv preprint arXiv:1606.08866*, 2016.
- Pham, Hung Viet, Vu, Phong Minh, Nguyen, Tung Thanh, et al. Learning API usages from bytecode: a statistical approach. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.
- Piech, Chris, Huang, Jonathan, Nguyen, Andy, Phulsuksombati, Mike, Sahami, Mehran, and Guibas, Leonidas J. Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.
- Post, Matt and Gildea, Daniel. Bayesian learning of a tree substitution grammar. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2009.
- Proksch, Sebastian, Lerch, Johannes, and Mezini, Mira. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015.
- Proksch, Sebastian, Amann, Sven, Nadi, Sarah, and Mezini, Mira. Evaluating the evaluations of code recommender systems: a reality check. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2016.
- Ratiu, Daniel and Deisenböck, Florian. From reality to programs and (not quite) back again. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2007.

- Ray, Baishakhi, Hellendoorn, Vincent, Godhane, Saheel, Tu, Zhaopeng, Bacchelli, Alberto, and Devanbu, Premkumar. On the naturalness of buggy code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.
- Raychev, Veselin, Vechev, Martin, and Yahav, Eran. Code completion with statistical language models. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*, 2014.
- Raychev, Veselin, Vechev, Martin, and Krause, Andreas. Predicting program properties from “big code”. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2015.
- Raychev, Veselin, Bielik, Pavol, Vechev, Martin, and Krause, Andreas. Learning programs from noisy data. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2016.
- Recommenders, Eclipse. Eclipse SnipMatch. <http://www.eclipse.org/recommenders/manual/#snipmatch>, 2014. Visited Sep 2016.
- Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- Riedel, Sebastian, Bošnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable Forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016.
- Rigby, Peter C. and Bird, Christian. Convergent software peer review practices. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.
- Robillard, Martin, Walker, Robert, and Zimmermann, Thomas. Recommendation systems for software engineering. *Software, IEEE*, 2010.
- Robillard, Martin P, Maalej, Walid, Walker, Robert J, and Zimmermann, Thomas. *Recommendation systems in software engineering*. Springer, 2014.
- Rossum, Guido van, Warsaw, Barry, and Coghlan, Nick. PEP 8—Style Guide for Python Code. <http://www.python.org/dev/peps/pep-0008/>, 2013. Visited June, 2015.
- Roy, Chanchal K, Cordy, James R, and Koschke, Rainer. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.
- Roy, Chanchal Kumar and Cordy, James R. A survey on software clone detection research. Technical report, Queen’s University at Kingston, Ontario, 2007.
- Rubin, Julia and Chechik, Marsha. A survey of feature location techniques. In *Domain Engineering*. Springer, 2013.
- Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

- Sadowski, Caitlin, Stolee, Kathryn T, and Elbaum, Sebastian. How developers search for code: a case study. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.
- Saraiva, Juliana, Bird, Christian, and Zimmermann, Thomas. Products, developers, and milestones: how should i build my n-gram language model. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2015.
- Sethuraman, Jayaram. A constructive definition of Dirichlet priors. Technical report, DTIC Document, 1991.
- Sharma, Abhishek, Tian, Yuan, and Lo, David. NIRMAL: Automatic identification of software relevant tweets leveraging language model. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.
- Siegelmann, Hava T. Neural programming language. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1994.
- Siegmund, Janet, Kästner, Christian, Apel, Sven, Parnin, Chris, Bethmann, Anja, Leich, Thomas, Saake, Gunter, and Brechmann, André. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.
- Silva, Danilo, Tsantalis, Nikolaos, and Valente, Marco Tulio. Why we refactor? confessions of github contributors. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.
- Simonyi, Charles. Hungarian notation. [http://msdn.microsoft.com/en-us/library/aa260976\(VS.60\).aspx](http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx), 1999. Visited June, 2016.
- Singh, Rishabh and Gulwani, Sumit. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, 2015.
- Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.
- Socher, Richard, Pennington, Jeffrey, Huang, Eric H, Ng, Andrew Y, and Manning, Christopher D. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, 2011.
- Socher, Richard, Huval, Brody, Manning, Christopher D, and Ng, Andrew Y. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, 2012.
- Socher, Richard, Perelygin, Alex, Wu, Jean Y, Chuang, Jason, Manning, Christopher D, Ng, Andrew Y, and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*, 2013.

- Soloway, Elliot and Ehrlich, Kate. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering (TSE)*, 1984.
- Spinellis, Diomidis. *Code reading: the open source perspective*. Addison-Wesley Professional, 2003.
- Sridhara, Giriprasad. *Automatic generation of descriptive summary comments for methods in object-oriented programs*. University of Delaware, 2012.
- Sridhara, Giriprasad, Hill, Emily, Muppaneni, Divya, Pollock, Lori, and Vijay-Shanker, K. Towards automatically generating summary comments for Java methods. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2010.
- Sridhara, Giriprasad, Pollock, Lori, and Vijay-Shanker, K. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.
- Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.
- Strunk Jr, William and White, E.B. *The Elements of Style*. Macmillan, 3rd edition, 1979.
- Sălcianu, Alexandru and Rinard, Martin. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'05*, 2005.
- Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013.
- Takang, Armstrong A, Grubb, Penny A, and Macredie, Robert D. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 1996.
- Teh, Y. W. and Jordan, M. I. Hierarchical Bayesian nonparametric models with applications. In Hjort, N., Holmes, C., Müller, P., and Walker, S. (eds.), *Bayesian Nonparametrics: Principles and Practice*. Cambridge University Press, 2010.
- Termier, Alexandre, Rousset, Marie-Christine, and Sebag, Michèle. Treefinder: a first step towards XML data mining. In *International Conference on Data Mining (ICDM)*. IEEE, 2002.
- Thummalapenta, Suresh and Xie, Tao. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2007.

- Tu, Zhaopeng, Su, Zhendong, and Devanbu, Premkumar. On the localness of software. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.
- Uddin, Gias, Dagenais, Barthélémy, and Robillard, Martin P. Analyzing temporal API usage patterns. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.
- van der Maaten, Laurens and Hinton, Geoffrey. Visualizing data using t-SNE. *Journal of Machine Learning Research (JMLR)*.
- Velez, Martin, Qiu, Dong, Zhou, You, Barr, Earl T, and Su, Zhendong. A study of “wheat” and “chaff” in source code. *arXiv preprint arXiv:1502.01410*, 2015.
- Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.
- Waldron, Rick. Principles of Writing Consistent, Idiomatic JavaScript. <https://github.com/rwaldron/idiomatic.js/>, 2014. Visited Sep 2016.
- Wang, Jue, Dang, Yingnong, Zhang, Hongyu, Chen, Kai, Xie, Tao, and Zhang, Dongmei. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013.
- Wang, Song, Chollak, Devin, Movshovitz-Attias, Dana, and Tan, Lin. Bugram: bug detection with n-gram language models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2016a.
- Wang, Song, Liu, Taiyue, and Tan, Lin. Automatically learning semantic features for defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016b.
- Wang, Xiaoran, Pollock, Lori, and Vijay-Shanker, K. Automatic segmentation of method code into meaningful blocks to improve readability. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2011.
- Wang, Xin, Liu, Chang, Shin, Richard, Gonzalez, Joseph E., and Song, Dawn. Neural code completion. <https://openreview.net/pdf?id=rJbPBt9lg>, 2016c.
- Wasylkowski, Andrzej, Zeller, Andreas, and Lindig, Christian. Detecting object usage anomalies. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.
- White, Martin, Vendome, Christopher, Linares-Vásquez, Mario, and Poshyanyk, Denys. Toward deep learning software repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2015.

- White, Martin, Tufano, Michele, Vendome, Christopher, and Poshyvanyk, Denys. Deep learning code fragments for code clone detection. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2016.
- Wikibooks. More C++ idioms. http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms, 2013. Visited Sep 2016.
- Wikipedia. Coding Conventions. http://en.wikipedia.org/wiki/Coding_conventions.
- Williams, Christopher KI and Rasmussen, Carl Edward. Gaussian Processes for Machine Learning, 2006.
- Xie, Tao and Pei, Jian. MAPO: Mining API usages from open source repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2006.
- Xu, Haiying, Pickett, Christopher J. F., and Verbrugge, Clark. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE*, 2007.
- Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Cho, Kyunghyun, Courville, Aaron C, Salakhutdinov, Ruslan, Zemel, Richard S, and Bengio, Yoshua. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.
- Yadid, Shir and Yahav, Eran. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016.
- Yang, Jinlin, Evans, David, Bhardwaj, Deepali, Bhat, Thirumalesh, and Das, Manuvir. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.
- Young, H Peyton. The economics of convention. *The Journal of Economic Perspectives*, 1996.
- Zaki, Mohammed J. Efficiently mining frequent trees in a forest. In *Conference on Knowledge Discovery and Data Mining (KDD)*, 2002.
- Zaki, Mohammed Javeed. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2005.
- Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.
- Zaremba, Wojciech, Kurach, Karol, and Fergus, Rob. Learning to discover efficient mathematical identities. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2014.

- Zhang, Cheng, Yang, Juyuan, Zhang, Yi, Fan, Jing, Zhang, Xin, Zhao, Jianjun, and Ou, Peizhao. Automatic parameter recommendation for practical API usage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.
- Zhang, Hongyu, Jain, Anuj, Khandelwal, Gaurav, Kaushik, Chandrashekhar, Ge, Scott, and Hu, Wenxiang. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016.
- Zheng, Alice X, Jordan, Michael I, Liblit, Ben, Naik, Mayur, and Aiken, Alex. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2006.
- Zhong, Hao, Xie, Tao, Zhang, Lu, Pei, Jian, and Mei, Hong. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*. 2009.