Empirical Software Linguistics: An Investigation of Code Reviews, Recommendations and Faults

Version of August 8, 2015

Vincent Hellendoorn
Empirical Software Linguistics: An Investigation of Code Reviews, Recommendations and Faults

THESIS
submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in

COMPUTER SCIENCE
by

Vincent Hellendoorn
born in Delft, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Empirical Software Linguistics: An Investigation of Code Reviews, Recommendations and Faults

Author: Vincent Hellendoorn
Student id: 4091302
Email: V.J.Hellendoorn@student.tudelft.nl

Abstract

Communication is fundamental to human nature and underlies many of its successes as a species. In recent decades, the adoption of increasingly abstract software languages has supported many advances in computer science and software engineering. Although in many regards distinct from natural language, software language has proven surprisingly similar to it as well and has been studied successfully using natural language models.

Recent studies have investigated this “naturalness” property of software in relation to a variety of applications including code completion, fault detection, and language migration. In this thesis, based on three research papers, we investigate three main aspects of software naturalness. Firstly, we investigate the relation between perceived (un)naturalness of source code (according to the statistical model) and the reaction to such code by software developers. In open-source projects, we find that those contributions which contain code that (statistically speaking) fits in less well are also subject to more scrutiny from reviewers and are rejected more often.

Secondly, we investigate an application of highly predictable code: code completion. Previous work had evaluated the performance of language models in this application in isolation; we compare the language model approach to a commonly used code completion engine. We find that it compares favorably, achieving substantially higher accuracy scores. In particular, a combination of the two approaches yielded the best results.

Finally, we investigate instances of highly unpredictable code in order to automatically detect faults. We find that buggy lines of code are substantially less predictable, becoming more predictable after a bug is fixed. Our bug detection approach yields performance comparable to popular static bug finders, such as FindBugs and PMD. Our results further confirm that statistical (ir)regularity of source code from a natural language perspectives reflects real-world phenomena.
Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, Delft University of Technology
University supervisor: Dr. A. Bacchelli, Faculty EEMCS, Delft University of Technology
Company supervisor: Prof. P.T. Devanbu, University of California, Davis
Committee Member: Dr. C. Hauff, Faculty EEMCS, Delft University of Technology
Committee Member: Prof. J. Pouwelse, Faculty EEMCS, Delft University of Technology
This document represents my MSc thesis, written between July 2014 and August 2015, in the second half of my Master’s in Computer Science at Delft University of Technology. In this thesis, I discuss three applications of language modeling to software engineering research: code review, code completion and fault detection. Each of these applications are discussed in a chapter derived from a research paper, two of which were published and one of which is part of ongoing research. I wrote this thesis partially while at the University of California at Davis, under the supervision of Prof. Devanbu and partially at Delft University of Technology under the supervision of Dr. Bacchelli.

The creation of the chapters in this thesis has been an exciting and rewarding road. I was fortunate enough to collaborate with a variety of people and spend six months in the USA; the research papers on which the chapters in this thesis are based testify of the opportunities my graduation period has brought. I would like to thank Dr. Bacchelli and Prof. Devanbu for their supervision, as well as all the collaborators that I had the pleasure of working with in the past year. I also want to thank to my friends and family, for bearing with me and supporting me.

Vincent Hellendoorn
Delft, the Netherlands
August 8, 2015
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Language Modeling and Software Engineering</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Overview of this thesis</td>
<td>2</td>
</tr>
<tr>
<td>2 Background and Related Work</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Statistical Language Modeling</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Language Modeling in Software Engineering</td>
<td>5</td>
</tr>
<tr>
<td>3 Evaluating Code Contributions With Language Models</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Foreword</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Background</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Methodology</td>
<td>13</td>
</tr>
<tr>
<td>3.5 Findings</td>
<td>19</td>
</tr>
<tr>
<td>3.6 Discussion</td>
<td>22</td>
</tr>
<tr>
<td>3.7 Threats to Validity</td>
<td>26</td>
</tr>
<tr>
<td>3.8 Conclusion</td>
<td>28</td>
</tr>
<tr>
<td>4 Code Completion</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Foreword</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>4.3 Features</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Architecture</td>
<td>32</td>
</tr>
<tr>
<td>4.5 Results</td>
<td>34</td>
</tr>
<tr>
<td>4.6 Related Work</td>
<td>36</td>
</tr>
</tbody>
</table>
CONTENTS

4.7 Conclusions .................................................................................. 37

5 Fault Detection .................................................................................. 39
  5.1 Foreword ...................................................................................... 39
  5.2 Introduction ................................................................................. 39
  5.3 Background ................................................................................... 40
  5.4 Methodology ................................................................................. 44
  5.5 Evaluation ..................................................................................... 51
  5.6 Threats to Validity ......................................................................... 59
  5.7 Related Work ................................................................................ 61
  5.8 Conclusion .................................................................................... 62

6 Conclusion ........................................................................................ 65
  6.1 Contributions ............................................................................... 65
  6.2 Discussion .................................................................................... 66
  6.3 Future Work .................................................................................. 67

Bibliography .......................................................................................... 69
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Distribution of commits over pull requests (accepted and rejected) in the selected projects.</td>
</tr>
<tr>
<td>3.2</td>
<td>An overview of the setup used to acquire entropy scores for pull requests.</td>
</tr>
<tr>
<td>3.3</td>
<td>Change of entropy (in bits/token) of accepted and rejected pull requests following debate during code review.</td>
</tr>
<tr>
<td>3.4</td>
<td>Distribution of contribution entropy among contributors with and without experience.</td>
</tr>
<tr>
<td>3.5</td>
<td>Probability of a PR being rejected, by comments and entropy.</td>
</tr>
<tr>
<td>4.1</td>
<td>CACHECA produces suggestions intelligently, based on localness.</td>
</tr>
<tr>
<td>4.2</td>
<td>The architecture of CACHECA’s Mix model.</td>
</tr>
<tr>
<td>5.1</td>
<td>Phase I Data Collection: note Project Time Line, showing snapshots (vertical lines) and commits (triangles) at c1...c4. For every bugfix file commit (c3) we collect the buggy version and the fixed version, and use diff to identify buggy &amp; fixed lines.</td>
</tr>
<tr>
<td>5.2</td>
<td>Histogram of the number of lines deleted per file commit. The mean is 5, marked by the dashed line.</td>
</tr>
<tr>
<td>5.3</td>
<td>Determining parameters of cache model. The experiments were conducted on Elasticsearch and Netty projects for one-line bugfix changes. Y axis represents difference of entropy of a buggy line w.r.t. non-buggy lines in the same file.</td>
</tr>
<tr>
<td>5.4</td>
<td>Entropy difference between non-buggy, buggy, and fixed lines. File commits upto 5 deleted lines are considered, since five is the average number of deleted lines per file commit (see Figure 5.2).</td>
</tr>
<tr>
<td>5.5</td>
<td>Overall AUCEC upto inspecting 20% lines for all the projects.</td>
</tr>
<tr>
<td>5.6</td>
<td>Closer look at low order AUCEC, upto inspecting 5% lines for individual project</td>
</tr>
<tr>
<td>5.7</td>
<td>AUCEC 3 performance of $gram+wType vs. PMD and the combination model.</td>
</tr>
<tr>
<td>5.8</td>
<td>AUCECL performance of $gram+wType vs. PMD and the combination model.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Language Modeling and Software Engineering

Modern software languages are the result of a long evolution of computer programming methods. The 20th century saw the advent of computing machines, leading from the Turing Machine [24] and Von Neumann’s architectures [100] to IBM’s card sorting machines and modern PCs. Although computers were initially programmed manually or using Assembly language, the need for easier input methods for formulae soon led to the creation of programming languages such as GEORGE [54] and FORTRAN [10]. The subsequent decades saw the adoption and expansion of compilers and interpreters to create increasingly high-level programming languages.

While programming languages were becoming increasingly high-level, advances in the Natural Language Processing domain led to the adoption and improvement of natural language models (see [23]). These models aim to capture statistical regularities in natural language. A properly trained language model can be applied to a variety of tasks, such as optical character recognition, spelling correction [23], machine translation [14] and, more generally, judging the fluency of a text [92].

Hindle et al. first observed that the same language models that were used on natural language were also applicable to modeling software language [47]. Software, he found, behaves like natural language in a variety of regards. In fact, software language (or source code) exhibits a greater degree of repetition and predictability than natural language [37, 47]. These findings have led to advances in a variety of software engineering disciplines and inspired the three papers that underly this thesis.

The main topic of interest in this thesis is the relation between statistical (ir)regularity of source code and the real-world impact of this source code on software developers. Our central thesis is this: statistical predictability, or “naturalness”, of software influences its engineering and development. To investigate this, we will study the following three main research questions, representing three perspectives on statistical modeling of source code:
1. What is the relation between predictability of source code to the perception thereof by developers?
2. How can we apply highly predictable source code to code completion?
3. How can we apply highly unpredictable source code to fault detection?
1. INTRODUCTION

1.2 Overview of this thesis

I will first discuss the core principles of statistical language modeling (and the ngram language model in particular) in Chapter 2. Here, I will give an overview of the related work on modeling source code in software engineering research. I will then investigate each research question in turn in chapters 3, 4, and 5. Each chapter is based on a research paper, as described below, and is included in its original form with only minor modifications to allow fluent reading.

In Chapter 3, I investigate the response of software developers to (un)predictable source code. To this end, I investigated the statistical regularity of source code that is submitted to on-line collaborative software repositories. I studied the relation between the statistically computed (ir)regularity of the submitted code and the reaction to this code by the reviewers. I answer questions including: does less regular code (code that “stands out”) prompt more rigorous reviews and is more regular code (code that “fits in easily”) more likely to be accepted? This chapter is based on the following paper: Will They Like It? Evaluating Code Contributions with Language Models. I was first author on this paper; it was co-authored by Dr. Alberto Bacchelli and Prof. Premkumar Devanbu. I defended this paper at the 2015 Mining Software Repositories Conference (co-located with the International Conference on Software Engineering - ICSE).

In the subsequent two chapters, I will discuss two applications of statistical modeling of source code. Firstly – as was already observed by Hindle et al. [47] and by many since [98, 84, 65, 71] – highly predictable patterns in source code can be applied to a code completion task. By computing highly predictable (or regular) examples of source code, we can substantially improve over code recommender systems that don’t rely on practical usage statistics. In Chapter 4, we expand on statistical code completion results from previous work [47, 98]: we created an Eclipse plug-in of language model-based code completion, compared it with Eclipse’s default Content Assist [31] on a corpus of source code and provide practical heuristics to merge the completions from both approaches to further improve performance. This work is based on the 2015 ICSE DEMO-track paper: CACHECA: A Cache Language Model Based Code Suggestion Tool, on which Christine Franks was first author, I was lead author and the co-authors were Zhaopeng Tu and Prof. Premkumar Devanbu.

Thirdly, we will investigate the counterpart of highly predictable code in Chapter 5. Here, we investigated the hypothesis that less predictable, or regular, code is also more bug-prone. This work is based on the paper: On the “Naturalness” of Buggy Code, on which Prof. Baishakhi Ray and I were both first author. This paper was originally submitted to the 2015 Foundations of Software Engineering Conference, but was rejected; a revised version of this paper is scheduled for the 2016 ICSE conference. Co-authors on this paper were Zhaopeng Tu, Connie Nguyen, Saheel Godhane, Dr. Alberto Bacchelli and Prof. Premkumar Devanbu.

Finally, I will discuss the main contributions of this thesis and the opportunities for further research in Chapter 6. In particular, I analyze the ramifications of our findings (and those of related work) for our central thesis and discuss the implications for the evolution and research of software engineering.
Chapter 2

Background and Related Work

2.1 Statistical Language Modeling

This section was adapted from: Will They Like It? Evaluating Code Contributions with Language Models (now Chapter [3]), and is included there as well for consistency.

Computing Language Models

To judge the similarity of a sequence of tokens with respect to a corpus, a language model assigns it a probability by counting the relative frequency of the sequence of tokens in the corpus [92]. In the natural language setting, these models are used for tasks such as speech recognition, machine translation, and spelling correction [23, 15, 20].

We can write the probability of a sequential language fragment \( s \) of \( N \) tokens \( w_1 \ldots w_N \) as:

\[
p(s) = p(w_1) \cdot p(w_2|w_1) \cdot \ldots \cdot p(w_N|w_1 \ldots w_{N-1})
\]

\[
= \prod_{i=1}^{N} p(w_i|w_1 \ldots w_{i-1})
\]

Each \( p(w_i|w_1 \ldots w_{i-1}) \) can then be estimated as:

\[
p(w_i|w_1 \ldots w_{i-1}) = \frac{c(w_1 \ldots w_i)}{c(w_1 \ldots w_{i-1})}
\]  \hspace{1cm} (2.1)

where \( c \) means ‘count’.

However, as the context (or history) of a token lengthens, it becomes increasingly less likely that the sequence has been observed in the training data, which is detrimental to the performance of the model. \( N \)-gram models approach this problem by approximating the probability of each token based on a context of the last \( n \) tokens only:

\[
p(s) = \prod_{i=1}^{N} p(w_i|w_1 \ldots w_{i-1}) \approx \prod_{i=1}^{N} p(w_i|w_{i-n+1} \ldots w_{i-1})
\]
Intuitively, this approximation states that a token is only influenced by the preceding \( n \) tokens (formally assuming a Markovian property of language). Estimating the probabilities in an \( n \)-gram model is analogous to Equation (3.1), with the counts only considering the last \( n \) words. The choice of \( n \) is important: Short contexts are observed often but hold little information, whereas longer contexts hold more information but are rarely seen. Good smoothing methods make use of these qualities by combining models of different length \([23]\). Smoothing methods are used to guarantee that unseen token sequences are not assigned zero probability, which would be destructive in applications such as speech recognition.

Although research in the Natural Language Processing (NLP) community has shown that this approximation discards a significant amount of information \([89, 22, 32]\), \( n \)-gram models are widely used as they can be easily generated from the training data while providing powerful models \([74]\).

Models on Source Code

Hindle et al. were the first to show that these models capture regularity in source code, and showed that source code is even more predictable than typical natural language corpora \([47]\). They define a ‘sentence’ as a program, composed of allowable program tokens.

Measuring Performance

The geometric mean of the probabilities assigned to each token in a sentence is at the core of the most common metrics used in NLP. It is well suited to deal with the widely varying range of probabilities that are typically assigned by a language model. Given a sentence \( s \) of length \( N \) and the probabilities for each word in \( s \): \( p(w_i|h) \) (where \( h \) denotes a context of some length), the geometric mean is computed as follows\([1]\):

\[
\prod_{i=1}^{N} p(w_i|h) = 2 \sum_{i=1}^{N} \log_2(p(w_i|h))
\]

(2.2)

In NLP, the most commonly used metrics to measure the performance of a language models \( p \) are cross-entropy (\( H_p \), measured in bits/token) and perplexity (\( PP_p \)):

\[
H_p(s) = -\frac{1}{N} \sum_{i=1}^{N} \log_2(p(w_i|h))
\]

\[
PP_p(s) = 2^{H_p(s)}
\]

The perplexity of a string is the inverse of the geometric mean of the probabilities of its tokens. The cross-entropy (also entropy) is the binary log of the perplexity; it was found to have a strong, negative correlation with predictive quality in NLP applications (e.g., in speech recognition) \([23]\). Previous work in the application of NLP techniques to source code has consistently reported cross-entropy (lower results imply higher predictive quality) and we follow this example.
2.2 Language Modeling in Software Engineering

The application of language models to software engineering research has led to rapid advances in a number of domains, including code completion \([98, 67, 84, 65]\), language migration \([64, 50, 68]\) and modeling of source code itself \([55, 101, 98, 48, 5, 65]\). The latter has attracted most attention. Each chapter contains a Related Work section relevant to topic of that section in particular; here we briefly discuss general work on language modeling in software engineering research.

2.2.1 Modeling Source Code

It was observed by a number of authors that there is substantial room for improvement over the simple \(n\)-gram models (estimated on lexical tokens), both with regards to specific applications and in the general task of computing predictability of source code. Although only a few of the recent papers have focused purely on modeling of source code \(i.e.\) without evaluation on a specific application \([55, 91, 63]\), improved modeling of source code has played a central role in many investigations, including:

- **mining idiomatic code**, by using a novel tree-modeling and mining technique from NLP \([6]\);
- **auto-folding source code** using a scoped topic model \([36]\);
- **probabilistically inferring program properties**, using Conditional Random Fields \([83]\);
- adding lexical \([68]\) and grammatical \([50]\) information to \(n\)-gram models for language migration;
- **improving code completion** by emphasizing information in the local context \([98]\).

Additionally, a number of authors have studied properties of source code by investigating the performance of a variety of modeling techniques. Recent investigations have approached modeling of source code in a variety of ways, including:

- adding information to a model based on lexical tokens \([98, 67]\);
- modeling source code at the level of the Abstract Syntax Tree \([55, 6, 65]\);
- modeling source code in terms of Control-Flow Graphs \([65, 48, 83]\);
- modeling source code using Neural Networks \([101, 84, 63, 62]\).

These investigations have led to insights about important structural components of software and have (often) improved state-of-the-art performance in SE applications. Tu \textit{et al.} introduced a cache-based language model: a language model that emphasizes local context \(e.g.,\) variable names used in the current file) by means of a cache component \([98]\). Their model is motivated by a study into local repetitiveness of source code, in which they find that source code is \textit{locally repetitive}, both in the single file and in closely related files. By contrast, they show that the cache-based language model yields substantially less improvement on a natural language corpus. Maddison & Tarlow investigate the creation of generative language models for source code \([55]\). They find that conventional tree-based models such as
2. BACKGROUND AND RELATED WORK

Context-Free Grammars (CFGs) work poorly because they fail to take data-dependencies into account. They propose a model that makes use of deterministic and latent variables, and demonstrate that this model substantially improves entropy scores on test data.

Both Mou et al. [63, 62] and White et al. [101] investigated the application of (deep learning) neural networks to source code. Their findings demonstrate that neural networks can make powerful language models; White et al. discuss that deep learning networks can model long-distance dependencies, whereas Mou et al. observe that neural networks perform well by exploring program features automatically. Nguyen et al.’s instead add semantical information to language models by annotating tokens with information such as type, scope and dependencies [67, 68]. Other work in this category includes Nguyen et al., who develop a graph-based language models for API prediction [65], and Mou et al.’s application of Neural Network Language Models to program classification [63, 62].

2.2.2 Applications to Software Engineering

Beyond studying properties of software through language modeling, recent research has tackled a number of applications to software engineering have arisen. I will enumerate the most prominent bodies of research, and those which have most relevance to this thesis in this section.

Code Completion

The most prominent application to SE, as also investigated by Hindle et al. in their original work [47], is to a code completion (or suggestion) task. Both Tu et al.’s cache-based model [98] and Nguyen et al.’s addition of semantic information [67] were demonstrated to improve the state-of-the-art results in this area. The best performance on API call suggestion was found by Nguyen et al. by modeling source code as a graph instead [65]. This work is related to their earlier (not naturalness-based) work on code completion [66]. Their work is closely related to earlier work by Raychev et al., who target synthesizing sequences of method calls [84]. Raychev et al. similarly mine sequences of method invocations and apply these to the task of completing partial programs with “holes” in which method invocations must be suggested.

Language Migration

Nguyen et al. explored the possibility of migrating source code between programming languages using natural language machine translation techniques [67]. They apply a phrase-based machine translation model to the task of migrating Java code to C#. They find that these methods yield high lexical translation accuracy (over 80%), but the produced translations are often syntactically incorrect. Nguyen et al. call for “a more program-oriented SMT model” and Karaivanov et al. make a step in this direction [50]. By making use of the grammatical structure of the target programming language, they find that the parse rate of proposed translations greatly improves (over 98%) and the compile rate increases from 48.5% to 60.7%. Like Nguyen et al. [67], they suggest incorporating program semantics as a worthwhile pursuit for further improvements. Nguyen et al. used the insights from their
earlier work to automatically learn API mappings between Java and C# [64]. Their tool, based on IBM’s Model 2 for alignment in natural language translation [15], manages to improve over state-of-the-art tools and discovers novel API mappings.

Fault Detection

Campbell et al. used language models to improve the error reporting on the location of a syntax error (e.g., a typo) [19]. They find that using language models can help the compiler in localizing the cause of syntax errors in Java. In a technical report, the proposed tool was extended to Python (as an example of a dynamically typed language), and a number of adaptations were proposed to accommodate Python as being “a different point in the space of programming languages” [18]. In a Master’s thesis, Devin Chollak used an n-gram model to infer programming rules for defect detection. Incorporating control-flow information, the author is able to detect 310 violations in 14 open source Java projects, of which 43 were marked as bugs by the author and 2 were confirmed by the developers (the other 41 awaiting response at the time of publishing the thesis).

Other Directions

Other applications studied include studying uniqueness of code and changes to code bases. Predominantly, they find that large portions of source code in existing projects appear at least once elsewhere [38], and that changes made to existing projects exhibit a fair degree of repetition as well [11, 70, 81]. Other work has studied program comments and other program-related artifacts. Finally, a body of work preceding Hindle et al. has studied plagiarism detection using a technique closely related to n-gram models [76, 72].
Chapter 3

Evaluating Code Contributions With Language Models

3.1 Foreword

This chapter is based on the research paper: Will They Like It? Evaluating Code Contributions with Language Models. I was first author on this paper and defended the paper at the 2015 Mining Software Repositories Conference (co-located with the International Conference on Software Engineering); it was co-authored by Dr. Alberto Bacchelli and Prof. Premkumar Devanbu. I was in charge of writing and the experimentation, and (co-)wrote all sections.

3.2 Introduction

Code review is the manual assessment of source code by human reviewers. Most open source software (OSS) projects, which often heavily rely on contributions of disparate developers, consider code review a best practice both to foster a productive development community [35] and to ensure high code quality [13]. Nowadays code reviews are often mediated by tools (e.g., [40, 41, 75, 52]), which record information that can be mined to better understand the factors influencing the code review process and the acceptability of contributions.

Prior research (in both commercial and OSS settings) has investigated meta-properties stored in code review data, such as size, number of commits, and time to review, and properties of the files that were changed (e.g., [43, 58, 12]). Other research investigated review information to explore the influence of social aspects on software development in OSS communities [12, 26], in particular on the actions of a reviewer when faced with a new contribution [50]. Rigby et al. used code review recorded data to triangulate an investigation on the influence of personal aspects of the reviewer, such as experience and efficiency on the code review process [86]. This research has led to valuable insights on how the reviewer’s attitude towards the (potentially unknown) contributor affects the process and outcome of code review.
3. Evaluating Code Contributions With Language Models

Little is known, yet, of what properties of the *submitted code* influence the review of a changeset and its acceptability. The earliest and most significant insights in this area are those by Gousios et al., who conducted a qualitative study with reviewers of proposed code changes in GitHub [44]. Gousios et al. reported that integrators consider style conformance the top factor when evaluating quality of submitted code. Code quality and code style were reported as the top two factors influencing the decision to accept a contribution.

Our goal is to extend these qualitative insights by quantitatively evaluating the influence of stylistic properties of submitted code on both the process and the outcome of code review. To achieve this, we make use of *language models* constructed on source code [47], which were proven to be well-suited to capture stylistic properties of code in the context of a project [6]. Hence, we use this tool to measure how well submitted code *fits in* with the project’s code as a whole and analyze how this influences code review.

We conduct our evaluation on projects that use GitHub, the popular social coding platform at the basis of the study by Gousios et al. GitHub offers built-in capability for code review by implementing the pull-based development model [43]. In this model, contributors do not have access to the main repository, rather they fork it, make their changes independently, and create a pull request with their proposed changes to be merged in the main repository. The project’s core team is then responsible for reviewing and (if found acceptable) eventually merging the changes on the main development line. We consider 22 popular and independent projects on GitHub and analyze a total of 1.4M lines of submitted code, across 6,000 pull requests.

The results of our evaluation show that accepted changesets are significantly more similar to the project at the time of submission than rejected pull requests, supporting that conformance to the code style is a factor that influences code review. We further show that contributions that were subject to more extensive reviews, such as debate regarding a changeset, were substantially less similar to the project’s code style. Further investigation supports our finding that highly dissimilar contributions are isolated by project maintainers and receive substantially different treatment during code review. Finally, we show that contributions by novel contributors show a substantial increase in similarity to the project’s code as the contributor gains experience.

**Structure of the paper.** In Section 3.3 we describe the technical background, particularly in the field of natural language processing (NLP) related to language models applied to source code. In Section 5.4 we introduce our research questions and detail the research method we follow. We present our findings in Section 3.5 and discuss them in Section 3.6. In Section 5.6 we identify a number of threats to the validity of our study. We conclude in Section 5.8.

3.3 Background

This work builds on literature on code reviews, both in industrial settings and, more recently, in OSS projects (e.g., [9] [13] [86]), and on recent discoveries in the application of language models to source code [47] [99] [5].
3.3. Background

3.3.1 Code Reviews

Modern Code Review, as described by Bacchelli & Bird, has gained popularity in several large software development companies as well as in OSS projects and is of increasing interest in recent research [9]. In OSS projects in particular, code review is primarily done by a core group of project maintainers who receive independent contributions from a wide variety of contributors [59]. Bacchelli & Bird find that understanding of the code and the reason for a change is the most important factor in the quality of code reviews [9].

Dabbish et al. also study code review in OSS settings, and find that reviewers make a rich set of inferences about contributors based on their activity [26]. Marlow et al. find that some of these inferences influence both the nature of the review, and the likelihood of the code being accepted [56]. These works study familiarity and social issues in code review, and are complementary to ours. Tsay et al. further expand on this line of work by studying Pull Requests on Github in particular and studying both social and technical aspects of code review from the perspective of project maintainers [97].

Successive papers by Rigby et al. study the use of code review in OSS settings, as well as the influence of a number of meta-properties on the efficacy of a code review [85, 87, 86]. They statistically model the relationship of meta-properties on code review outcomes (e.g., interval, efficiency). *Inter alia*, they study how reviewer’s expertise in the submitted code influences the promptness of the review. Our focus on the properties of the submitted code per se complements theirs.

In this paper we focus on code-reviews using the pull-based software development model, which is offered by widely used platforms, such as Github and BitBucket. The pull-based software development model is gaining huge popularity [43], with more than half a million projects adopting it on GitHub alone. Our work uses language models to evaluate the stylistic content of the code in pull requests and to compare it with the code already existing in the project.

3.3.2 Language Models

Hindle et al. show that source code has regularities that can be captured with statistical models developed for natural language [47]. They find that the local regularity arises primarily from repetitiveness within a project rather than patterns belonging to the programming language. This intra-project regularity suggests that language models could quantify the extent to which new code fits in an existing code base. They evaluate the practical potential of the high regularity that language models found in source code by building a code completion tool and showing that it can significantly improve the default suggestions given by the Eclipse IDE[1].

Subsequent work investigates the potential of these models in a number of settings, such as their applicability to mining repositories at massive scale [5], and the influence factors such as semantic information [71] and local context [99] on code completion performance. In particular, Allamanis et al. investigate whether coding conventions can be extracted automatically from OSS projects using language models [6]. In this sense their work is closely

1https://www.eclipse.org/
related to ours, as our results suggest the presence of implicit conventions and coding standards in OSS projects. In similar work, Allamanis et al. investigate the possibility of mining code ‘idioms’: generalized, non-trivial patterns of code that occur frequently across a project [6]. They show a number of examples of such idioms that can be mined using language models, which reveal promise for future work that investigates how new contributors can be supported in writing acceptable code.

Computing Language Models

To judge the similarity of a sequence of tokens with respect to a corpus, a language model assigns it a probability by counting the relative frequency of the sequence of tokens in the corpus [92]. In the natural language setting, these models are used for tasks such as speech recognition, machine translation, and spelling correction [23, 15, 20].

We can write the probability of a sequential language fragment $s$ of $N$ tokens $w_1 \ldots w_N$ as:

$$p(s) = p(w_1) \cdot p(w_2|w_1) \cdots p(w_N|w_1 \cdots w_{N-1})$$

$$= \prod_{i=1}^{N} p(w_i|w_1 \cdots w_{i-1})$$

Each $p(w_i|w_1 \cdots w_{i-1})$ can then be estimated as:

$$p(w_i|w_1 \cdots w_{i-1}) = \frac{c(w_1 \cdots w_i)}{c(w_1 \cdots w_{i-1})}$$

where $c$ means ‘count’.

However, as the context (or history) of a token lengthens, it becomes increasingly less likely that the sequence has been observed in the training data, which is detrimental to the performance of the model. $N$-gram models approach this problem by approximating the probability of each token based on a context of the last $n$ tokens only:

$$p(s) = \prod_{i=1}^{N} p(w_i|w_1 \cdots w_{i-1}) \approx \prod_{i=1}^{N} p(w_i|w_{i-n+1} \cdots w_{i-1})$$

Intuitively, this approximation states that a token is only influenced by the preceding $n$ tokens (formally assuming a Markovian property of language). Estimating the probabilities in an $n$-gram model is analogous to Equation (3.1), with the counts only considering the last $n$ words. The choice of $n$ is important: Short contexts are observed often but hold little information, whereas longer contexts hold more information but are rarely seen. Good smoothing methods make use of these qualities by combining models of different length [23]. Smoothing methods are used to guarantee that unseen token sequences are not assigned zero probability, which would be destructive in applications such as speech recognition.

Although research in the Natural Language Processing (NLP) community has shown that this approximation discards a significant amount of information [89, 22, 32], $n$-gram models are widely used as they can be easily generated from the training data while providing powerful models [74].
3.4 Methodology

Models on Source Code

Hindle et al. were the first to show that these models capture regularity in source code, and showed that source code is even more predictable than typical natural language corpora [47]. They define a ‘sentence’ as a program, composed of allowable program tokens.

Measuring Performance

The geometric mean of the probabilities assigned to each token in a sentence is at the core of the most common metrics used in NLP. It is well suited to deal with the widely varying range of probabilities that are typically assigned by a language model. Given a sentence \( s \) of length \( N \) and the probabilities for each word in \( s \): \( p(w_i|h) \) (where \( h \) denotes a context of some length), the geometric mean is computed as follows:

\[
\sqrt[N]{\prod_{i=1}^{N} p(w_i|h)} = 2^{\frac{1}{N} \sum_{i=1}^{N} \log_2(p(w_i|h))}
\]

In NLP, the most commonly used metrics to measure the performance of a language models \( p \) are cross-entropy (\( H_p \), measured in bits/token) and perplexity (\( PP_p \)):

\[
H_p(s) = -\frac{1}{N} \sum_{i=1}^{N} \log_2(p(w_i|h))
\]
\[
PP_p(s) = 2^{H_p(s)}
\]

The perplexity of a string is the inverse of the geometric mean of the probabilities of its tokens. The cross-entropy (also entropy) is the binary log of the perplexity; it was found to have a strong, negative correlation with predictive quality in NLP applications (e.g., in speech recognition) [23]. Previous work in the application of NLP techniques to source code has consistently reported cross-entropy (lower results imply higher predictive quality) and we follow this example.

3.4 Methodology

The main goal of this research is to quantitatively evaluate the influence of stylistic similarity of submitted code (to existing code) on both the code review process and outcome. In prior work, we have shown that language models capture similarity within projects and differences across projects [17]; we have also found coarser differences (across/within application domains) and finer ones (between files within the same project) [33]. In this paper, we study the value of language models to gauge the relation between the acceptability of submitted code and its naturalness: its similarity to a project’s code. Therefore, we structure our goal around the following research questions, which we iteratively refined while analyzing our results:

---

\( 2 \)The right-hand (equivalent) form in Equation (3.2) avoids rounding problems that typically arise with a product over many probabilities.
3. Evaluating Code Contributions With Language Models

RQ1: Are rejected PRs less natural than accepted ones?
RQ2: Are more debated PRs less natural?
RQ3: Does reviewing affect the entropy of debated PRs?
RQ4: Does contributions’ naturalness grow with experience?

In the first place, we seek to answer whether the outcome of code review is correlated with statistical properties of the submitted source code. Then, we study correlations with the process of the code review by looking at submissions that were subject to debate before being decided. In particular, we contrast these with submissions that were accepted with little feedback. Finally, we use the found correlations to investigate other properties that influence code reviews, such as author experience and goal of the contributions.

3.4.1 Experimental Setup

We focus our research on pull requests (PRs) submitted to Github projects. This has several advantages: Github data is readily accessible, both via an API and through the git command-line tool. In particular, this allows us to revert a copy of a project to its exact state at the time of any code review, which is necessary for our model to work. Future work may study the existence of similar patterns in industrial code bases and other OSS projects.

Our approach works in three steps. (i) For each project we extract the lines added by each pull request, as well as the lines added by each commit contained within that pull request. These lines constitute the test set. (ii) For each PR, we extract the training set from a copy of the project that is reverted to the project’s state when the PR was submitted. (iii) We train a language model on all .java files in the training set and test it on the lines of java code in the test set. The output of the language model is an entropy score, reflecting the similarity of the pull request to the code base at the time of submission.

Additionally, we used the Github API to extract a number of properties for each pull request, such as author, commit message and number of (review) comments. With this information we analyze factors that influence the results and discover less obvious correlations with entropy. In the following, we detail the approach used in selecting and collecting the data, the procedure used to extract our training corpus and test set, and finally the criteria by which we processed the data.

3.4.2 Project selection

We use two criteria for selecting the projects: (1) The project must primarily contain java code, and (2) the project must make active use of pull requests. The first requirement is a matter of choice: the language models described in section 3.3.2 are fairly agnostic w.r.t. the language and we discuss extensions to other languages in Section 3.6. The second requirement relates to the findings by Gousios et al. [43], who reported that only a small fraction of projects on Github make active use of pull requests. Among these projects, most have used no more than 20 pull requests in total (25 among projects with more than one
3.4. Methodology

Figure 3.1: Distribution of commits over pull requests (accepted and rejected) in the selected projects.

They also found that a few projects account for a very large number of pull requests.

In light of the diverse usage of the pull-based software development model across Github, we define ‘making active use of pull request’ according to three criteria. Primarily, we looked for projects that had 26 or more closed pull requests. This guarantees that we consider only projects belonging to the 5% most active users of the pull-based model. Secondly, we omitted projects in which the majority of contributions were made outside the pull-based model. In most of the resulting projects, 80% or more of all commits were part of a pull request. Finally, we omitted projects in which no rejected PRs were found that qualified as debated, as described in section 3.4.5.

The projects were selected based on these three criteria from among the Java projects that are currently popular, as reported by Github’s ‘Trending Repositories’ feature. Following these criteria, we totaled 22 projects, containing approximately 10,000 pull requests and over 20,000 commits. We then filtered out the PRs that did not include Java code and PRs that contained an inordinate number of additions (a fair number of PRs submitted several thousands of lines, typically either by mistake or by merging two branches), which left a little over 7,500 PRs and 13,500 commits. Of these, 1,634 PRs received no comments of any kind before being closed, typically by the original author. As we are concerned with code that is reviewed, we omit these from our final dataset in all results except in the evaluation of RQ1, to avoid a potential confound. Figure 3.1 shows the distribution of the number of commits in accepted and rejected PRs. The majority of PRs, accepted or rejected, consist of a single commit, but the distribution is skewed in that a small number of PRs contain a large number of commits. The largest PRs contain over 40% of the total commits.

The selected PRs contain a total of 1.4 MLOC submitted to 22 popular Github projects.
over a period of 2-40 months. Table 3.1 reports the selected projects with the number of PRs linked to the projects in ascending order. The project names are reported with the corresponding user account, as many forks exist. The considered projects were aged between two months and eight years and contained between 56 and 1009 PRs (after filtering) from between six (Physical-web) and 234 (Jenkins) distinct contributors (median: 49). This corpus represents a diverse array of size, age and diversity of contributors and provides a large enough base to report statistically significant results on.

3.4.3 Data extraction

For each project, we used the Github API to create a list of indices, corresponding to all PRs that were closed at that time. These indices were collected between Sep 9, 2014 and Nov 3, 2014. Additionally, we used the Github API on each PR to extract properties such as author, commit message, (review)comments, and size.

---

Table 3.1: Considered pull requests and commits by project. These numbers include PRs that were isolated because they received no comments.
3.4. Methodology

Figure 3.2: An overview of the setup used to acquire entropy scores for pull requests.

In the following we describe the process we used to generate entropy scores for each pull request (illustrated in Section 3.4.2). Step 1: we use the information on the lines that were added and deleted in each PR (provided by GitHub in text format through a URL) to create a test set for each PR, which consists of the lines added in the submission. As can also be seen in Section 3.4.2, modified lines in this diff are represented by a removal of the old line and addition of the new line. Hence, by constructing our test set of all added lines, we include both newly added code and lines that were modified by the pull request. Furthermore, we derive the test set for each commit part of the PR. Here, the test set contains the added and modified lines that the changeset contained up to that point in time. Step 2: we extract the training set (used to estimate the language model) from a local clone of the project. Using the git reset command, we revert the local copy of the project to its state when the PR is submitted. Step 3: we use the extracted testing set to build a language model out of the Java code in the resulting project, using the techniques outlined in Section 3.3.2. We use the test set as input to the language model. The output is an entropy score reflecting how similar the submitted code was to the project at submission time.

3.4.4 Analysis

To answer our research questions, we need to know whether the pull request is eventually accepted. Gousios et al. find that a significant fraction of PRs in GitHub are (partially) merged in ways that are different from the official merge tool [43]. As a consequence, the Github API shows a significant fraction of pull requests as PRs as rejected, even in projects that rely heavily on pull requests. To better identify whether a pull request was merged, Gousios et al. develop four heuristics to find if the submitted code was merged into the code base via different means. In this study, we use the same heuristics (we found an acceptance ratio of 79% across the 26 projects, comparable to the results of Gousios et al.).

To study the effect of entropy on the process of a code review, we further divide the PRs into the categories debated and undebated. We define a PR as debated if it received at least

---

6e.g., https://github.com/ReactiveX/RxJava/pull/1094.diff
three comments during review (review + discussion comments), following Gousios et al. who find that a PR on average receives 2.77 comments (review + discussion comments) and who suggest that a small fraction of pull requests receives the majority of comments [43]. For the sake of comparison, we furthermore define a PR as undebated if it received at least one comment but was not debated. We find that approximately 21% of all pull requests received no comments before being decided, 38% received feedback but not more than two comments, and 41% was debated.

We furthermore intend to investigate the influence of authorship on the entropy of contributions. In order to derive statistically significant results related to authorship, we restrict our focus to the ten largest projects, as these have sufficiently many distinct authors and contributions per author. Furthermore, in each project’s history there was a small number of contributors (typically three or less) who were responsible for the vast majority of contributions. These authors may therefore have contributed so much that statistical similarities which we find between the project and their PRs may be due to them having shaped the project in the past. Hence, we restrict ourselves to PRs from authors who have submitted less than 20 PRs in the entire project history, which guarantees that all remaining authors have contributed no more than 5% of the project’s code. Setting this cut-off to 10 PRs produced similar results but substantially reduced the number of data points.

Observing that the median number of contributions per author among the remaining PRs was 3, we set the threshold for experienced at having contributed at least three times before. Here too, similar thresholds yielded comparable results.

3.4.5 Evaluation

We employ language models to evaluate aspects of submitted code that impact the code review process. To this end, we typically divide the studied pull requests into two categories (e.g., accepted and rejected) and compute the mean entropy for both categories in every project. We then pair the results per project and compare the outcome between projects using both a paired t-test and a non-parametric, Wilcoxon signed rank test [96]. In general, we found that the tests yielded very comparable significance levels, suggesting that the data is predominantly normally distributed. Where one test yielded a lower level of significance, we make mention of this and report the lesser value. Furthermore, to quantify the effect of the difference in means between two categories, we compute Cohen’s D [96] on the difference scores of the paired data.

The ecological fallacy states that findings at an aggregated level (here: projects) may not apply to a disaggregated level (here: individual PRs); this was recently found to apply to empirical software engineering research as well [77]. Where applicable, we therefore validate our findings at the level of individual PRs by reporting results of a Wilcoxon rank sum test across the complete corpus of PRs (disregarding the projects). We generally aggregate to the project level because typical entropy scores of PRs differ substantially between projects (up to 1.5 bits).

In general, we found that results that held overall also consistently held on the largest projects, but not necessarily on the smaller projects (see Section 5.6 for a discussion on
this). We furthermore found that the effect size of significant effects differed substantially between these two categories (in favor of the larger projects). To avoid a potential size confound, we divide the selected projects into two categories: large and small, where the large projects contain approximately 1/3rd of the projects and 2/3rd of the pull requests. The category is shown in the last column of Table 3.1. Aside from reporting results for the average across all projects, when appropriate we report results for the average within these categories.

3.5 Findings

In this section, we report the results of our analysis.

RQ1: Are rejected PRs less natural than accepted ones?

For each of the 22 projects studied, we averaged the entropies over all accepted and rejected pull requests per project. The mean entropies of accepted and rejected PRs are 4.18 and 4.35 bits/token respectively (note that this measure, is log scaled; corresponding perplexities are 18.12 and 20.4, respectively). A paired t-test confirms that the average entropy of the rejected PRs is significantly higher than that of accepted PRs ($p < 0.01$) and computing the effect size (Cohen’s D on the paired data) reveals a moderate effect (0.61). This result is consistent across the corpus of individual PRs (disregarding the projects), where the mean entropies of accepted and rejected PRs were 4.14 and 4.44 respectively ($p < 10^{-5}$).

Next, we divide the projects into the groups large and small as explained in Section 3.4.5. The resulting statistical difference between accepted and rejected PRs can be seen on the first row of Table 3.2. A paired t-test confirms that rejected PRs are significantly less natural in the group of larger projects ($p < 0.05$) but cannot confirm this difference among smaller projects ($p > 0.1$). Furthermore, the effect size between accepted and rejected pull requests is substantially larger when considering just the bigger projects. A similar result was found on the corpus of individual PRs (large projects: $p < 10^{-5}$, small projects: $p < 10^{-3}$). These results suggest that, particularly among more established projects, the relative similarity of submitted code to a project as a whole has a measurable influence on the outcome of a code review.

Additionally, we investigated the influence of the PRs that were removed because they did not receive any comments before being decided (1,634 PRs, 93% accepted) and found that the inclusion of these PRs did not harm the previous results (in fact, the significance level on small projects came within the $p < 0.1$ range). Additionally, we noted that these unreviewed PRs have significantly lower entropies than reviewed ones ($p < 0.05$, moderate effect). Given these results, we may expect to see a difference in entropy with more extensively discussed PRs as well.

RQ2: Are more debated PRs less natural?

We separate debated (3,201 PRs, 80% accepted) and undebated PRs (2,958 PRs, 85.7% accepted), as defined in section section 3.4.5. If less natural code is more likely to be
3. Evaluating Code Contributions With Language Models

Table 3.2: Significance and effect size (Cohen's $D$) of entropy difference between different categories of Pull Requests. A is accepted, R is rejected; U is undebated, D is debated. $D < 0.5$ is considered “small” effect; $D < 0.8$ “medium”; otherwise “large”. Black $p$-values reflect significance at least at the $p < 0.05$ level, orange values: only evidence of significance found ($0.05 < p < 0.1$) and red values: no significant correlation found.

<table>
<thead>
<tr>
<th>id</th>
<th>PR type</th>
<th>Test</th>
<th>All projects</th>
<th>Large Projects</th>
<th>Small Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overall</td>
<td>A $&lt;$ R</td>
<td>$0.61 &lt; 0.01$</td>
<td>$1.05 &lt; 0.05$</td>
<td>$0.42 &gt; 0.1$</td>
</tr>
<tr>
<td>2</td>
<td>Undebated</td>
<td>A $&lt; R$</td>
<td>$0.64 &lt; 0.01$</td>
<td>$1.04 &lt; 0.05$</td>
<td>$0.57 &lt; 0.1$</td>
</tr>
<tr>
<td>3</td>
<td>Debated</td>
<td>A $&lt; R$</td>
<td>$0.11 &gt; 0.1$</td>
<td>$0.83 &lt; 0.1$</td>
<td>$-0.22 &gt; 0.1$</td>
</tr>
<tr>
<td>4</td>
<td>Rejected</td>
<td>U $&lt; D$</td>
<td>$0.06 &gt; 0.1$</td>
<td>$1.11 &lt; 0.05$</td>
<td>$-0.1 &gt; 0.1$</td>
</tr>
<tr>
<td>5</td>
<td>Accepted</td>
<td>U $&lt; D$</td>
<td>$0.65 &lt; 0.01$</td>
<td>$1.18 &lt; 0.05$</td>
<td>$0.63 &lt; 0.05$</td>
</tr>
</tbody>
</table>

Rejected, we may also expect that less natural contributions generate more discussion during review, so that debated PRs would have significantly higher entropies (i.e. be significantly less natural) than undebated PRs.

We particularly expect (eventually) accepted PRs that triggered debate to be less natural than PRs that were accepted quickly. We do not necessarily expect a comparable result for rejected PRs, however: code that is rejected with little feedback might be the least natural contributions of all.

We first investigate the previous result within the groups of debated and undebated PRs; rows 2 and 3 of Table 3.2 report the results. We first repeat the previous investigation on undebated PRs and find a slightly stronger version of the results for RQ1, particularly among smaller projects (paired $t$-test: $p < 0.05$, signed rank test: $p < 0.1$). We conduct the same analysis among debated PRs and find that, although no significant difference was found overall, there is evidence of the previous result among debated PRs in large projects. Results on the corpus of individual PRs are comparable: For undebated PRs the results from Section 3.5 hold; for debated PRs, a significant difference was found both overall and on the large projects ($p < 10^{-4}$) but none on the small projects ($p > 0.1$).

Comparisons 4 and 5 in Table 3.2 compare the results between debated and undebated PRs, first for (eventually) rejected PRs and then for accepted PRs. Here we find the largest difference between more established and smaller projects. Both rejected and accepted PRs were significantly more entropic when subject to debate in large projects, but on small projects only the latter result held and only with moderate effect.

The above results show several distinctions between undebated and debated pull requests: a) among the former, low entropy PRs were substantially more likely to be accepted, whereas among the latter we only found some evidence of this phenomenon on large projects; b) debated pull requests as a whole had substantially higher entropies than undebated ones, particularly those that were eventually accepted. The correlation between debate and entropy appears to be at least as large as that between acceptability and entropy. This raises a question: does entropy still play a role among controversial PRs? Or do our results primarily reflect that PRs with low entropy are likely to be accepted and receive little
3.5. Findings

Do reviewing affect the entropy of debated PRs?

We start by analyzing the entropy of the commits that compose PRs. In particular, we investigate PRs that were revised under debate. To this end, we collected 11,600 commits that were part of PRs that were both debated and revised with subsequent commits (these PRs had an average of approximately 10 commits per PR). For each commit, we calculated the entropy of the PR after that commit with respect to the core project; then, we computed the average change in entropy for accepted and rejected PRs between the first and last commit. The results are shown in Figure 3.3.

A t-test confirms that both accepted and rejected PRs increase slightly in entropy during revision \((p < 0.05)\). By manual inspection, we observed that contributors of debated PRs were asked to add novel code to their PRs (e.g., test cases). As new code is in general less predictable, this may have worked as a confound. No significant difference in entropy increase was found between accepted and rejected PRs.

To rule out confounds related to novel code, we then investigated changes at file level between successive commits. For each review comment \((i.e, a comment on a part of the

![Figure 3.3: Change of entropy (in bits/token) of accepted and rejected pull requests following debate during code review.](image-url)
submitted code), we compared the entropy of the file under discussion before and after revision. We found no significant evidence that the discussed files decrease in entropy during code review (neither among accepted nor rejected PRs), even when restricting ourselves to revisions in which the number of changed lines remained the same. We intend to refine this criterion in future work.

In the following question we study whether new contributors learn the project’s style conventions as their experience grows.

**RQ4: Does contributions’ naturalness grow with experience?**

Given the experimental setup, it can be argued that language models effectively measure the similarity in implicit coding style between a pull request and the existing code base. Hence, we expect the coding style of the author to have a significant impact. In particular, we expect novel contributors to write code that is more “fluently similar” (with respect to the project) than experienced contributors.

We first evaluate whether a difference in naturalness exists between PRs from more and from less experienced authors, where we use the definition of experience as in Section 3.4.4. We only include authors who contributed four or more times in the project’s history and therefore have contributions both before and after they qualified as experienced. We computed the difference in entropy of their contributions before and after they pass the ‘experience’ threshold. Applying the above restrictions left us with 1,432 PRs from 127 distinct authors.

The resulting distributions of entropies are shown in Figure 3.4. A t-test confirms that the entropy of contributions is significantly lower among PRs from contributors with prior experience than among PRs from contributors with little to no prior experience (p < 0.01). We furthermore found a large effect (D = 1.11). Further investigation revealed that the largest difference exists between contributors with one or two prior contributions and contributors with three or more prior contributions. Contributors with no prior experience showed a large degree of variation in entropy scores, which may be caused by this group often submitting relatively simple changesets upon their first contact with a project.

### 3.6 Discussion

A variety of previous research has studied the question why certain contributions to OSS projects are rejected or not accepted until after revision. Among others, this has produced anecdotal evidence that conformance to the project’s code style is considered a strong indicator of quality. The aim of this research was to quantitatively evaluate this aspect of code review using language models.

In the first research question we asked whether rejected pull requests have a higher entropy with respect to project’s source code than accepted ones. We found that the statistical similarity between submitted code and the core project is both measurable by language models and reveals statistically significant differences between accepted and rejected pull requests. Code in accepted pull requests has significantly lower entropy to project code than code in rejected ones, across all projects. This result corroborates the qualitative findings
3.6. Discussion

Figure 3.4: Distribution of contribution entropy among contributors with and without experience.

by Gousios et al. [44] on the importance of style conformance when reviewers evaluate contributions.

We also found that the effect size is larger and more significant in larger projects than smaller ones. This finding is in line with earlier work by Tsay et al., who found that well-established projects are more conservative in accepting pull requests [97]. Nevertheless, projects such as the Mozilla project[7] and the Linux Kernel[8] are both several times older and larger than most projects studied in this work, making them very interesting candidates to determine if and how our results generalize to other cases.

In the second research question we investigated whether contributions that are more debated are less similar to the project source code. We found that contributions that are directly accepted without discussion are significantly less entropic than debated ones, regardless of their eventual acceptability. Furthermore, the most entropic PRs were those that were both debated and eventually rejected, suggesting further that highly unconventional code is more likely to attract the attention (and face the disapproval of) project maintainers.

[1] https://www.mozilla.org/
Figure 3.5: Probability of a PR being rejected, by comments and entropy
3.6. Discussion

Figure 3.5 shows the probability of a pull request being rejected considering the number of comments it receives and its entropy. As can be seen, an increase in entropy almost consistently increases the likelihood of rejection, whereas when considering comments alone, we primarily see a distinction between PRs that received no more than two comments and those that received three or more comments. A larger number of comments is only correlated with a greater probability of rejection among PRs with a very high entropy. Furthermore, whereas comments can be seen as an artifact of the code review itself, the entropy of a changeset can be computed on submission. A ripe opportunity for future research would be to study whether entropy can be used to direct reviewers’ attention to code that is more likely to generate debate or to be rejected.

The third research question dives deeper into the context of debated pull requests and investigates whether the debated contributions decrease their entropy due to the review. We found no significant evidence of such a decrease. The results suggest that, while reviewers are quick to recognize and approve low entropy PRs, the naturalness of submitted code plays less of a role during more extensive code review. By manual inspection, we found that extensive code review was often concentrated around (changes of) functionality of the submitted code.

The aforementioned results do not preclude that code under review is asked to better conform to the project’s coding conventions. Code review comes in many shapes and we found many instances where novel contributors were asked to better conform to coding standards (often besides requests for functionality improvement) during code review. In our fourth research question we investigated a possible effect of these comments: whether novel contributors tend to adhere more to project’s style as their experience with the project grows. Our results indeed confirmed that contributors write code that is more similar over time. Nevertheless, we did not analyze further whether the project conformance comes naturally from a better knowledge of the project rather than comments received in code reviews. Studies can be designed and carried out to investigate this phenomenon.

Although we considered projects coming from a variety of communities we limited ourselves to OSS systems and the pull-based development model. We did not verify our results with industrial projects or on systems that make use of different code review models: these settings deserve additional study and may provide new insights on style conformance.

Throughout RQ2, 3 and 4, we have investigated a number of factors that influence both acceptability and entropy of PRs. Previous work provided evidence that many other factors influence the eventual acceptability of a contribution (e.g., whether the contribution adds novel code or modifies existing code, personal relationship with project maintainers). Although a full investigation of the relationship with other factors is beyond the scope of this work, it is of particular importance to investigate the type of PRs that are likely to be rejected. For instance, it might be that the observed correlation is caused by one type of PRs that has both high entropy and a high rate of rejection. If this is the case, the language models may be mirroring an underlying effect: the type of the PR. As an initial step in this direction, we investigated the influence of the type (or intent) of PRs on entropy and acceptability: We classified all PRs into five categories (‘Bug fix’, ‘Performance’, ‘Styling’, ‘Quality assurance’, and ‘Other’), by matching a set of manually derived keywords against the submission message. We found that the gap between accepted and rejected PRs was sig-
significant in all categories except among bug-fixes (only evidence of significance), although the mean entropies varied substantially between categories. The rate of acceptance was similarly consistent across the categories. PRs related to performance improvement (e.g., memory usage, speed) had substantially higher entropies than other PRs, with bug-fixes being the second highest. This meets our expectations, as these PRs (particularly the first category) generally contribute novel code to a project. Styling PRs (e.g., refactorings, enforcing coding conventions), on the other hand, showed the lowest entropy scores, as well as the largest gap between accepted and rejected PRs. This matches results from previous work, which found that low-entropy refactorings likely constitute acceptable contributions to OSS projects [6]. These results suggest that (a) the observed difference in entropy between accepted and rejected PRs is robust with respect to different types of PRs, and (b) language models may be most powerful when evaluating contributions that modify existing code. These results, and the combination of entropy with other factors that influence code review, warrant more extensive evaluation in future work.

Our study also effectively used language models as a tool to automatically compare new code to an existing code base. This opens a number of possible future applications: language models may be able to derive coding conventions corresponding to code that a new contributor is writing (e.g., code idioms [6]), to save effort on both the developer’s and the reviewer’s end. Moreover, as previously mentioned, language models can be used to direct reviewers’ attention to code that is more likely to generate debate. Finally, we hope to inspire new research by showing the effects that authorship and intent have on entropy scores, and the practical ramifications of these effects.

3.7 Threats to Validity

**Internal Validity** Assumptions we made in the selection of the projects may harm the internal validity of this study. The manual selection of the projects may have led to an unrepresentative sample of Github projects. To avoid this confound, we included every project listed on Github’s “Trending Repositories” page and satisfied the criteria. The projects selected were “trending” in the last month and collected in three turns between September and November 2014.

The three criteria we used to filter the potential projects are another potential threat to validity. The first criterion concerns the use of 26 or more pull requests. This number is selected based on previous work by Kalliamvakou et al., which found that 95% of the projects that have used pull requests for actual collaboration used no more than 25 pull requests in the project history [49]. In our eventual test set, (after filtering out unreviewed changes) the smallest project had 37 PRs and all others had more than 50.

The second criterion concerns the use of PRs as the primary source of contributions. On Github, the alternative to PRs is to commit directly to the main repository, something that is permitted to the maintainers of the project. Hence, we chose to identify projects as ‘actively using the pull-based software development model’ by taking the ratio of the count of the number of commits that were part of a PR to the total number of commits. Here, we took special care to count only commits made after the project adopted the pull request model, as some projects started before this feature was added. We excluded projects for which this
3.7. Threats to Validity

ratio is less than the 0.5. Although most projects in our study scored 0.8 or higher, some project owners used direct commits more frequently than PRs. This may affect the results on author experience, although we found no substantial difference in these projects’ results when compared with the remaining test set.

In a study using language models, it is critical that the training set and test set do not overlap. In the context of this study, that means that the code base must be fully reverted to a point in time before a PR has been accepted or rejected, otherwise the model would be biased towards accepted PRs. Using the combination of the ‘base SHA’ from the Github API and the `git reset --hard` command should guarantee this, and we manually verified the correctness of this combination for a small number of cases. We furthermore ran a series of tests under more strict conditions. We reverted the repository to its state before the first pull request and then for each PR from oldest to newest, updated the repository only to the moment the project was forked. By effectively testing on the PRs in reverse order, we found that our earlier results still held.

Finally, it is possible that the observed correlation of the difference in entropy with PR acceptability and debate is spurious, mirroring another factor of influence. We discussed and minimized an important threat of this category in Section 3.6, namely that certain types of PRs may both have high entropy and high rates of rejection. We found that all types of PRs display the same phenomenon (although to different extents), despite having different overall entropy scores, suggesting entropy is a general factor of influence in PR review.

External Validity The current study focuses only on popular Java projects that use the Github OSS platform. The aspect of popularity can be seen as a necessity for this approach to yield significant results. However, a number of potential threats to the external validity of the study follow. (1) The results may not be representative for other languages on Github (see also Section 3.6). (2) The results may not be representative for other OSS platforms. In particular, none of the projects in this study made use of pull requests for more than 40 months (when the feature was improved on Github). Other OSS projects, such as Eclipse, Mozilla, or many Apache projects, are both older and larger. (3) The results may not hold for industrial projects that make use of (modern) code review. Due to the financial concerns that play a role in such settings, investigating the potential of language models in identifying code style violations may lead to different results.

In general, our results held both on the entire set of projects and on the largest projects. For instance, we evaluated whether a significant difference in entropy exists between accepted and rejected PRs within the individual projects using a Benjamini-Hochberg test. We found a significant difference on 5 out of the 7 large projects ($p < 0.05$), but only on 1 out of the 15 small projects. This may in part be explained by the smaller projects having insufficient data points to achieve significant results. However, a number of the results presented in Table 3.2 held with significance and strong effect on the group of large projects but did not hold at all on the collection of small projects (as defined in Section 3.4.5). This suggests a size confound, where the larger (and generally also older) projects have a stronger sense of coding conventions, which is also in line with the results of a recent study by Tsay et al. [97]. We have dealt with this threat to the validity of our study by dividing the projects into the categories large and small and reporting results for both of these categories when
3. Evaluating Code Contributions With Language Models

applicable.

3.8 Conclusion

Source code contributions to OSS projects are evaluated through code reviews before being accepted and merged into the main development line. Surveying core members of popular projects on GitHub, Gousios et al. found that reviewers perceive conformance, in terms of style and quality, as the top factor when evaluating a code contribution [44]. In this paper, we extend this qualitative insights and use language models to quantitatively evaluating the influence of stylistic properties of code contributions on the code review process and outcome.

By analyzing 22 popular OSS projects on GitHub, totaling more than 6,000 code reviews, we found that (1) accepted code is significantly more similar to the project from a language model perspective than that rejected, while (2) highly dissimilar contributions receive a different treatment in code review, and (3) more debated ones are significantly less similar. Finally, (4) the contributed code shows a substantial increase in similarity as the contributor gains experience.

In this paper we make the following main contributions:

1) The first application of language models for an automated evaluation of properties of the submitted code on the code review outcome and process.

2) An extensive quantitative analysis on 22 OSS projects of the effect of similarity, from a language model perspective, between contributed code and project code on code review outcome and code review process.

3) A discussion of the implications of our findings with recommendations for future opportunities of research.
Chapter 4

Code Completion

4.1 Foreword

This chapter is based on the research paper: CACHECA: A Cache Language Model Based Code Suggestion Tool. This paper was accepted in the 2015 ICSE DEMO track, and was presented there by me. On this paper, Christine Franks was first author, I was lead author and the co-authors were Zhaopeng Tu and Prof. Premkumar Devanbu. I was in charge of writing following the initial draft, revised and (co-)wrote all sections. I collaborated with Christine Franks on the tool following her creation of the initial version, helped remove bugs and created the mixing algorithm for mixing with the Eclipse suggestions. I conducted the quantitative experiments reported in section 4.4 and presented the paper at the ICSE conference.

4.2 Introduction

The task of code completion is concerned with suggesting appropriate code tokens to a user during editing. Many Integrated Development Environments provide code suggestion, which can improve the programmers’ productivity. Generally, such engines deduce what tokens “might” apply in the current syntactic context based on pre-defined syntactic and semantic rules. In this way, the suggestions produced by these engines are typically valid (in terms of syntax and semantics), but not necessarily likely (in terms of real-world usage).

N-gram models have been shown to successfully enhance the built-in suggestion engine by using corpus statistics (i.e. by suggesting what most often applies) [47]. However, standard ngram models fail to deal with a special property of software: source code is also very localized [99]. Due to module specialization and focus, code tends to take especially repetitive forms in local contexts. The ngram approach, rooted as it is in NL, focuses on capturing the global regularities over the whole corpus, but fails to capture local regularities. To overcome this weakness, Tu et al. introduced a novel cache language model to capture the localness of source code [99].

The cache language model (denoted as $gram$) extends the traditional n-gram models by deploying an additional cache to capture regularities in the locality. The ngram and
C
ODE
COMPLETION

cache components capture different regularities: the ngram component captures the corpus linguistic structure, and offers a good estimate of the mean probability of a specific linguistic event in the corpus; around this mean, the local probability fluctuates, as code patterns change in different localities. The cache component models these local changes, and provides variance around the corpus mean for different local contexts. Technical details can be found in [99].

This paper presents the novel features and core architecture of CACHECA, an Eclipse plugin for code suggestion based on the cache language model. CACHECA (a portmanteau of Cache and Content Assist, Eclipse’s term for code suggestions) combines suggestions from two different sources: (1) the Eclipse built-in suggestion engine offers syntactic and semantic suggestions based on type information available in context; (2) a $gram model suggestion engine offers suggestions that commonly occur in either the whole corpus (from the global ngram model) or the local context (from the local cache model). Our experiments show that the $gram based suggestion tool greatly enhances Eclipse’s built-in engine by incorporating both the corpus and locality statistics, especially when no type information is available.

CACHECA is available for download as plugin from: http://macbeth.cs.ucdavis.edu/CACHECA/ and the source code is available at: https://github.com/christinef/CACHECA/.

4.3 Features

CACHECA operates within Eclipse’s editor and can be invoked either automatically (wherever the user has set Content Assist to normally be invoked) or manually (using Ctrl+Space). By leveraging the cache language model, it has three appealing features to enhance Eclipse’s built-in suggestion engine:

1. It is still able to offer suggestions even when no type information is available, making it applicable to both dynamic and static typed programming languages.

2. The suggestions and corresponding probabilities from the cache language model help to rerank and complement the original suggestions from the built-in plugin. In this way, CACHECA enforces a natural ordering on the suggestions.

3. Based on the observation that a token used in the immediate past is much more likely to be used again soon, CACHECA is able to capture the short-term shifts in token frequencies in the locality (e.g., a single file).

4.3.1 Natural Code Suggestion

Different projects have idiosyncratic token frequencies [47, 99]. Therefore, even given the same context, suggestions in different orders should be provided for different projects. This is overlooked by the built-in suggestion engine, and therefore by incorporating a language model engine, we can sort the suggestions for different projects using corpus statistics.
4.3. Features

Figure 4.1: CACHECA produces suggestions intelligently, based on localness.

4.3.2 Intelligent Code Suggestion

If a programmer uses a certain token when (s)he is coding, there is an increased likelihood of this token being used again in the near future. For example, the declaration of an identifier in a method scope makes usage of this identifier highly likely in the subsequent code. The cache component of CACHECA is designed to capture such localized regularity by estimating the probability of a suggestion from its recent frequency of use. Indeed, Tu
et al. [99] showed that, by adding a cache component, the accuracy of predicting identifiers is more than two times the accuracy using only the ngram model.

Consider the example in Figure 4.1. Unlike Eclipse’s default engine, CACHECA suggests the correct `getDocument()` method first, because it was invoked twice in the lines above it. The re-ranking of the suggestions (through the Mix model; see section IV-D for more detail) is one of Cacheca’s strong points; because Eclipse’s default relies on a low-granularity ranking system, mostly based on types, often the anticipated completion is ranked very low on the list (9th here). CACHECA, however, anticipates reappearances and ranks them higher.

### 4.3.3 Type Information

A drawback of Eclipse’s built-in suggestion engine is that it relies heavily on type information in the context. In dynamically typed languages, where type information is inherently unavailable, code suggestion based on semantic and syntactic information is particularly difficult. While CACHECA yields a substantial improvement for Java, there is also great potential for languages that are dynamically typed.

### 4.4 Architecture

CACHECA combines suggestions from two sources: the suggestions that Eclipse provides (ordered by a custom ‘relevance score’), and the suggestions from the ngram model (ranked by their probabilities).

#### 4.4.1 Methodology

In order to build the ngram model, the tool detects (by means of a listener) when a new file is visible (i.e. a new file opened, reopened, or brought to the front). At this point, the ngram model is read in from a file with word counts and a new, empty cache is generated. For statistics on time and memory usage, see [99].

When CACHECA is invoked, either by pressing a key character (such as the dot operator) or using Ctrl+Space, CACHECA starts by retrieving Eclipse’s suggestions, as described in section 4.4.2. Subsequently, the ngram suggestions are computed (as described in [99]) and the lists are mixed to obtain the proposal list. The mixing process is described in section 4.5.4 and Figure 4.2 provides a high level illustration. To preserve the ordering imposed on the suggestions by the mixing procedure, CACHECA includes a custom sorter (“CACHECA Sort”) which extends the JavaCompletionProposalSorters extension point. The final generated list is then displayed to the user by Eclipse.

Since the ngram model is estimated using only lexical code tokens, it cannot produce Javadoc material, nor can it provide parameter information or distinguish between data members and methods. However, utilizing the fact that Eclipse presents a laundry list of suggestions (albeit typically in a poor order), we search through Eclipse’s default list of suggestions to find matches to CACHECA’s suggestions. By using CACHECA’s ranking
4.4. Architecture

CACHECA Mix Model

Eclipse list + n-gram list

Case 1: either list has less than 3 (INCLUDE STEP 2)
Case 2: no list has less than 3 (SKIP STEP 2)

1. Occur within top 3 of both models
2. Top 1 from list(s) with less than 3
3. Interleave remaining (combining Eclipse with CACHECA where possible)

Figure 4.2: The architecture of CACHECA's Mix model.

preferences and Eclipse’s additional information, CACHECA provides a robust code completion system.

4.4.2 Implementation Details

CACHECA is written entirely in Java. Initially, the aim was to combine statistical modeling code (written in C++) and the plugin code (necessarily written in Java, due to the Eclipse plugin framework) using sockets and the Java Native Interface (JNI). This approach was unsuccessful due to cross-platform limitations and latency issues, after which the decision was made to convert the existing modeling code to Java. Due to restrictions placed on plugins by the Eclipse extension framework, CACHECA must ‘collect’ Eclipse’s unordered list of completion suggestions from the compilation unit and then sort the suggestions exactly as Eclipse does internally (i.e. first by the internal ‘relevance’ value and then alphabetically). This results in an ordered list exactly like that which Eclipse would display if unaided, which we can then incorporate into the Mix model. Ideally, a future iteration of CACHECA would place this tool inside Eclipse itself, eliminating overhead.
4. Code Completion

4.5 Results

The goal of CACHECA is to combine the advantages of the two core suggestors: Eclipse’s Java Proposals and the $gram model. We seek to make use of the different information that is captured by these approaches to yield superior suggestion performance over Eclipse alone.

4.5.1 Data

We evaluated our models on a corpus of token sequences in seven Java projects of varying sizes, which was used previously for evaluation of code completion performance: Ant, Batik, Cassandra, Log4J, Maven2, Maven3 and Xalan \[47, 67, 99\]. The included projects were cloned from public Git repositories between December 2009 and January 2011 and contain between 60KLOC and 367KLOC for a total of 5 million tokens. On each project, we used 10-fold validation by training our model on 90% of the lines, counting the token sequences, and testing on the token sequences in the remaining 10% of the lines. Each test-case can be seen as a scenario, where a certain history of tokens is known and the model is tasked with predicting the next token.

We specifically excluded a number of tokens from the task of suggesting, namely all parentheses and the symbols: ; . = as these symbols are used to trigger a completion in Eclipse. Tu et al. \[99\] showed that separators and operators such as these are relatively easy to predict and make up a large part of source code. Therefore, we expect the completion performance of the cache model to be lower than the overall scores reported in their work and more in line with the performance reported on identifier and keyword completion.

4.5.2 Metrics

In each testing scenario (a context of tokens from which the suggestors must predict the next token), each suggestor returns a list of suggestion in ranked order, which may be empty and is cut off at ten suggestions. The rank of the correct suggestion in this list is used as a measure of performance. Following the common settings in the code suggestion task \[47, 67, 99\], we measure the performance of the suggestors in terms of both MRR score and TopK accuracies. The TopK accuracies are computed as the percentage of contexts in which the suggester ranks the correct suggestion at index $K$ or less. We report accuracies for $K = 1$, 5 and 10. The Mean Reciprocal Rank (MRR) score is computed by averaging the reciprocal of the rank of the correct suggestion over all suggestion tasks in the test data. If the correct suggestion is not returned, a score of 0 is assigned. Hence, an MRR score of 0.5 implies that the correct suggestion is on average found at index 2 of the suggestion list.

4.5.3 Models

We have two baseline models: the Eclipse content assist and the $gram suggestor. The Eclipse suggestions are retrieved as described in section \[4.4.2\]. The $gram suggestor consists of the $n$-gram and cache components as described in \[99\].
Both models produce an ordered list of suggestions based on a score system: a ‘relevance score’ for the Eclipse model and a probability for the $gram$ model. The former is computed based on a large number of hard-coded weights, linked to possible properties that a suggestion can have. Besides the lack of justification for these (rather diverse) weights, the relevance scores for different suggestions were often identical (e.g., the same weight for all possible method invocations on an object). Beyond the ordering, we were unable to gain much insight from the relevance scores.

The output of CACHECA is a new model, named Mix, which mixes the suggestions returned by the baseline models. The aim is to leverage information that is captured by the individual models and use this to improve suggestion accuracy. We constructed the Mix model based on a case study on the Ant project (section 4.5.4) and verified the results across the entire dataset.

4.5.4 Mixing

As shown in Table 4.1, the $gram$ model (model 2) performs approximately twice as well on MRR score and even better in Top1 accuracy as the Eclipse model (model 1). Hence, we compare the performance of the Mix model with the stronger baseline, the $gram$ model, in the following experiments. We construct the Mix model based on a collection of heuristics. Each heuristic is applied in order and will take zero or more items from both of the suggestors and add these to the mix. The scores for the Mix model with one or more heuristics (in order) are listed in table 4.1 as Mix (H$_1$, ···, H$_j$).

**Heuristic 1: interleave suggestions of the baseline models.**

At the core of the Mix model is a simple interleaving of the results, which starts with the best suggestion of the $gram$ model (the alternative performs worse than 2.). As can be seen from the results of model 3 in table 4.1, interleaving yields superior MRR scores, primarily by improving Top5 accuracy. To improve the mix model further, we add two heuristics based on properties of the models under consideration. Due to its general nature, interleaving is always executed last.

**Heuristic 2: If either model returns no more than 3 suggestions, add the first element to the mix.**

In the case of the $gram$ model, a shorter list indicates that the context was only seen in the cache and is therefore local to the code. Similarly, when encountering an unusual context, Eclipse will only suggest results that match the needed type of a variable. We found that top suggestions in shorter lists were significantly more likely to be correct for both models, and that a large part of the improvement in performance of H2 over H1 is due to Eclipse suggestions, since simple interleaving already prioritizes the top suggestion from the $gram$ model. As can be seen from table 4.1 (model 4), this heuristic substantially improves Top1 suggestions and the MRR score.

**Heuristic 3: If a suggestion occurs among the top 3 suggestions of both baseline models, add it to the mix.**

This heuristic gives extra weight to an element that occurs near the top of both baseline models. We use the ‘top 3’ criterion to ensure that the overlap is meaningful, i.e. that the
<table>
<thead>
<tr>
<th>Model</th>
<th>MRR</th>
<th>Top1</th>
<th>Top5</th>
<th>Top10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baseline</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Eclipse</td>
<td>24.35%</td>
<td>18.23%</td>
<td>33.30%</td>
<td>39.77%</td>
</tr>
<tr>
<td>2. $gram</td>
<td>48.11%</td>
<td>38.95%</td>
<td>60.40%</td>
<td>67.10%</td>
</tr>
<tr>
<td><strong>Mix</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Mix (H1)</td>
<td>49.76%</td>
<td>38.95%</td>
<td>65.04%</td>
<td>73.58%</td>
</tr>
<tr>
<td>4. Mix (H2, H1)</td>
<td>51.19%</td>
<td>41.24%</td>
<td>65.44%</td>
<td>73.98%</td>
</tr>
<tr>
<td>5. Mix (H3, H1)</td>
<td>51.44%</td>
<td>40.76%</td>
<td>66.41%</td>
<td>75.05%</td>
</tr>
<tr>
<td>6. Mix (H3, H2, H1)</td>
<td>52.29%</td>
<td>42.34%</td>
<td>66.40%</td>
<td>75.05%</td>
</tr>
</tbody>
</table>

Table 4.1: Accuracy with various settings on Ant.

suggestion is deemed likely by both models (particularly for $gram$) and is not simply part of a group of suggestions that are all deemed equally accurate (particularly for Eclipse). From table 4.1 we see that H3 yields slightly higher MRR performance and slightly lower Top1 performance than H2. Applying first H3 and then H2 (model 6) yields superior performance in both regards.

The results were consistent across the tested projects. CACHECA achieved an average improvement of 29.83 percent points over Eclipse’s Content Assist in terms of MRR. Top10 accuracy improved by an average of 34.81 percent points and Top1 accuracy more than doubled from 25.36% to 55.19%.

Finally, although the prediction performance of the Mix model is dominated by the $gram$ model, the Eclipse suggestions play another important role. As was observed in 4.4.1, the $gram$ model can only produce lexical completions, whereas the Eclipse suggestor typically returns template completions. Hence, for each suggestion returned by the $gram$ model we first attempt to find a matching template from an extended list of Eclipse suggestions (the correct completion is often contained in the top 30) and return this template instead. This strategy proves successful in approximately half of the cases.

### 4.6 Related Work

Another Eclipse plugin, dubbed Calcite[61], operates on the same assumption as CACHECA (that more commonly-used suggestions should rank higher) but attempts to accomplish this by capturing the personalized usage across groups of programmers through crowdsourcing, rather than capturing the local regularities. In particular, Calcite presents suggestions above and below Eclipse’s, whereas CACHECA generates a blended list of suggestions. Bruch et al [17] describe a plugin that suggests method calls, based on co-occurrence statistics; CACHECA is not restricted to method calls. A follow-on work [16] also uses a “social suggestion” feature where actual uses by real users are monitored and used to improve performance; this approach is potentially complementary to CACHECA. GraPacc [66], another code Suggestion plugin, focuses on APIs and incorporating the popularity of certain Suggestions. There are several other suggestion engines; we refer the reader to longer papers [47, 17, 88] for a more complete literature survey.
4.7 Conclusions

We created a plugin for Eclipse, CACHECA, that combines the default suggestions with suggestions returned by a cache language model [99]. We found that the cache language model yields substantially superior performance to the Eclipse suggestion engine. Furthermore, by combining the suggestions returned by both models, CACHECA improves the Top1 accuracy of suggestions by 26.43 percent points and the Top10 accuracy by 34.81 percent points. We demonstrate that the cache language model introduced in [99] has practical application and that there is substantial potential for CACHECA to improve code suggestion. Building on the results, we suggest extension to dynamically-typed languages. This material is based on work supported by NSF under Grants 1247280 and 1414172.
Chapter 5

Fault Detection

5.1 Foreword

This chapter is based on the research paper: On the “Unnaturalnes” of Buggy Code. It was originally submitted to the 2015 Foundations of Software Engineering Conference, but was rejected; a revised version of this paper is scheduled for the 2016 ICSE conference. Co-authors on this paper were Zhaopeng Tu, Connie Nguyen, Saheel Godhane, Dr. Alberto Bacchelli and Prof. Premkumar Devanbu. My role in this paper, as shared first author, included conducting the experiments related to RQs 3-5, (co-)writing sections 5.3, 5.4, 5.5 and reviewing writing of sections 5.1, 5.2, 5.6 and 5.7. I suggested the usage of the $gram+type and $gram+wType models.

5.2 Introduction

Communication is ordinary, everyday human behavior, something we do naturally. This “natural” linguistic behavior is characterized by efficiency and fluency, rather than creativity. Most natural language (NL) is both repetitive and predictable, thus enabling humans to communicate reliably & efficiently in potentially noisy and dangerous situations. This repetitive property, i.e. naturalness, of spoken and written NL has been exploited in the field of NLP: Statistical language models (from hereon: language models) have been employed to capture it, and then use to good effect in speech recognition, translation, spelling correction, etc.

As it turns out, so it is with code! People also write code using repetitive, predictable, forms: Recent work [47] showed that code is amenable to the same kinds of language modeling as NL, and language models have been used to good effect in code suggestion [47, 84, 99], cross-language porting [67 64 69 50], coding standards [3], idiom mining [6], and code de-obfuscation [83]. Since language models are useful in these tasks, they are capturing some property of how code is supposed to be. This raises an interesting question: What does it mean when a code fragment is considered improbable by these models?

Language models assign higher naturalness to code (tokens, syntactic forms, etc.) frequently encountered during training, and lower naturalness to code rarely or never seen.
In fact, prior work [19] showed that syntactically incorrect code is flagged as improbable by language models. However, by restricting ourselves to code that occurs in repositories, we still encounter unnatural, yet syntactically correct code; why? We hypothesize that
unnatural code is more likely to be wrong, thus, language models actually help zero-in on potentially defective code; in this paper, we explore this.

To this end, we consider a large corpus of 8,296 bug fix commits from 10 different projects, and we focus on its language statistics, evaluating the naturalness of defective code and whether fixes increase naturalness. Language models can rate probabilities of linguistic events at any granularity, even at the level of characters. We focus here on line-level defect analysis, giving far finer granularity of prediction than traditional defect prediction methods, which most often operate at the granularity of files or modules. In fact, this approach is more commensurate with static analysis or static bug-finding tools, which also indicate potential bugs at line-level. For this reason, we also investigate our language model approach in contrast and in conjunction with two well-known static bug finders (namely, PMD [25] and FindBugs [33]).

Overall, our results corroborate our initial hypothesis that code with bugs tends to be more unnatural. In particular, the main findings of this paper are:

1. Buggy code is rated as significantly more “unnatural” (improbable) by language models.
2. This unnaturalness drops significantly when buggy code is replaced by fix code.
3. Using cost-sensitive measures, inspecting “unnatural” code indicated by language models works quite well: Performance is comparable to that of static bug finders FindBugs and PMD.
4. Ordering warnings produced by the FindBugs and PMD tools, using the “unnaturalness” of associated code, significantly improves the performance of these tools.

Our experiments are mostly done with Java projects, but we have strong empirical evidence indicating that the first two findings above generalize to C as well; we hope to confirm the rest in future work.

5.3 Background

Our main goal is evaluating the degree to which defective code appears “unnatural” to language models, and to what extent language models can actually enable programmers to zero-in on bugs during inspections. Furthermore, if language models can actually help direct programmers towards buggy lines, we are interested to know how they compare against static bug-finding tools. In this section, we present relevant technical background and the main research questions. We begin with a brief technical background on language modeling.

5.3.1 Language Modeling

**Basics.** Language models are statistical models that assign a probability to every sequence of words. Given a code sequence \( S = t_1t_2 \ldots t_N \), a language model estimates the probability
of this sequence occurring as a product of a series of conditional probabilities for each token:

$$P(S) = P(t_1) \cdot \prod_{i=2}^{N} P(t_i|t_1, \ldots, t_{i-1})$$  (5.1)

Each probability $P(t_i|t_1, \ldots, t_{i-1})$ denotes the chance that the token $t_i$ follows the previous tokens, the prefix, $h = t_1, \ldots, t_{i-1}$. In practice, however, the probabilities are impossible to estimate, as there is an astronomically large number of possible prefixes. The most widely used approach to combat this problem is to use the ngram language model, which makes a Markov assumption that the conditional probability of a token is dependent only on the $n-1$ most recent tokens. The ngram model places all prefixes that have the same $n-1$ tokens in the same equivalence class:

$$P_{ngram}(t_i|h) = P(t_i|t_{i-n+1}, \ldots, t_{i-1})$$  (5.2)

The latter is estimated from the training corpus as the fraction of times that the prefix $t_{i-n+1}, \ldots, t_{i-1}$ was followed by the token $t_i$. Note that, given a complete sentence, we can also compute each token given its epilog (the subsequent tokens), essentially computing the probability of the sentence in reverse. We make use of this approach to better identify buggy lines, as described in Section 5.4.3.

The ngram language models have been shown to successfully capture the highly repetitive regularities in source code, and were applied to code suggestion tasks [47]. However, the ngram models fail to deal with a special property of software: source code is also very localized. Due to module specialization and focus, code tends to take special repetitive forms in local contexts. The ngram approach, rooted as it is in NLP, focuses on capturing the global regularities over the whole corpus, and neglects local regularities, thus ignoring the localness of software. To overcome this, Tu et al. [99] introduced a cache language model to capture the localness of code.

**Cache language models.** These models (for short: $\$gram) extend the traditional language models by deploying an additional cache to capture the regularities in the locality. It combines the global (ngram) model with the local (cache) model as

$$P(t_i|h, cache) = \lambda \cdot P_{ngram}(t_i|h) + (1 - \lambda) \cdot P_{cache}(t_i|h)$$  (5.3)

$cache$ is the list of ngrams extracted from the local context, and $P_{cache}(t_i|h)$ is estimated from the frequency with which $t_i$ followed the prefix $h$ in the cache. To avoid hand-tuned parameters, Tu et al. [99] replaced the interpolation weight $\lambda$ with $\gamma/(\gamma + H)$, where $H$ counts the times the prefix $h$ has been observed in the cache, and $\gamma$ is a concentration parameter between 0 and infinity.

$$P(t_i|h, cache) = \frac{\gamma}{\gamma + H} \cdot P_{ngram}(t_i|h) + \frac{H}{\gamma + H} \cdot P_{cache}(t_i|h)$$  (5.4)

If the prefix occurs few times in the cache ($H$ is small), then the ngram model probability will be preferred; vice versa. This setting makes the interpolation weight self-adaptive for different ngrams.
The \textit{n}gram and cache components capture different regularities: the \textit{n}gram component captures the corpus linguistic structure, and offers a good estimate of the mean probability of a specific linguistic event in the corpus; around this mean, the local probability fluctuates, as code patterns change in different localities. The cache component models these local changes, and provides variance around the corpus mean for different local contexts.

We use a \textit{S}gram here to judge the “improbability” (measured as cross-entropy) of lines of code; the core research questions being, can cross-entropy provide a useful indication of the likely bugginess of a line of code, and how does this approach performs against/with comparable approaches, such as static bug finders.

### 5.3.2 Static Bug-finders (\textit{SBF})

The goal of \textit{SBF} is to use syntactic and semantic properties of source code to indicate locations of common errors, such as undefined variables and buffer overflows. They rely on methods ranging from informal heuristic pattern-matching to formal algorithms with proven properties. These tools typically report warnings at build time; programmers can choose to fix them. Pattern-matching tools (e.g., PMD and FindBugs [25, 33]) are unsound, but fast and widely used; more formal approaches are sound, but slower. In practice all tools have false positives and/or false negatives, thus inspecting and fixing all the warnings is not always cost-effective.

Suffice for our purposes to note here that both \textit{SBF} and \textit{S}gram both (fairly imperfectly) indicate likely locations of defects; so our goal here is to compare these rather different approaches, and see if they synergize. It should be noted that \textit{S}gram is quite easy to implement, since it requires only lexical information; however, as we see below it can be improved with some syntactic information.

<table>
<thead>
<tr>
<th>Project</th>
<th>Study Period</th>
<th>S*</th>
<th>#Files</th>
<th>NCSL</th>
<th>Changes</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmosphere</td>
<td>May-10 to Jan-14</td>
<td>17</td>
<td>17,206</td>
<td>6,329,400</td>
<td>2,481</td>
<td>1,130</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>Feb-10 to Jan-14</td>
<td>17</td>
<td>103,727</td>
<td>22,156,904</td>
<td>4,922</td>
<td>1,077</td>
</tr>
<tr>
<td>Facebook-android -sdk (fdk)</td>
<td>May-10 to Dec-13</td>
<td>16</td>
<td>3,981</td>
<td>1,431,787</td>
<td>320</td>
<td>143</td>
</tr>
<tr>
<td>Netty</td>
<td>Aug-08 to Jan-14</td>
<td>24</td>
<td>57,922</td>
<td>12,969,858</td>
<td>3,906</td>
<td>1,485</td>
</tr>
<tr>
<td>Presto</td>
<td>Aug-12 to Jan-14</td>
<td>23</td>
<td>57,922</td>
<td>12,969,858</td>
<td>3,906</td>
<td>1,485</td>
</tr>
<tr>
<td>Derby</td>
<td>Sep-04 to Jul-14</td>
<td>41</td>
<td>143,906</td>
<td>61,192,709</td>
<td>5,275</td>
<td>1,453</td>
</tr>
<tr>
<td>Lucene</td>
<td>Sep-01 to Mar-10</td>
<td>36</td>
<td>47,270</td>
<td>11,744,856</td>
<td>2,563</td>
<td>469</td>
</tr>
<tr>
<td>OpenJPA</td>
<td>May-06 to Jun-14</td>
<td>34</td>
<td>131,441</td>
<td>27,709,778</td>
<td>2,956</td>
<td>558</td>
</tr>
<tr>
<td>Qpid</td>
<td>Sep-06 to Jun-14</td>
<td>33</td>
<td>94,790</td>
<td>24,031,170</td>
<td>3,362</td>
<td>657</td>
</tr>
<tr>
<td>Wicket</td>
<td>Sep-04 to Jun-14</td>
<td>41</td>
<td>159,332</td>
<td>28,544,601</td>
<td>10,583</td>
<td>994</td>
</tr>
<tr>
<td>Overall</td>
<td>Sep-01 to Jul-14</td>
<td>266</td>
<td>782,661</td>
<td>202,607,212</td>
<td>38,003</td>
<td>8,296</td>
</tr>
</tbody>
</table>

Table 5.1: Summary data of projects that are analyzed for finding all the defects including development time bugs. The first five projects were taken from the Github ecosystem, the second five were retrieved from the Apache ecosystem.

### 5.3.3 Evaluating Defect Predictions

In our setting, we view \textit{SBF} and \textit{S}gram as two commensurate approaches to selecting lines of code to which drawing the programmers’ attention as locations worthy of inspection,
since they just might contain real bugs. To emphasize this similarity, from here on we refer to language model based bug prediction as $\mathcal{NBF}$ (“Naturalness Bug Finder”). With either $\mathcal{SBF}$ or $\mathcal{NBF}$, programmers will spend effort on reviewing the code and hopefully find some defects. Comparing the two approaches requires a performance measure. We adopt a cost-based measure that has become standard [7]: AUCEC (Area Under the Cost-Effectiveness Curve). AUCEC (like ROC) is a non-parametric measure, which does not depend on the defects’ distribution. AUCEC assumes that the cost is the inspection effort and the payoff is the count of bugs found.

We normalize both to 100%, measure the payoff increase as we inspect more and more lines and draw a ‘lift-chart’ or Lorenz curve. AUCEC is the area under this curve. Suppose we inspect x% code at random; in expectation, we would find x% of the bugs, thus yielding a diagonal line on the lift chart; so the expected AUCEC if inspecting 5% lines at random would be $0.00125^1$. Typically, inspecting 100% code is very expensive; one could reasonably assume that 5% or even just 1% of the code, in a large system, could realistically be inspected; therefore, we compare AUCECs for $\mathcal{NBF}$ and $\mathcal{SBF}$ for this much smaller proportion.

Additionally, we investigate defect prediction performance under several credit criteria. A prediction model is awarded credit, ranging from 0 to 1, for each line marked as defective. Previous work by Rahman et al. has compared $\mathcal{SBF}$ and $\mathcal{DP}$ (a file level statistical defect predictor) models using two types of credit: full (or optimistic) and partial (or scaled) credit [79], which we adapt to line level defect prediction. The former metric awards a model one credit point for each bug iff at least one line of the bug was marked buggy by the model. Thus, it assumes that a programmer will spot a bug as soon as one of its lines is identified as such. Partial credit is more conservative: For each bug, the credit awarded to the model is the fraction of the bug’s defective lines that the model marked. Hence, partial credit assumes that the probability of a developer finding a bug is proportional to the fraction of the bug that is marked by the defect prediction model.

5.3.4 Research Questions

At the core of our research is the question whether “unnaturalness” (measured as entropy, or improbability) is indicative of poor code quality. The abundant history of changes (including bug fixes) in OSS projects allows the use of standard methods [93] to find code that was implicated in bug fixes (“buggy code”).

| RQ1. Are buggy lines less “natural” than non-buggy lines? |

In project histories, we can find numerous samples of bug fixes, where buggy code is replaced by bug-fix code to correct defects. Do language models rate bug-fix code as more natural than the buggy code they replaced? This would essentially mean that the bug fix code is assigned a higher probability than the buggy code. Such a finding would also have

---

1Calculated as $0.5 \times 0.05 \times 0.05$. This could be normalized differently; but we consistently use this measurement, so our comparisons work.
implications for automatic, search-based bug repair: If fixes tend to have higher probability, then a good language model might provide an effective organizing principle for the search, or perhaps (if the model is generative) even generate possible candidate repairs.

**RQ2. Are buggy lines less “natural” than bug-fix lines?**

Even if defective lines are indeed more often fingered as unnatural by language models, it is likely to be an unreliable indication; thus one can expect many false positives (correct lines indicated as unnatural) and false negatives (buggy lines indicated as natural). It would be interesting to know, however, how well naturalness (*i.e.* entropy) is a good ordering principle for directing inspection.

**RQ3. Is “naturalness” a good way to direct inspection effort?**

One can view ordering lines of code for inspection by ‘naturalness’ as a sort of defect-prediction technique; we are inspecting lines in a certain order, because prior experience suggests that certain code is very improbable, and thus possibly defective. Traditional defect-prediction techniques typically rely on historical process data (*e.g.*, number of authors, previous changes and bugs); however, defectiveness is predicted at the granularity of files (or methods), thus, it is reasonable to compare naturalness as an ordering principle with $SBF$, which provide warnings at the line level.

**RQ4. How do $SBF$ and $NB$ compare in terms of ability to direct inspection effort?**

It is reasonable to expect that, if $SBF$ provides a warning on a line and it appears unnatural to a language model, then it is even more likely a mistake. We therefore investigate whether naturalness is a good ordering for warnings provided by static bug-finders.

**RQ5. Is “naturalness” a useful way to focus the inspection effort on warnings produced by $SBF$?**

### 5.4 Methodology

In this section, we describe the projects that we studied and our approaches to data gathering and analysis.
5.4. Methodology

<table>
<thead>
<tr>
<th>Project</th>
<th>NCSL #K</th>
<th>#Warnings #K</th>
<th>FindBug PMD #Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby (7)</td>
<td>420–630</td>
<td>1527–1688</td>
<td>140–192K 89–147</td>
</tr>
<tr>
<td>Lucene (7)</td>
<td>68–178</td>
<td>137–300</td>
<td>12–31K 24–83</td>
</tr>
<tr>
<td>OpenJPA (7)</td>
<td>152–454</td>
<td>51–340</td>
<td>62–171K 36–104</td>
</tr>
</tbody>
</table>

Table 5.2: Summary data of projects that are analyzed for locating bugs reported in issue database. The dataset is taken from Rahman et al.

5.4.1 Study Subject

We studied 10 OSS java projects, as shown in Table 5.1: Among them Atmosphere (an asynchronous web socket framework), FDK (an Android SDK for building Facebook application), Elasticsearch (a distributed search engine for cloud), Netty (an asynchronous network application framework), and Presto (a distributed SQL query engine) are Github projects, while Derby (a relational database), Lucene (a text search engine library), OpenJPA (a Java Persistence API), Qpid (a messaging system), and Wicket (a lightweight web application framework) are taken from Apache Software Foundation. We deliberately chose the projects from different application domains to measure \( NBF \)’s performance in various types of systems. The Apache projects are relatively older; Lucene, the oldest one, started in 2001. The earliest Github project in our dataset (Netty) started in 2008. All projects are under active development.

We analyzed \( NBF \)’s performance on this data set in two settings. In the first setting (see Phase-I in Section 5.4.2) we consider all the bugs—both development time and post release—that have appeared in the project’s evolution. The performance is analyzed at different stages of each project’s evolutionary history. We extracted snapshots of individual projects at an interval of 3 months from the version history. Such snapshots represent the current states of the projects at that time period (see Section 5.4.2 for details). In total, we analyzed 266 snapshots across 10 projects that include 782,661 distinct file versions, and 202.6 Million total non-commented source code lines (NCSL). These snapshots contain 38,003 distinct commits, of which 8,296 were marked as bug fixing changes using the procedures outlined in Section 5.4.2. The corresponding bugs include both development-time bugs as well as post-release bugs.

In the second setting, we only focus on post-release bugs that are reported in an issue tracking system. We used the data set prepared by Rahman et al. [79], in which snapshots of the five Apache projects were taken at selected project releases. At each snapshot, the project size varies between 68 and 630K NCSL. The bugs were extracted from Apache’s JIRA issue tracking system and the total number of bugs reported against each release across all the projects varies from 24–194. Table 5.2 summarizes this dataset.

At each release version, Rahman et al. further collected warnings produced by two static bug finding tools, namely FINDBUGS [8] and PMD [25]. PMD operates on source code and produces line-level warnings; FINDBUGS operates on Java bytecode [8] and reports warning at line, method, and class level. For this reason FINDBUGS produces warnings covering significantly more lines, though the number of unique warnings is smaller than that of PMD (see Table 5.2). To make the comparisons between \( NBF \) and \( SBF \) fair, we...
further filtered out warnings for commented lines, since BF’s entropy calculation does not consider the commented lines. In fact, we have noticed that a majority of FINDBUGS line-level warnings are actually commented lines. Thus after removing comments, we are left with primarily method and class level FINDBUGS warnings.

5.4.2 Data Collection

As mentioned earlier, our experiment has two distinct phases. First, we describe the process of collecting data for Phase-I, which tries to locate all the bugs that developers fix during an ongoing development process. Next, we briefly summarize data collection of Phase-II, which locates bugs at project release time; this data set is taken from Rahman et al. [79].

Figure 5.1: Phase I Data Collection: note Project Time Line, showing snapshots (vertical lines) and commits (triangles) at c1...c4. For every bugfix file commit (c3) we collect the buggy version and the fixed version, and use \texttt{diff} to identify buggy \& fixed lines.

**Phase-I.** All our projects used Git; we downloaded a snapshot of each project at 3-month intervals, beginning with the project’s inception. A snapshot represents the project’s state at that point of time, as shown by the dashed vertical lines in Figure 5.1. Then we retrieve all the commits (c’s in the Figure) made between each pair of consecutive snapshots. Each commit involves an old version of a file and its new version. Using \texttt{git diff} we identify the lines that are changed between the old and new versions. We also collected the number of deleted and added lines in every commit. We then removed the commits comprising more than 30 deleted lines (30 lines was at the 3\textsuperscript{rd} quartile of the sample of deleted lines per commit in our data set). We further removed the commits with no deleted lines, because we are only interested in locating buggy lines present in the old versions. Figure 5.2 shows a histogram of number of deleted lines per file commit; it ranges from at least one line to at max 30 lines deletion per commit with a median at 5.

Each commit has an associated commit log. We mark a commit as bugfix, if the corresponding commit log contains at least one of the error related keywords: ‘error’, ‘bug’, ‘fix’, ‘issue’, ‘mistake’, ‘incorrect’, ‘fault’, ‘defect’ and ‘flaw’, as proposed by Mockus and

\[ \text{In the unfiltered data set the median was at 2} \]
5.4. Methodology

Votta [60]. In this step, we first convert each commit message to a bag-of-words; we then remove words that appear only once among all of the bug fix messages to reduce project specific keywords; finally, we stem the bag-of-words using standard natural language processing (NLP) techniques. This method was taken from our previous work [82]. The deleted lines corresponding to the old version of a bugfix commit are marked as buggy lines. The added lines associated with new corrected version are marked as fixed lines.

Thus, from three sets of files (the files that did not change between two snapshots and the old and new versions of the changed files), we retrieve three sets of lines: (1) unchanged lines: all lines of the unchanged files and unchanged lines of the changed files. (2) buggy lines: lines that were corrected in the old version of the bugfix commits, and (3) fixed lines: lines that were fixed in the new version of the bugfix commits. In total, we compared 58,374,475 unchanged lines, 88,058 buggy lines, and 204,242 fixed lines across all the snapshots of all the projects.

Phase-II. To begin with, certain release versions of each Apache project were selected. Then, from the JIRA issue tracking system of the Apache projects, the post-release bugfix commits (corresponding to the selected release) were identified. Next, by blaming the old buggy file version associated with a bugfix commit using git blame, the corresponding buggy lines were detected. Since in this phase we are interested in locating post-release bugs, the identified buggy lines were further mapped to the release time file version using an adopted version of SZZ algorithm [93]. For each project release version, final outcome of Phase-II is two sets of lines: (1) buggy lines: lines that were marked as buggy lines based on post release fix and (2) non-buggy lines: all the other lines across all the Java files present at the release version.

Figure 5.2: Histogram of the number of lines deleted per file commit. The mean is 5, marked by the dashed line
5. FAULT DETECTION

5.4.3 Measuring entropy using cache language model

Entropy of code snippets. We measure the naturalness of a code snippet using statistical language model with a widely-used metric – cross-entropy (entropy in short) [47, 3]. The key intuition is that snippets that are more like the training corpus (i.e. more natural) would be assigned higher probabilities or lower entropy from an LM trained on the same corpus. Given a snippet \( S = t_1 \ldots t_N \), of length \( N \), with a probability \( P_M(S) \) estimated by a language model \( M \). The entropy of the snippet is calculated as:

\[
H_M(S) = -\frac{1}{N} \log_2 P_M(S) = -\frac{1}{N} \sum_{i=1}^{N} \log_2 P(t_i|h) \tag{5.5}
\]

\( P(t_i|h) \) is calculated by the cache language model via Equation 5.4.

Building a Cache Language Model. For each project and each pair of snapshots, we are interested in the entropy of lines that were marked as buggy in some commit between these snapshots. We would like to contrast these entropy scores with those of lines that were not changed in any bug-fix commit in this same period. To compute these entropy scores, for each file, we first train a language model on the ‘old’ version of all other files (the version at the time of the previous snapshot), counting the sequences of tokens of various lengths; we then run the language model on the current file, computing the entropy of each token based on both the prolog (the preceding tokens in the current file) and epilog (the succeeding tokens); finally, we compute the entropy of each line as the average of the entropy of each token on that line.

As an optimization step, we divided all ‘old’ versions of files into ten bins. Then, whenever testing on a file, we use the pre-counted training set on the nine bins that the old version of the current file is not in. This removes the need to compute a training set for each file separately. Since we use the cache-based language model, the entropy scores within each file are calculated using both the training set on the other nine bins and a locally estimated cache, built on only the current file, since Tu et al. [99] reported that building cache on both the prolog and epilog achieves best performance.

Determining parameters for cache language model. Several factors of the locality would affect the performance of cache language model [99]: cache context, cache scope, cache size, and cache order. For the fault localization task, we build the cache on all the existing code in the current file. In this light, we only need to tune the cache order (i.e. the maximum and minimum order of n-grams stored in the cache). In general, longer n-grams are more reliable but quite rare, thus we back-off to shorter matching prefixes [51] when needed. We follow Tu et al. [99] to set the maximum order of cache n-grams to 10. To determine the minimum back-off order, we performed experiments on the Elasticsearch and Netty projects to find the optimal performance, measured in terms of difference in entropy between buggy and non-buggy lines (Figure 5.3). The figure shows entropy difference with varying minimum backoff order and three different backoff weights (increasing, decreasing, no-change). We observed maximum difference in entropy between buggy and non-buggy
5.4. Methodology

5.4.4 Adjusting the entropy scores

An important assumption underlying the applicability of language models to defect prediction is that higher entropy is associated with bug proneness. In practice, buggy lines are quite rare, thus a few non-buggy lines with high entropy scores could substantially increase false negatives and worsen performance. We undertook some tuning efforts to sharpen NBF’s prediction ability.

Figure 5.3: Determining parameters of cache model. The experiments were conducted on Elasticsearch and Netty projects for one-line bugfix changes. Y axis represents difference of entropy of a buggy line w.r.t. non-buggy lines in the same file.

lines at minimum backoff order of 4 with no change in the backoff weight. Thus, we set the minimum backoff order be 4 and the backoff weight be 1.0.

lines at minimum backoff order of 4 with no change in the backoff weight. Thus, we set the minimum backoff order be 4 and the backoff weight be 1.0.
### Table 5.3: Buggy lines, in general, have higher entropy than non-buggy lines. Difference is measured with t-test for 95% confidence interval, and effect is Cohen’s D. Wilcox non-parametric test also confirmed buggy lines have higher entropy with statistical significance. ‘max delete’ represents maximum number of buggy lines that are fixed in a file commit.

<table>
<thead>
<tr>
<th>project</th>
<th>max delete = 2 difference</th>
<th>max delete = 2 effect</th>
<th>max delete = 5 difference</th>
<th>max delete = 5 effect</th>
<th>max delete = 10 difference</th>
<th>max delete = 10 effect</th>
<th>max delete = 20 difference</th>
<th>max delete = 20 effect</th>
<th>max delete = 30 difference</th>
<th>max delete = 30 effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>atmosphere</td>
<td>1.17 to 1.69</td>
<td>0.50</td>
<td>0.94 to 1.23</td>
<td>0.38</td>
<td>0.74 to 0.94</td>
<td>0.30</td>
<td>0.36 to 0.53</td>
<td>0.16</td>
<td>0.38 to 0.53</td>
<td>0.16</td>
</tr>
<tr>
<td>derby</td>
<td>1.56 to 1.96</td>
<td>0.56</td>
<td>1.63 to 1.85</td>
<td>0.55</td>
<td>1.32 to 1.47</td>
<td>0.44</td>
<td>1.01 to 1.13</td>
<td>0.34</td>
<td>0.84 to 0.94</td>
<td>0.28</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>1.58 to 1.90</td>
<td>0.57</td>
<td>1.37 to 1.53</td>
<td>0.48</td>
<td>1.01 to 1.14</td>
<td>0.35</td>
<td>0.84 to 0.95</td>
<td>0.29</td>
<td>0.73 to 0.82</td>
<td>0.25</td>
</tr>
<tr>
<td>fdk</td>
<td>0.74 to 1.64</td>
<td>0.40</td>
<td>1.33 to 1.83</td>
<td>0.53</td>
<td>1.03 to 1.41</td>
<td>0.41</td>
<td>1.04 to 1.33</td>
<td>0.40</td>
<td>0.92 to 1.17</td>
<td>0.35</td>
</tr>
<tr>
<td>lucene</td>
<td>1.27 to 1.80</td>
<td>0.48</td>
<td>0.97 to 1.28</td>
<td>0.35</td>
<td>0.83 to 1.06</td>
<td>0.30</td>
<td>0.97 to 1.14</td>
<td>0.33</td>
<td>0.79 to 0.96</td>
<td>0.28</td>
</tr>
<tr>
<td>netty</td>
<td>1.97 to 2.24</td>
<td>0.68</td>
<td>1.58 to 1.74</td>
<td>0.54</td>
<td>1.32 to 1.44</td>
<td>0.45</td>
<td>1.12 to 1.22</td>
<td>0.38</td>
<td>0.98 to 1.07</td>
<td>0.33</td>
</tr>
<tr>
<td>openjpa</td>
<td>1.61 to 2.12</td>
<td>0.59</td>
<td>1.15 to 1.42</td>
<td>0.41</td>
<td>0.89 to 1.10</td>
<td>0.32</td>
<td>0.68 to 0.84</td>
<td>0.24</td>
<td>0.60 to 0.75</td>
<td>0.21</td>
</tr>
<tr>
<td>presto</td>
<td>1.12 to 1.73</td>
<td>0.47</td>
<td>0.95 to 1.30</td>
<td>0.37</td>
<td>0.88 to 1.14</td>
<td>0.33</td>
<td>0.76 to 0.96</td>
<td>0.28</td>
<td>0.72 to 0.90</td>
<td>0.27</td>
</tr>
<tr>
<td>qpid</td>
<td>1.35 to 1.75</td>
<td>0.51</td>
<td>1.19 to 1.40</td>
<td>0.42</td>
<td>0.98 to 1.13</td>
<td>0.35</td>
<td>0.65 to 0.77</td>
<td>0.23</td>
<td>0.58 to 0.68</td>
<td>0.21</td>
</tr>
<tr>
<td>wicket</td>
<td>1.51 to 1.88</td>
<td>0.56</td>
<td>1.44 to 1.64</td>
<td>0.51</td>
<td>1.18 to 1.33</td>
<td>0.41</td>
<td>0.95 to 1.08</td>
<td>0.33</td>
<td>0.92 to 1.03</td>
<td>0.32</td>
</tr>
<tr>
<td>overall</td>
<td>1.67 to 1.80</td>
<td>0.56</td>
<td>1.41 to 1.48</td>
<td>0.47</td>
<td>1.13 to 1.18</td>
<td>0.37</td>
<td>0.91 to 0.95</td>
<td>0.30</td>
<td>0.80 to 0.84</td>
<td>0.26</td>
</tr>
</tbody>
</table>
5.5. Evaluation

We manually examined entropy scores of sample lines and found strong associations with lexical and syntactic properties. In particular, lines with many and/or previously unseen identifiers, such as package, class and method declarations, had substantially higher entropy scores than average. Lines such as the first line of for-statements and catch clauses had much lower entropy scores, being often repetitive and making use of earlier declared variables. We use these variations in entropy scores by introducing the notion of line types, based on the code’s syntactic structure, i.e. the abstract syntax tree (AST), and computed a syntax-sensitive entropy score.

First, with each line, we associated syntax-type, corresponding to the grammatic entity that is the lowest AST node that includes the full line. These are typically AST node types such as statements (e.g., if, for, while), declarations (e.g., variable, structure, method) or nodes that typically span one line, such as switch cases and annotations. We then compute a normalized Z-score for the entropy of the line, over all lines with that node type.

\[ z_{\text{line, type}} = \frac{\text{entropy}_{\text{line}} - \mu_{\text{type}}}{SD_{\text{type}}} \]

The above normalization essentially uses the extent to which a line is “unnatural” with respect to other lines of the same type, to predict how likely it is to be buggy. In addition, we don’t expect all line types to be equally buggy; package declarations (import...) are probably usually correct, when compared to error handling (try...catch). The previously computed line-types come in handy here too: we can compute the relative bug-proneness of a type based on the fraction of bugs and total lines it had in all previous snapshots. Hence, we use the first snapshot as a ‘training set’ for this model and compute the bug-weight of a statement as:

\[ w_{\text{type}} = \frac{\text{bugs}_{\text{type}}/\text{lines}_{\text{type}}}{\sum_{\text{types}} \text{bugs}_t/\text{lines}_t} \]

where the bugs and lines of each type are counted over all previous snapshots. We then scale the z-score of each line by it’s weight \( w \) to achieve our final model, which we name $gram+wType$.

5.5 Evaluation

We begin with the question at the core of this paper:

**RQ1. Are buggy lines different from non-buggy lines?**

For each project, we compare line entropies of buggy and non-buggy lines. For a given snapshot, non-buggy lines consist of all the unchanged lines and the deleted lines that are not part of a bug-fix commit. The buggy lines include all the deleted lines in all bug-fix commits. Figure 5.4 shows the result, averaged over all the studied projects. Buggy lines are associated with higher entropies. Table 5.3 further details the average entropy difference between the buggy and non-buggy lines (buggy > non-buggy) and the effect sizes (Cohen’s
Note that both entropy difference and effect size decrease as we increase the threshold for the maximum number of deleted lines (max_delete) in a file commit. For example, for a max_delete size of 2, the entropies of buggy lines are on average 1.67 to 1.80 bits higher, with a high effect size of 0.56. However, when we consider all the studied commits (max_delete = 30), the entropies of buggy lines are, on average, less than a bit (0.80 to 0.84 bits) higher, with a small-to-moderate effect size of 0.26. One possible explanation: particularly in larger bug-fix commits, some of the deleted (or modified) lines might only be indirectly associated with the erroneous lines that most improbable (“unnatural”). These indirectly associated lines might actually be common, and thus have lower entropy; this would diminish overall entropy differences between the buggy and non-buggy lines. However, with statistical significance, we have the overall result:

\[ \text{non-} \text{buggy} \quad \text{buggy} \quad \text{fixed} \]

\[ \begin{align*} 
4 & \quad 8 & \quad 12 & \quad 16 
\end{align*} \]

Figure 5.4: Entropy difference between non-buggy, buggy, and fixed lines. File commits up to 5 deleted lines are considered, since five is the average number of deleted lines per file commit (see Figure 5.2).

Result 1: Buggy lines, on average, have higher entropies than non-buggy lines.

RQ2. Does entropy of a buggy line drop after the bug is fixed?

In a bug-fix commit, the lines deleted from the original versions are considered buggy lines and lines added in fixed versions are considered fixed lines. To answer RQ2, we collected all the buggy and the fixed lines across all the projects and compared their average entropies. It is hard to establish a one-to-one correspondence between a buggy and a fixed
5.5. Evaluation

Example 1: Wrong Initialization Value
Facebook-Android-SDK (2012-11-20)
File: Session.java
Entropy dropped after bugfix: **4.12028**

```java
if (newState.isClosed()) {
    // Before (entropy = 6.07042):
    - this.tokenInfo = null;
    // After (entropy = 1.95014):
    + this.tokenInfo = AccessToken.createEmptyToken(Collections.<String>emptyList());
}
...
```

Example 2: Wrong Method Call
Netty (2013-08-20)
File: ThreadPerChannelEventLoopGroup.java
Entropy dropped after bugfix: **4.6257**

```java
if (isTerminated()) {
    // Before (entropy = 5.96485):
    - terminationFuture.setSuccess(null);
    // After (entropy = 1.33915):
    + terminationFuture.trySuccess(null);
}
```

Example 3: Unhandled Exception
Lucene (2002-03-15)
File: FSDirectory.java
Entropy dropped after bugfix: **3.87426**

```java
if (!directory.exists())
    // Before (entropy = 9.213675):
    - directory.mkdir();
    // After (entropy = 5.33941):
    + if (!directory.mkdir())
        + throw new IOException("Cannot create directory: " + directory);
...
```

Table 5.4: Examples of bug fix commits that NBF detected successfully. These bugs evinced a large entropy drop after the fix. Bugs with only one defective line are shown for simplicity purpose. The errors are marked in red, and the fixes are highlighted in green.
5. Fault Detection

Example 4: Wrong Argument (\(\mathcal{NF}\) could not detect)
Netty (2010-08-26)
File: HttpMessageDecoder.java
Entropy increased after bugfix: 5.75103

```java
if (maxHeaderSize <= 0) {
    throw new IllegalArgumentException(
        // Before (entropy = 2.696275):
        // Before (entropy = 2.696275):
        - "maxHeaderSize must be a positive integer: "+ maxChunkSize);
    // After (entropy = 8.447305):
    + "maxHeaderSize must be a positive integer: "+ maxHeaderSize);
}
```

Example 5: (\(\mathcal{NF}\) detected incorrectly)
Facebook-Android-SDK (multiple snapshots)
File: Request.java

```
// Entropy = 9.892635
Logger logger = new Logger(LoggingBehaviors.REQUESTS, "Request");
...
```

Table 5.5: Examples of bug fix commits where \(\mathcal{NF}\) did not perform well. In Example 4, \(\mathcal{NF}\) could not detect the bug successfully (marked in red) and after bugfix the entropy has increased. In Example 5, \(\mathcal{NF}\) incorrectly detected the line as buggy due to its high entropy value.

line, because often buggy lines are fixed by a different number of new lines. Hence, we compare the mean entropies between buggy and non-buggy hunks. Figure 5.4 shows the result. On average, entropy of the buggy lines drop after the bug-fixes, with a drop of 1.19 to 1.26 bit, with 95% confidence.

Table 5.4 shows three examples of code where entropy of buggy lines dropped significantly after bug-fixes. In the first example, a bug was introduced in Facebook-Android-SDK code due to a wrong initialization value —tokenInfo was incorrectly reset to null (see the commit log). This specific initialization rarely occurred elsewhere, so the buggy line had a rather high entropy of 6.07. Once the bug was fixed, the fixed line followed a repetitive pattern (indeed, with two prior instances in the same file). Hence, entropy of the fixed line dropped to 1.95, an overall 4.12 bit reduction. The second example shows an example of incorrect method call in the Netty project. Instead of calling the method trySuccess (used three times earlier in the same file), the code incorrectly called the method setSuccess, which was never called in a similar context. After the fix entropy drops by 4.6257 bits. Finally, example 3 shows an instance of missing conditional check in Lucene. The developer should check whether directory creation is successful by checking return value of directory.mkdir() call, following the usual code pattern. The absence of this check raised the entropy of the buggy line to 9.21. The entropy value drops to 5.34 after the fix.

The table below shows the average drop of entropy and the Cohen’s D effect size of buggy vs. fixed lines, with varying thresholds for the maximum bug size in terms of deleted
5.5. Evaluation

Similar to the result of RQ1, both the entropy difference and effect size vary with maximum delete threshold: when the delete threshold increases, the other two decrease. For example, at maximum delete size 2, mean entropy drops from 1.52 to 1.62 bit (with 95% confidence) with statistical significance, with a large effect size (> 0.50). However, with delete threshold at 30, mean entropy difference between the buggy and non-buggy lines are only half a bit with a small effect size of 0.18. For all the studied ranges, the Wilcox non-parametric test confirms with statistical significance that the entropy of buggy lines is higher than the entropy of the fixed lines.

However, in certain cases these observations do not hold. For instance, in the example 4 of Table 5.5, entropy increased after the bug fix by 5.75 bits. In this case, developer copied maxChunkSize from a different context but forgot to update the variable name. This is a classic example of copy-paste error [80]. Since, the statement related to maxChunkSize was already present in the existing corpus, the line was not surprising. Hence, its entropy was low although it was a bug. When the new corrected statement with maxHeaderSize was introduced, it increased the entropy. Similarly, in Example 5 of Table 5.5 the statement related to logger was newly introduced in the corpus. Hence, its entropy was higher although it was not a bug.

Result 2: Entropy of the buggy lines drops after bug fixes, with statistical significance.

![Figure 5.5: Overall AUCEC upto inspecting 20% lines for all the projects](image-url)
5. Fault Detection

RQ3. Is “naturalness” a good way to direct inspection effort?

Having established that buggy lines are significantly less natural than non-buggy lines, we investigate whether entropy of a line can be used to direct inspection effort towards buggy code. In particular, we start by asking whether ordering lines by entropy will better guide inspection effort that ordering lines at random. For the reasons outlined in Section 5.3.3, we evaluate the performance of entropy-ordering, with the AUCEC scores at 5% of inspected lines (AUCEC_{5} in short). Furthermore, as outlined in section 5.3.3, we evaluate performance according to two types of credit: partial and full (in decreasing order of strictness). Finally, we disregarded all bugs that were part of a bug-fix which removed 15 or more lines, which we found this to be the 95th percentile of bug-fix sizes. As shown in Table 5.3, entropy plays a substantially smaller role in lines belonging to larger bug-fixes, hence we leave the identification of these lines to future research. We remind the reader that AUCEC_{5} is a non-parametric, cost-sensitive measure, and the comparison to random choice is done on an equal credit basis.

Figure 5.5 shows the AUCEC scores for partial credit, averaged over all projects, up to 20% of the inspected lines. Figure 5.6 offers a closer look at the performance on the 10 studied projects, up to 5% of the inspected lines. We see that, under partial credit, the default $gram model (without the syntax weighting described in §5.4.4) performs significantly better than random, particularly at more than 10% of inspected lines. However, at 5% of inspected lines its performance varies, consistently performing better than random but often just slightly. Indeed, average performance of $gram was significantly better than random at 20% (nearly twice as good) but only marginally so at 5% (17% better than random).

This picture changes substantially with the introduction of linetypes. Scaling the entropy scores by line type improves AUCEC_{5} performance in all but one case (Wicket) and significantly improves performance in all cases where $gram performed no better than random. Including the bugginess history of linetypes ($gram+wType) furthermore improves prediction performance in all but one system (Elasticsearch). The latter model consistently
5.5. Evaluation

outperforms random and $gram$ (except on Wicket), achieving AUCEC5 scores of more than twice that of random. These results were quite the same under full credit. Since $gram+wType$ is the best-performing “naturalness” approach, we hereafter refer to it as $NBF$.

**Result 3:** Entropy is a better way to choose lines for inspection than random

In previous work, Rahman *et al.* compared static bug FINDers with statistical defect prediction approaches [79]. To this end, they created a dataset consisting of 32 releases of 5 popular Apache projects and annotated the lines in each release with both bug information and $SBF$ information, as described in §5.4.1. Among others, they found that ordering $SBF$ warnings based on statistical defect prediction methods can improve the native $SBF$ ordering. This provides an interesting challenge for the $NBF$ algorithm: how does the $NBF$ algorithm compare to $SBF$, and can we improve the default ordering of the $SBF$ by using the techniques presented before? We investigate this in the next two research questions.

![Figure 5.7: AUCEC performance of $gram+wType$ vs. PMD and the combination model](image)

**RQ4. How do $SBF$ and $NBF$ compare in terms of ability to direct inspection effort?**

To compare $NBF$ with $SBF$, we computed entropy scores for each line in Rahman’s dataset using the $gram+wType$ model. Here we again use the threshold of 14 lines for bugs, which roughly corresponded to the fourth quartile of bug-fix sizes on this dataset. Indeed, we found that Rahman’s dataset had substantially more ‘large’ bugs compared to our earlier experiments, hence we also report results without imposing this threshold.

Rahman *et al.* developed a measure named AUCECL to compare $SBF$ and $DP$ methods on an equal footing. In this method, the $SBF$ under investigation sets the line budget based on the number of warnings it returns and the $DP$ method may choose a (roughly) equal number of lines. The models’ performance can then be compared by computing the AUCEC scores both approaches achieve on the same budget. We repeat to compare $SBF$ with $NBF$.
Furthermore, we also compare the AUCEC₅ scores of the algorithms. For the $gram+wType model this is analogous to the results in RQ3. To acquire AUCEC₅ scores for the $SBF$ model we simulate them as follows: First assign each line the value zero if it was not marked by the $SBF$ and the value of the $SBF$ priority otherwise ({1, 2} for FINDBUGS, {1 - 4} for PMD); then, add a small random amount (tie-breaker) from $U[0,1]$ to all line-values and order the lines by descending value. This last step simulates the developer randomly choosing to investigate the returned by $SBF$: first from those marked by the $SBF$ in descending (native, $SBF$ tool-based) priority, and within each priority level at random. We repeat the simulation multiple times and average the performance.

Figure 5.7 and 5.8 show the AUCEC₅ and AUCECL scores for PMD on the dataset by Rahman et al. [79] using partial credit. The results for FINDBUGS were comparable, as were the results using full credit. As can be seen, performance varied substantially between projects and between releases of the same project. Across all releases and under both AUCEC₅ and AUCECL scoring, all models performed significantly better than random (paired t-test: $p < 10^{-3}$), with large effect (Cohen’s D > 1). $SBF$ and $NBF$ performed comparably; $NBF$ performed slightly better when using both partial credit and the specified threshold for bug-sizes, but when dropping the threshold, and/or with full credit, no significant difference remains between $NBF$ and $SBF$. No significant difference in performance was found between FINDBUGS and PMD either.

In all comparisons, all approaches retrieved relatively bug-prone lines by performing substantially better than random.

Result 4: Entropy achieves comparable performance to commonly used $SBF$ in defect prediction.

Notably, $NBF$ had both the highest mean and standard deviation of the tested models, whereas PMD’s performance was most robust. This suggest a combination of the models: We can order the warnings of the $SBF$ using the $gram+wType$ model. In particular, we
found that the standard priority ordering of the $SBF$ is already powerful, so we propose to re-order the lines within each priority category.

**RQ5.** Is “naturalness” a useful way to focus the inspection effort on warnings produced by $SBF$?

Given the comparable performance of the $SBF$ and $NB$ models and the robustness of the $SBF$ algorithms, we may expect a combination of the models to yield superior performance. To this end, we again assigned values to each line based on the $SBF$ priority as in RQ4. However, rather than add random tie-breakers, we rank the lines within each priority bin by the (deterministic) $gram+wType$ score. The results for PMD are shown in Figure 5.8 first using the AUCEC$_5$ measure (5.7) and then using the AUCECL measure (5.8). PMD$_{Mix}$ refers to the combination model as proposed.

Overall, the combined model produced the highest mean performance in both categories. It significantly outperformed the two $SBF$s in all cases ($p < 0.01$) and performed similarly to the $NB$ model (significantly better on Lucene and QPid, significantly worse on Derby ($p < 0.05$), all with small effect). These results extended to the other evaluation methods, using full credit and/or removing the threshold for max bug-fix size. In all cases, the mix model was either significantly better or no worse than any of the other approaches when averaged over all the studied releases.

We further evaluated ranking all warnings produced by the $SBF$ by entropy (ignoring the $SBF$ priorities) and found comparable but slightly weaker results. These results suggest that both $NB$ and $SBF$ contribute valuable information to the ordering of bug-prone lines and that their combination yields superior results.

### Result 5

**Ordering $SBF$ warnings by priority and entropy significantly improves $SBF$ performance.**

## 5.6 Threats to Validity

A number of threats to the internal validity of the study arise from the experimental setup.

**Identifying Buggy Lines.** The identification of buggy lines is a possible source of error. We used the procedure as proposed by Mockus and Votta to identify bugfix commits [60], which may have lead to both false negatives and false positives in the identification of buggy lines.

Programmers may fail to indicate bug fixes in log messages, leading to false negatives (missing bugs). There is no reason to suspect that these missing buggy lines have a significantly different entropy-profile. Second, as yet unfixed bugs may linger in the code, constituting a right-censorship of our data; these bugs might have a different entropy-profile (although, again, no reason to suspect that this is so). Finally, it has been noted that developers may combine multiple unrelated changes in one commit [46, 29]. This work (op. cit.) observed that, when multiple changes were combined, bug-fix commits were mostly combined with refactorings and code formatting efforts. This may have lead to the deletion
of non-buggy high-entropy lines in a bugfix commit, although we expect these lines to form the minority of studied lines.

The threats identified above may certainly have lead to the misidentification of some lines; however, given the high significance of the difference in entropy between buggy and non-buggy lines, we consider it unlikely that these threats could invalidate our overall results. Furthermore, the performance in defect prediction of a model using entropies on the (higher quality, JIRA-based) bug dataset by Rahman et al. confirms our expectations regarding the validity of these results. A final threat regarding RQ2 is the identification of ‘fixed’ lines, lines that were added in the place of ‘buggy’ lines during a bugfix commit. It is possible that the comparisons between these categories is skewed, e.g., because bugfix commits typically replace buggy lines with a larger number of fixed lines. We found no evidence of such a phenomenon but acknowledge the threat nonetheless. Future research may apply entropy to defect correction and further study this relation of buggy lines to fixed lines in terms of entropy.

In RQ4 and RQ5 we investigated the performance of the proposed NBF in comparison to (and in combination with) static bug finders. We note that here too the identification of buggy lines may be a cause for systematic error, for which we point both to the above discussion and to Section 5 of Rahman et al., in which they identify a number of threats to the validity of their study [79].

Our comparison of SBF and NBF assumes that indicated lines are equally informative to the inspector, which is not entirely fair; NBF just marks a line as “surprising”, whereas SBF provides specific warnings. On the other hand, we award credit to SBF whether or not the bug has anything to do with the warning on the same lines; indeed, earlier work [95] suggests that warnings are not often related to the buggy lines which they overlap. So this may not be a major threat to our RQ4 results.

Finally, the use of AUCEC to evaluate defect prediction has been criticized for ignoring the cost of false negatives [102]; the development of better, widely-accepted measures remains a topic of future research.

**Generalizability.** The selection of systems constitutes a potential threat to the external validity of this research. We attempted to minimize this threat by using systems from both Github and Apache, having a substantial variation in age, size and ratio of bugs to overall lines (see table 5.1).

Finally, does this approach generalize to other languages? There’s nothing language-specific about the implementation of n-gram and $gram models (the $gram+wType model, however, does require parsing, which depends on language grammar). Our own prior research [47, 99] showed that these models work well to capture regularities languages such as Java, C, and Python, yielding low cross-entropies when well trained on a corpus of code. The question remains then whether language models can identify buggy code in other languages as well, and whether the entropy drops upon repair. As a sanity check, using the same methods described in Section 5.4, we gathered data from 3 C/C++ projects (Libuv, Bitcoin and Libgit). These projects together constituted just over 10M LOC. We gathered snapshots spanning the period of November 2008 - January 2014. The data comprised a total of 8298 commits, including 2518 bug-fix commits (identified as described in Sec-
5.7. Related Work

In the following we analyze work related to our investigation.

5.7.1 Statistical Defect Prediction

Software development is an incremental process. This incremental progress is successfully logged in version control systems like git, svn and issue databases. Learning from such historical data of reported (and fixed) bugs, Statistical Defect Prediction (DP) aims to predict location of the defects that are yet to be detected. This is a very active area (see [21] for a survey of the area), even having the dedicated PROMISE series of conferences (See [1] for recent proceedings). The state of the art DP not only leverages bug history, it also takes into account several other product (file size, code complexity, code churn etc.) and process metrics [78] (developer count, code ownership, developer experience, change frequency etc.) to improve the prediction model. Thus, using different supervised learning techniques like logistic regression, Support Vector Machine (SVM) etc., DP associates various software entities (e.g., methods, files and packages) with their respective defect proneness.

Given a fixed budget of SLOC that needs to be inspected to effectively find most bugs, DP ranks files that one should inspect to detect most of the errors. DP doesn’t necessarily have to work at the level of files; one could certainly use prediction models at the level of modules, or even at the level of methods. To our knowledge, no one has done purely statistical models to predict defects at a line-level, and this constitutes a novel aspect of our work. While earlier work evaluated models using IR measures such as precision, recall and F-score, more recently non-parametric methods such as AUC and AUCEC have gained in popularity.

5.7.2 Static Bug Finders

The core idea of static bug finding is to develop an algorithm that automatically finds likely locations of known categories of defects in code. Some use heuristic pattern-matching; others are sophisticated algorithms that compute well-defined semantic properties over abstractions of programs carefully designed to accomplished specific speed-vs-accuracy tradeoffs in detecting certain categories of bugs. The former tools include FindBugs and PMD, which
we studied; the latter includes tools like ESC-Java [34]. The former category can have both false positives and negatives. The over-riding imperative in the latter approach is to never falsely certify a program (that actually has e.g., memory leak bugs) to be bug-free; typically however, false positives can be expected.

The field has advanced rapidly, with many developments; researchers identify new categories of defects, and seek to invent clever methods to find these defects efficiently, either heuristically or though well-defined algorithms and abstractions. Since neither method is perfect, the actual effectiveness in practice is an empirical question. Since our goal here is just to compare SBF and NBF, we refer the reader for a more complete discussion of related work regarding SBF and their evaluation to Rahman et al. [79].

5.7.3 Grammatical Error Correction in NLP

Grammatical error correction is an important problem in natural language processing (NLP), which is to identify grammatical errors and provide possible corrections for them. The pioneering work on grammatical error correction was done by Knight and Chander [53] on article errors. Along the same direction, researchers have proposed different classifiers with better features for correcting article and preposition errors [45, 94, 39, 27]. However, the classifier approaches mainly focus on identifying and correcting specific types of errors (e.g. preposition misuse). To approach this problem, some researchers have begun to apply the statistical machine translation approach to error correction. For example, Park and Levy [73] model various types of human errors using a noisy channel model, while Dahlmeier and Ng [28] describe a discriminative decoder to allow the use of discriminative expert classifiers.

There is one fundamental difference between grammatical error correction in natural languages and defect localization in programming languages. Natural languages are close-vocabulary (i.e. have limited number of vocabulary), thus lead to limited types of grammatical errors (e.g. articles, prepositions, noun number) with enumerable corrections (e.g. possible article choices are a/an, the, and the empty article ε). In contrast, programming languages are open-vocabulary (e.g. programmers could arbitrarily construct new identifiers). Therefore, the defects in programming languages are more flexible and thus harder to localize. Based on the observation that software corpora are highly repetitive [47] and localized [99], we exploit a cache language model [99] to locate the defects that are not natural in the sense that the sequences of code are not observed frequently either in the training code repository or in the local file.

5.8 Conclusion

The repetitive, predictable nature (“naturalness”) of code suggests that code that is improbable (“unnatural”) might be wrong. We investigate this intuition by using entropy, as measured by statistical language models, as a way of measuring unnaturalness.

We find that unnatural code is more likely to be implicated in a bug-fix commit. We also find that buggy code tends to become more natural when repaired. We then turned to applying entropy scores to defect prediction and find that, when adjusted for syntactic
variances as well as syntactic variance in defect occurrence, it is about as cost-effective as the commonly used static bug-finders PMD and FindBugs.

Finally, applying the (deterministic) ordering of entropy scores to the warnings produced by these static bug-finders produces the most cost-effective method. These findings suggest that entropy scores are a useful adjunct to defect prediction methods. The findings also suggest that certain kinds of automated search-based bug-repair methods might do well to have the search in some way influenced by language models.
Chapter 6

Conclusion

Over the course of this thesis, we have investigated the so-called “naturalness” property of software: that source code can (to a large extent) be modeled statistically as natural language. Investigations using this perspective have led to substantial insights into the structure of software and its development, and have led to improvements in a variety of Software Engineering applications. In this thesis, we have expanded upon this body of knowledge by investigating three topics of interest: collaborative software engineering, programming tool support and fault detection. I will first list the main contributions of the findings in this thesis, then I will discuss the ramifications of our findings in relation to our central thesis.

6.1 Contributions

Our main contributions are as follows. Firstly, we investigated the influence of the predictability of submitted code on the reaction thereto by those in charge of code review. We found that more predictable code submissions in open source development were also reviewed more concisely and were less often rejected. Less predictable code, on the other hand, triggered more comments during review and was more often (eventually) rejected. Code that was both reviewed diligently and of high entropy was most likely to be rejected (up to 50% rejection rates). These results suggest that understandability and “surprisingness” of source code plays an important part in the perceived quality and desirability of submissions, at least in an open source setting. Vice versa, code that resembles code that is already in the project appears to be of reduced interest to reviewers.

Secondly, we studied the application of (mined) highly predictable code snippets to a code completion task. We demonstrated that a code recommender system based on statistically likely code achieves substantially higher accuracy than a commonly used rule-based code recommendation system. In particular, we found substantially improved performance resulting from a combination of the two approaches, suggesting that both approaches identify and employ different useful features.

Finally, we investigated highly unpredictable code as a candidate for faulty code. We found that lines with high entropy were both significantly more likely to be buggy and reduced substantially in entropy after fixing. Ordering lines by entropy in a code inspection setting yielded performance on par with commonly used static bug finders.
Overall, our results demonstrate that the naturalness phenomenon reflects an underlying factor of importance in software engineering and development. Whereas natural code is perceived as less interesting, acceptable, and can be employed to reduce the cost of programming, unnatural code is more fault-prone and is treated with more skepticism in collaborative software development.

6.2 Discussion

Our central thesis was: statistical predictability, or “naturalness”, of software impacts software engineering and development. We investigate three perspectives on this question in the previous chapters, and concluded that “the naturalness phenomenon reflects an underlying factor of importance in software engineering and development”. The ramifications and philosophical nature of these findings are worthy of some discussion, after which we will discuss the potential directions for future research that these findings warrant in Section 6.3.

6.2.1 How natural is software?

Firstly, the conclusions following from the main chapters of this thesis support a correlation between statistical regularity of code and the perception thereof by software developers. This may appear surprising to many software developers: the “traditional” perspective on source code perceives it as a rigid, emotionless and uniform entity. In popular culture, these expectations are fueled by representations of code as sequences of binary instructions or endless series of machine code instructions.

When comparing software to natural language, however, the presented results are much more in line with expectations: people have distinctive writing styles and pursue uniform quality of writing when collaborating (cf. Chapter 3); auto-complete functionality in our smartphones learns from general language structure and from our personal writing style, and has become all but unmissable (cf. Chapter 4); complex sentences and complicated words are a staple of language education and spelling tests due to our proneness to make mistakes in them. (cf. Chapter 5).

Similarly, we found that open source projects value an (often implicit) uniform style of coding, studying deviant contributions with increased scrutiny (Chapter 3). We saw that a relatively simple statistical code completion model performed rather well, simply by learning from common and locally typical usage (Chapter 4). Finally, we found that mistakes are detected exactly in examples of more complex and irregular source code.

Do the above suggest that software really is just like natural language? Recent research adopting this “naturalness” assumption – that source code may be modeled like natural language – has consistently found two things: 1. that such a representation provides an powerful tool in software engineering research, particularly in high-level languages (e.g., Java, C#), and 2. that software nonetheless behaves differently from natural language and that best results are obtained by adapting the modeling techniques towards the peculiarities of source code.

My conjecture is that software (at least the modern, high-level software languages) behaves naturally to a large extent. Adopting such a perspective may help us better understand
how humans perceive software and how its development can be simplified. However, software languages should not be viewed as fully natural, nor completely modeled after it (see also [30]). In fact, we may benefit from a balanced perspective, for instance by incorporating the rich amounts of statically available compiler information into our models.

6.2.2 Do developers treat software as natural language?

Secondly, we have observed that software naturalness appears to influence developer behavior. For instance, coding conventions are a popular topic of discussion in open-source projects and we often found that a consensus on a convention was reached during the discussion of a contribution (Chapter 3). Furthermore, developers appear to have more trouble writing complex software, inducing more faults in the process and reducing the complexity when fixing detected errors (Chapter 5). Finally, the introduction of increasingly high-level software languages (w.r.t. machine code) in recent decades suggests that (many) developers prefer a more natural language-esque way to write code.

On this topic, I cannot offer a definitive answer based on the findings in this thesis. Our findings suggest that repetition and predictability of source code are a factor on which developers have come to rely to write and collaborate on code. Whether this is inherent to human nature, or a property only of presently popular software languages is not clear; further study is needed to investigate this phenomenon. A better understanding of the degree to which code is (and can be) natural can help us create software languages that better suit our mental expectations of programming and make software development accessible to a much larger part of the population.

6.3 Future Work

The results presented in this thesis warrant further study of language modeling in software engineering. In the previous section, we have described some general points for further investigation; here we focus on the three topics in this thesis.

6.3.1 Further study of code review

We found that the notion of code “fitting in” in a project is an implicit factor of interest to the project maintainers. Hindle et al. found that language models capture project-specific code style [47] and our results suggest that project maintainers are aware of this code style, enforcing it through more critical review of surprising code. This suggests the presence of a “mental model” of what the code in a project ought to look like. This warrants further research.

Related work has found that roles within a project tend to be divided within the code-base, with some developers having substantially higher degrees of ownership of certain modules. Furthermore, recent work finds that developers have a specific code style that can be captured by language models [90]. It would be worthwhile to investigate if developers are particularly critical to code that violates “their” conventions in “their” modules. Thus, a more developer-centric investigation of code review is an interesting candidate for further
6. CONCLUSION

study. Additionally, such a study could help create code-style recommendation systems for novel contributors in order to increase the acceptability of their code.

Secondly, an overlap with fault detection presents itself: do less regular changes actually induce more faults? Or are they received with more criticism due to social factors, such as maintaining community standards?

Finally, if programmers maintain a mental model of the source code in their project, it ought to be possible to capture this model in terms of statistics as well. This presents a more general agenda in which a computer may aide both project maintainers and novel contributors in advancing the agenda of their project in terms of (among others) coding conventions, performance and scope.

6.3.2 Code completion

Code completion has attracted much interest since its introduction as an application of language models by Hindle et al. [47]. In particular, whereas previous research has often focused on API completion, these statistical methods can provide any kind of completions in any context. It remains to be studied how developers would use such “ubiquitous” completion engines and what types or scope of completion deserve special attention. Additionally, completion approaches may be adapted to suggestion mechanisms for many goals, such as method names (see also [4]), code style conformance (see Section 6.3.1, [3]) or refactorings.

6.3.3 Fault detection

Finally, the topic of automated fault detection has been explored extensively in software engineering research. In this thesis, statistical properties of source code have been shown to be a useful factor in this task. However, it remains the topic of further work to put these results in context: how does this relate to other source code metrics or process metrics? Is unnatural code fault-prone by definition or only when it stands out in context (e.g., Allamanis et al. found that some classes have much higher entropies in general [2])?

Of more general interest is the question if machines can suggest better implementations as well. Recent research has used Genetic Algorithms to sample rewritings that will pass a set of test case [42]. However, there is evidence that software fixes are often predictable as well, relying on pre-existing code [57]. Thus, a broader agenda emerges in which statistical models work side-by-side with software developers to improve software quality.


