Analyzing the Predictability of Source Code and its Application in Creating Parallel Corpora for English-to-Code Statistical Machine Translation

Musfiqur Rahman

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

March 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Musfiqur Rahman**

Entitled: **Analyzing the Predictability of Source Code and its Application in Creating Parallel Corpora for English-to-Code Statistical Machine Translation**

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

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Sabine Bergler

_____ Examiner
Dr. Weiyi Shang

_____ Supervisor
Dr. Peter Rigby

Approved by _____
Dr. Volker Haarslev, Graduate Program Director

23 March 2018 _____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Analyzing the Predictability of Source Code and its Application in Creating Parallel Corpora for English-to-Code Statistical Machine Translation

Musfiqur Rahman

Analyzing source code using computational linguistics and exploiting the linguistic properties of source code have recently become popular topics in the domain of software engineering. In the first part of the thesis, we study the predictability of source code and determine how well source code can be represented using language models developed for natural language processing. In the second part, we study how well English discussions of source code can be aligned with code elements to create parallel corpora for English-to-code statistical machine translation. This work is organized as a "manuscript" thesis whereby each core chapter constitutes a submitted paper.

The first part replicates recent works that have concluded that software is more repetitive and predictable, *i.e.* more natural, than English texts. We find that much of the apparent "naturalness" is artificial and is the result of language specific tokens. For example, the syntax of a language, especially the separators *e.g.,* semi-colons and brackets, make up for 59% of all uses of Java tokens in our corpus. Furthermore, 40% of all 2-grams end in a separator, implying that a model for autocompleting the next token, would have a trivial separator as top suggestion 40% of the time. By using the standard NLP practice of eliminating punctuation (*e.g.,* separators) and stopwords (*e.g.,* keywords) we find that code is less repetitive and predictable than was suggested by previous work. We replicate this result across 7 programming languages.

Continuing this work, we find that unlike the code written for a particular project, API code usage is similar across projects. For example a file is opened and closed in the same manner irrespective of domain. When we restrict our n-grams to those contained in the Java API we find that the entropy for 2-grams is significantly lower than the English corpus. This repetition perhaps explains the successful literature on API usage suggestion and autocompletion.

We then study the impact of the representation of code on repetition. The n-gram model assumes that the current token can be predicted by the sequence of n previous tokens. When we extract program graphs of size 2, 3, and 4 nodes we see that the abstract graph representation is much more concise and repetitive than the n-gram representations of the same code. This suggests that future work should focus on graphs that include control and data flow dependencies and not linear sequences of tokens.

The second part of this thesis focuses cleaning English and code corpora to aid in machine translation. Generating source code API sequences from an English query using Machine Translation (MT) has gained much interest in recent years. For any kind of MT, the model needs to be trained on a parallel corpus. We clean StackOverflow, one of the most popular online discussion forums for programmers, to generate a parallel English-Code corpora. We contrast three data cleaning approaches: standard NLP, title only, and software task. We evaluate the quality of each corpus for MT. We measure the corpus size, percentage of unique tokens, and per-word maximum likelihood alignment entropy. While many works have shown that code is repetitive and predictable, we find that English discussions of code are also repetitive. Creating a maximum likelihood MT model, we find that English words map to a small number of specific code elements which partially explains the success of using StackOverflow for search and other tasks in the software engineering literature and paves the way for MT. Our scripts and corpora are publicly available.

Acknowledgments

I would like to begin with acknowledging the guidance, encouragement, and support from my supervisor, Dr. Peter Rigby. Without his proper supervision this work would not have came into existence. I wholeheartedly thank him for his patience, effort and time towards me and my thesis.

I am also thankful to my fellow colleagues and paper collaborators, especially Dharani Kumar Palani, Nafiz-al-Naharul Islam, and Nicolas Chausseau-Gaboriault.

I am thankful to the Dr. Sabine Bergler for her valuable suggestions that helped me generate new ideas for the research. Additionally, I would like to thank Dr. Tien Nguyen (University of Texas at Dallas) and Dr. Christoph Treude (University of Adelaide) for their time and guidance.

Last but certainly not least, I thank my parents who encouraged and supported me throughout the time of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Leveraging methods and algorithms fundamentally developed for Natural Language Processing [29] in modelling and analyzing programming languages is an interesting topic of research in the domain of software engineering. In this thesis we first investigate the regularity of multiple programming languages using n-gram language models and then use various NLP techniques to process software engineering documentation to create a bilingual English-code parallel corpus.

Language models are a very popular approach in the field of *Statistical Machine Translation (SMT)* [32] and *Natural Language Processing (NLP)* [29]. The growing popularity of this approach has resulted in the application of language modelling techniques in diverse fields. In the field of Software Engineering recent works have exploited the benefits of language modelling to study the 'naturalness' of software source code [26, 50, 13, 57]. Although the term *naturalness* apparently does not refer to any mathematical notion, it has been presented mathematically by using the theory of statistical language modelling [26]. In essence, language models, being trained on a large corpus, assign higher naturalness to previously seen code, while assigning lower naturalness to unseen or rarely seen code. For example, Campbell *et al.* [13] showed that language models mark code which is syntactically faulty as *unlikely* or *less likely*. The goal of Chapter 3 is to explain the repetitive behaviour of source code for multiple programming languages and to compare the repetitiveness of n-grams with graph representations of code. We perform our experiments from the point of view of the token distribution to determine if naturalness can be found in popular programming languages. We want to understand if there are any language specific features that make one language more repetitive than the others. We compare the n-gram representation (*i.e.* sequences of n tokens) used by previous works [71, 26, 13] with a graph based representation. We conjecture that low-level lexical tokens may artificially inflate the repetitiveness of source code. For example, in most of the popular programming languages the `if` token is always followed by a `(` token. This trivial repetitiveness is

not present in graphs. We particularly focus on the following:

- the impact of different types of tokens on the repetitiveness of source code. We examine keywords, operators, and separators to observe their repetitiveness in source code,
- changes in token repetitiveness after the removal of language specific tokens,
- repetitiveness of API element usage, and
- graph-based representation of source code and its impact on source code repetitiveness.

Chapter 3 is broken into the following sections. In Section 3.3, we describe our data. We replicate previous results in Section 3.4. In Section 3.5, we study the impact of types of tokens on repetitiveness. In Section 3.6, we examine the repetitiveness of API code elements. In Section 3.7, we compare graphs with n-gram. Since we extract different tokens and graphs, we describe the extraction methodology in this section where they are used. In Section 3.8, we discuss limitations of our work and threats to validity. In Section 3.9, we position our work in the context of the literature. In Section 3.10, we summarize our contribution and conclude the chapter.

In Chapter 4, we leverage our finding regarding the 'naturalness' of source code. Since source code APIs are much more repetitive than natural languages we try to process source code and software engineering discussion in order to automatically translate from English to source code.

The process of translating between two languages automatically is known as Machine Translation (MT). Recent advances and computational power have increased the popularity of MT. MT techniques can be broadly classified into three classes: Statistical Machine Translation (SMT) [33, 2, 38], Example-based Machine Translation (EBMT) [69], and Neural Machine Translation (NMT) [6, 46]. Although MT approaches differ in terms of theory, algorithms, and efficiently, all approaches require a high volume and low noise *parallel corpus* [31, 7] or *bitext* [25]. Application of MT algorithms is not limited to translation between natural languages. In recent years, MT techniques have been used to translate from natural language to programming languages [24, 49, 56].

StackOverflow can be seen as a bilingual corpus that discusses programming in both English and code. However, StackOverflow posts are noisy because people write posts in an informal manner. Examples of noise in StackOverflow includes incorrect spelling, inappropriate use of punctuation, use of acronyms without elaboration, and grammatical mistakes. From a linguistic point of view, this results in a degradation of the quality of corpus texts. Removing the noise from the StackOverflow data without any significant loss of relevant information is challenging. In this chapter our goal is to clean the data using techniques ranging from general Natural Language Processing (NLP) [29] to Software Engineering specific techniques [70], and determine which techniques yield a corpus that can be used for MT. We process data using three different methods and determine the quality of the processed corpora using three evaluation metrics.

Chapter 4 is structured as follows. In Section 4.3, we detail our data and data cleaning approaches. In Section 4.4, we evaluate each corpus for MT. Finally we conclude the chapter by summarizing our contribution and briefly discussing some potential future works in Section 4.5.

This thesis is organized as a "manuscript" thesis whereby each core chapter constitutes a submitted paper. There is also a background section and conclusion section that combine the manuscripts into a thesis. Chapter 2 briefly introduces the literature and background necessary for the thesis. Chapters 3 describes our paper on the "natural" properties of software. Chapter 4 is our paper on cleaning StackOverflow for machine translation. Chapter 5 summarizes our contributions and suggest potential future directions.

Chapter 2

Background and Literature

We break the related work into the following categories:

- Application of NLP in software engineering
- Research on code validation
- Research into autocompletion and recommenders
- Statistical translation
- Use of StackOverflow in software engineering research

2.1 Application of NLP in software engineering

Basic research into understanding redundancy and measuring entropy in languages has a long history. Shannon [68] developed statistical measures of entropy for the English language. Gabel and Su [22] noted high levels of redundancy in code and Hindle *et al.* [26] continued this work,s demonstrating that software is highly repetitive and predictable. Recent work has replicated these software findings on a giga-token corpus [3] and looked at the entropy in local code contexts [71]. Other have examined repetition at the line level [59] or in other domains such as Android Apps [36, 5]. In each case, code has been found to be repetitive and predictable.

Besides language entropy, many software engineering researchers used other NLP approaches such as text classification, latent semantic analysis etc in their works. For example, Huang *et al.* and Maldonado *et al.* in [27, 42] respectively took the text classification approach for identifying self admitted technical debt. Lormans *et al.* used latent semantic indexing for designing implementation by linking requirements and test cases [39]. Latent Dirichlet Allocation (LDA) technique was used

by Wang *et al.* [72] to study developers' interaction on StackOverflow. In [65] authors studied what mobile App developers ask about on StackOverflow. They also use LDA technique for topic identification from the discussion forum.

2.2 Research on code validation and checking

Most existing tools to find defects and other code faults use static analysis. Recent works have focused on using the statistical properties of the languages to find bugs and to suggest patches. For example, Campbell *et al.* [13] find that syntax errors can be identified using n-gram language models. Ray *et al.* [58] identified bugs and bug fixes in code because buggy code is less natural and has a higher entropy. Santos and Hindle [67] used the n-gram cross entropy of text in commit messages to identify successfully commits that were likely to make a build fail.

2.3 Research into autocompletion and suggestions

Modern IDEs contain an autocompletion feature that usually uses the structure of the language to make suggestions. Research into code suggestion have long known intuitively that code is repetitive. For example, textual similarity of program code [4], commit messages [14], and API usage patterns [44] have been exploited to guide developers during their engineering activities. Building on this work, Zimmermann *et al.* [76] used association rule mining on CVS data to recommend source code that is potentially relevant to a given change task. Recent work by Azad *et al.* [5] has extended this work to make change rule predictions from a large community of similar Apps and the code discussed in StackOverflow discussions.

Advanced autocompletion techniques have leveraged the history of applications and the repetitive nature of programming to suggest code elements to developers. Robbes and Lanza [62] filtered the suggestions made by code completion algorithms based on, for example, where the developer had been working in the past and the changes he or she had made. Bruch *et al.* [9] suggested appropriate method calls for a variable based on an existing code base that makes similar calls to a library. Buse and Weimer [10] automatically generate code snippets from a large corpus of applications that use an API. Duala-Ekoko and Robillard [21] use structural relationships between API elements, such as the method responsible for creating a class, to suggest related elements to developers. Works by Nguyen *et al.* [50] use statistical language models to autocomplete code accurately. Nguyen and Nguyen [47] expanded this work to graphs in order to create suggestions that are syntactically valid.

2.4 Statistical translation

Recent works have mirrored the success of Statistical Machine Translation in natural languages, *e.g.,* Google Translate, and applied these approaches to translating English to code. For example, SWIM [56] uses a corpus of queries from Bing to align code and English and generates sequences of API usages. DeepAPI [24] uses recurrent neural networks to translate aligned source code comments with code to translate longer sequences of API calls. T2API [49] uses alignments between English and code on StackOverflow to generate a set of API calls. These calls are then rearranged based on the likelihood of existing program graphs. T2API can generate long graphs of common API usages from English. Our work provides a frame in which to understand these works. For example, the sequences of SWIM and DeepAPI tend to be short and simplistic as they are restricted by a left-to-right processing of tokens. In contrast, T2API which re-orders API elements in a graph can produce more complex usages.

2.5 Use of StackOverflow in software engineering research

Many researchers in their works used StackOverflow posts for performing experiments on issues related to empirical software engineering, program comprehension, code completion, etc. This data source is very popular among the software engineering research community due to its availability as well as volume. Wong *et al.* mined StackOverflow data to autogenerate source code comments [73]. Pinto *et al.* studied software energy consumption from StackOverflow discussion in [53]. Wong *et al.* in [73] studied how developers interact in StackOverflow discussion. In a similar work, Chowdhury *et al.* worked on filtering out off-topic from online discussion forums by mining StackOverflow [15]. Rigby *et al.* in [61] developed a tool for extracting salient code elements from StackOverflow posts, which we use in this thesis.

Chapter 3

Natural Software Revisited

Note: This chapter has been submitted to a conference and has been included verbatim in this manuscript thesis.

3.1 Abstract

Recent works have concluded that software is more repetitive and predictable, *i.e.* more natural, than English texts. These works included "simple/artificial" syntax rules in their language models. We find that while syntax is important, it is trivially predictable. For example, in "while (...)", bracket always follows "while", the compiler has a rule for this. When we remove these and other SimpleSyntaxTokens we find that code is still repetitive and predictable but only at levels slightly above English. Furthermore, previous works have compared individual Java programs to general English corpora, such as Gutenberg. Gutenberg contains a historically large range of styles and subjects (*e.g.,* Saint Augustine to Oscar Wilde). We perform an additional comparison of StackOverflow English discussions with source code and find that this restricted English is almost as repetitive as code. Our results hold across seven programming languages.

Although we find that code is less repetitive than previously thought, we suspect that API code element usage will be repetitive across software projects. For example, a file is opened and closed in the same manner across domains. When we restrict our n-grams to those contained in the Java API we find that the entropy for 2-grams is significantly lower than the English corpus. This repetition partially explains the successful literature on API usage recommendation and autocompletion.

Previous works have focused on sequential sequences of tokens. While n-grams work well for sequential natural languages, we suspect that they obscure abstract patterns in code. When we extract program graphs of size 2, 3, and 4 nodes we see that the abstract graph representation is much more concise and repetitive than the n-gram representations of the same code. This suggests

that future work should focus on graphs that include control and data flow dependencies and search for new representations that go beyond linear sequences of tokens. Our replication package makes our scripts and data available to future researchers [1].

3.2 Introduction

Language modelling is a popular approach in the field of *Statistical Machine Translation (SMT)* [32] and *Natural Language Processing (NLP)* [29]. The growing popularity of this approach has resulted in the application of language modelling techniques in diverse fields. In the field of Software Engineering, language modelling has revealed power-law distributions and an apparent 'naturalness' of software source code [26, 50, 13, 57]. Although the term *naturalness* is vague, it has been expressed mathematically with statistical language models [26]. In essence, language models trained on a large corpus, assign higher naturalness to previously seen code, while assigning lower naturalness to unseen or rarely seen code. For example, Campbell *et al.* [13] showed that language models mark code which is syntactically faulty as *unlikely* or *less likely* than code without syntax errors. The goal of this paper is to revisit the "natural" code hypothesis in new contexts. As in NLP, the SE tasks and context will require different tuning and cleaning of a corpus. For example, if the goal is to create an English grammar correction tool, then stopwords such as 'the' are necessary. In contrast, if the goal is to extract news topics then stopwords must be removed as these dominant tokens will introduce noise and reduce the quality of predictions. Analogously, if the goal is to find syntax errors then the corpus must include SimpleSyntaxTokens. In contrast, if the goal is to recommend multi-element API usages, then SimpleSyntaxTokens will dilute predictions. For example, Hindle *et al.* [26] did not remove SimpleSyntaxTokens and in their autocompletion model they suggest a SimpleSyntaxToken approximately 50% of the time. As a result, a recommender tool would suggest an obvious separator before a useful token such as an API call. In this work, we examine the repetitive behaviour of source code for multiple programming languages, we determine the impact of SimpleSyntaxTokens on repetition, we quantify how repetitive API usages are, and we compare the repetitiveness of n-grams vs graph representations of code. We examine each topic in the following four research questions.

RQ1, Replication: how repetitive and predictable is source code?

We replicate the work of Hindle *et al.* [26]. We also examine 6 additional programming languages: C#, C, JavaScript, Python, Ruby, and Scala. Our replication gives us confidence that our dataset is large and diverse enough to test the "naturalness" hypothesis in new contexts.

RQ2, Artificial Repetition: how repetitive and predictable is code once we remove SimpleSyntaxTokens?

In NLP, it is standard practice to remove punctuation and stopwords. We examine the contribution of three types of SimpleSyntaxTokens to the language distribution: separators such as bracket and semi-colon; keywords, such as `if` and `else`; and operators, such as plus and minus signs.

RQ3, API Usages: how repetitive and predictable are Java API usages?

Frameworks and APIs provide reusable functionality to developers. Unlike the code written for a particular project, API code is similar across projects. For example, a file is opened and closed in the same manner whether it is used in banking or healthcare. We examine only Java API tokens and determine how repetitive and predictable their usage is. Given the large and successful literature on API usage recommendations and autocompletions, we suspect that API elements may be more repetitive and predictable than general program code.

RQ4, Code Graphs: how repetitive and predictable are graph representations of Java code?

An n-gram language model assumes that the current token can be predicted by the sequence of $n-1$ previous tokens. However, compilers and humans do not process programs sequentially. In the case of compilers, parse trees or syntax trees are generated to provide abstract representations of code. Eyetracking studies of developers reading code show a nonlinear movement along the control and data flow of the program [11]. We extract the *Graph-based Object Usage Model (Groum)*[51] from Java programs and compare how repetitive graphs of nodes sizes 2, 3, and 4 are with equivalent sized n-grams from the same Java programs.

The remainder of this paper is structured as follows. In Section 3.3, we describe our data. In Sections 3.4, 3.5, 3.6, and 3.7, we report the results of our experiments for each of the research questions. Since we extract different tokens and graphs, we describe the extraction methodology in section in which it is used. In Section 3.8, we discuss limitations of our work and threats to validity. In Section 3.9, we position our work in the context of the literature. In Section 3.10, we summarize our contribution and conclude the paper. We also publicly release a replication package [1] which includes all processed n-gram and graph data as well as the scripts used in our processing pipeline.

3.3 Data Sources

Project Source Code: We create our source code corpus from 134 open source projects on GitHub. As a starting point, we select the Java and Python project used in a prior study [71]. To ensure that we processed a consistent number of tokens for each language, between 20M and 25M tokens, we added Java and Python projects as well as projects from 5 additional programming languages. These projects were selected from the most popular projects on GitHub for each language.[1] For all the

[1]Top GitHub projects per language: `https://github.com/trending/` accessed Nov 2016

Table 1: Corpus size in tokens per language

Language	Files	Total Tokens	Unique Tokens
Java	26,938	24,091,076	388,399 (1.61%)
C#	23,186	24,217,086	389,800 (1.61%)
C	10,932	25,255,417	938,434 (3.72%)
JavaScript	10,544	25,157,297	257,606 (1.02%)
Python	15,454	23,198,691	513,728 (2.21%)
Ruby	60,371	25,896,601	715,157 (2.76%)
Scala	34,242	23,634,250	333,794 (1.41%)

projects, we examine only the master branch. Since each research question requires the source code to be processed differently, *e.g.,* n-grams vs graphs, we describe the extraction methodology for each research question. The list of projects, scripts, and the processed n-grams and graphs can be found in our replication package [1]. A summary for each programming language is shown in Table 1.

English and StackOverflow text: Following Hindle *et al.* [26] we process the Gutenberg corpus. We use a subset of the Gutenberg corpus which includes over 3.4k English works [35]. The corpus represents a range of styles, topics, and timeperiods making Gutenberg a diverse corpus. In contrast, the programming corpora are for single programming languages. To make a more comparable English corpus, we process StackOverflow posts that discuss programming tasks in English for each programming language.

We extract $200,000$ posts from StackOverflow by removing code and keeping only the English text.[2] Furthermore, we use the following constraints to reduce noise and poorly constructed English when selecting posts:

1. We only use posts which are the accepted answer.
2. Each post has at least 10 positive votes. The corresponding question post has at least 1 positive vote.
3. We take posts which have at least 300 characters in the text body excluding the code snippet and any code words in the text. This ensures that our corpus has sufficient English tokens. Although we exclude code words, we take only posts that contain a code snippet to ensure that the discussion is about code and not, for example, configuration of an IDE.

To extract the English tokens in StackOverflow posts we extract the necessary data (body *without*

[2]https://archive.org/details/stackexchange, September 2016

code) with a Python HTML library. We merge the posts into a single file and perform the NLP process steps of stemming, lematization, lexicalization and stopword removal.

3.4 Replication

RQ1: How repetitive and predictable is software?

We replicate the work of Hindle *et al.* [26] to ensure that the data we sample produces similar results. We also examine C#, C, JavaScript, Python, Ruby, and Scala. We want to understand if the language and programming paradigm influence the repetitive nature of programming.

3.4.1 Theoretical background and methodology

We give the definitions of n-gram language models, cross entropy, and SelfCrossEntropy and describe how we extract n-grams.

n-gram Language Model

We use the term language model (LM) to mean the probability distributions over a sequence of n tokens $P(k_1, k_2, ..., k_n)$. A LM is trained on a corpus containing sequences of tokens from the language. Using this LM our goal is to assign high probability to tokens with maximum likelihood, and low probability to n-grams with lower likelihood. The primary purpose of modelling a language statistically using LMs is to model the uncertainty of the language by determining the most probable sequence of tokens for a given input.

Consider a sequence of tokens $k_1, k_2, k_3, \ldots k_{n-1}, k_n$ in a document, D. n-gram models statistically calculate the likelihood of the nth token given the previous n-1 tokens. We can estimate the probability of a document based on the product of series of conditional probabilities:

$$P(D) = P(k_1)P(k_2|k_1)P(k_3|k_1, k_2)...P(k_n|k_1, k_2, ..., k_{n-1})$$

Here, $P(D)$ is the probability of the document and $P(k_i)$ is the conditional probability of tokens. We can transform the above equation to a more general form which is given below.

$$P(k_1, k_2, k_3, ..., k_{n-1}, k_n) = \sum_{i=1}^{n} P(k_i|k_1, ..., k_{n-1})$$

This transformation uses the **Markov Property** which assumes that token occurrences are influenced only by limited prefix of length n [75]. Furthermore, we can consider this as a **Markov Chain** which assumes that the outcome of the next token depends only on the previous $n-1$ tokens

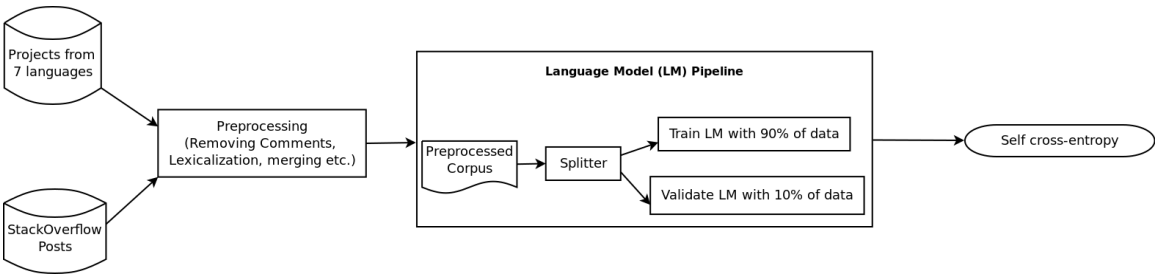


Figure 1: Pipeline for experiments performed in this study

[52]. Thus we can write:

$$P(k_i|k_{i-(n-1)}, ..., k_{i-1}) = P(k_i|k_{i-(n-1)})$$

This equation requires the prior knowledge of the conditional probabilities for each possible n-gram. Computing these conditional probabilities is calculated from the n-gram frequencies. We use these n-grams to determine the entropy of a language corpus including source code.

SelfCrossEntropy

Hindle *et al.*'s [26] calculate the average number of bits, *i.e.* entropy, required to predict the nth token of the n-grams in a document. They use the standard formula for cross-entropy. They define cross-entropy in the context of n-grams. Given a language model M, the entropy of a document D, with n tokens, is

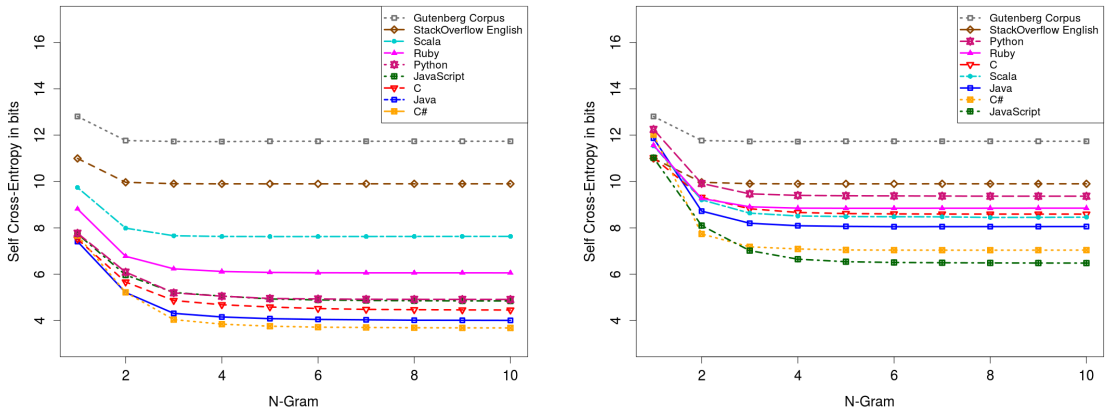
$$H(D, M) = -\frac{1}{n}\sum_{i=1}^{n} log_2 P(k_i|k_1...k_{i-1})$$

They use cross-entropy in a unique manner to define SelfCrossEntropy. Instead of estimating the language model M from another document or corpus, they divide a single corpus into 10 folds. M is then calculated from 9 of the folds and $H(D, M)$ is calculated with D being the remaining fold. The final SelfCrossEntropy is the average value across all folds.

Extracting n-grams

We replicate Hindle *et al.* [26] using the same tools and methodology as shown in Figure 1. We remove the source code comments. We lexicalize each source file in the project using ANTLR[3] to extract code tokens. Then we merge all the lexicalized files to create a corpus. For example, to get the SelfCrossEntropy of the Java language, we process all `.java` files. Then we merge the

[3]ANTLR4 `http://www.antlr.org/`



(a) SelfCrossEntropy with SimpleSyntaxTokens (raw source code)

(b) SelfCrossEntropy without SimpleSyntaxTokens

Figure 2: SelfCrossEntropy of programming languages with and without SimpleSyntaxTokens

processed files to create our final corpus. To calculate the SelfCrossEntropy, a single corpus is split into 10 folds. Ten-fold cross validation is used with the probability estimated from 90% of the data and validated on the remaining 10%. The results are averaged over the 10 test folds. We use MIT Language Model (MITLM) toolkit[4] to calculate the SelfCrossEntropy for each data set. MITLM uses techniques for n-gram smoothing to deal with unseen n-grams in the test fold (see Hindle *et al.* [26] for further discussion). We calculate the SelfCrossEntropy for token sequences, *i.e.* n-grams, from 1-grams to 10-grams for each programming corpus, the Gutenberg corpus and English text on StackOverflow corpus. The processing pipeline for the experiments is shown in Figure 1.

3.4.2 Replication Result

How repetitive and predictable is software?

Figure 2a shows the replication of Hindle *et al.*'s [26] work, including six additional programming languages and StackOverflow posts. All the programming languages under consideration for this study show the same pattern of SelfCrossEntropy. The highest SelfCrossEntropy is observed for unigram language models. The value of SelfCrossEntropy declines significantly for bigram and trigram models. From 3-grams to 10-grams the SelfCrossEntropy remains nearly constant. Since we are able to replicate Hindle *et al.*'s result, we are confident that our dataset is large and diverse enough to test the "naturalness" hypothesis in new contexts.

[4]MITLM `https://github.com/mitlm/mitlm`

While the pattern is the same, the values of SelfCrossEntropy are substantially different for each language. With Scala being much less repetitive than C#. We conclude that the pattern of decreasing SelfCrossEntropy across n-grams holds from the Hindle *et al.*'s work. However, the difference among languages forces us to conjecture that the syntax of the language is artificially reducing its SelfCrossEntropy.

3.5 Artificial Repetition

RQ2. how repetitive and predictable is code once we remove SimpleSyntaxTokens*?*

Standard preprocessing steps in NLP involve the removal of stopwords and punctuation [66, 45]. Stopwords, including articles, *e.g.,* "the", and prepositions, *e.g.,* "of", are removed in information retrieval tasks because they introduce noise in the data set reducing the likelihood of retrieving interesting information. In our work, we examine the impact of three types of SimpleSyntaxTokens: separators such as brackets and semi-colons; keywords, such as `if` and `else`; and operators, such as plus and minus signs. Hindle *et al.* did not remove these SimpleSyntaxTokens and in their autocompletion model they suggest a SimpleSyntaxToken approximately 50% of the time. As a result, an autocompletion tool would suggest an obvious separator before a useful token such as an API call. In this section, we examine the impact of each type of SimpleSyntaxToken token on the apparent repetitiveness of code.

3.5.1 Background and Methodology

To identify the SimpleSyntaxTokens for each programming language, we examined the language specification to identify the keywords, separators, and operators. We calculate the percentage of SimpleSyntaxTokens in each programming language. Then we remove SimpleSyntaxTokens from the corpus and measure the entropy of n-grams without the language specific tokens. We report the change in SelfCrossEntropy of the n-grams after the removal of language specific tokens and answer the following questions:

1. What percentage of total tokens are SimpleSyntaxTokens?
2. What is the change in SelfCrossEntropy after removing SimpleSyntaxTokens?
3. How repetitive is code without SimpleSyntaxTokens compared to English?[5]

[5]For the English corpora we removed the standard stopwords with the NLTK toolkit.

Table 2: Percentage of language specific token, *i.e.* SimpleSyntaxTokens, for each programming language corpus

Language	Separators	Keywords	Operators	Total
Java	44.00%	9.36%	5.85%	59.21%
C#	42.57%	10.96%	7.55%	61.08%
C	39.23%	5.50%	15.14%	59.87%
JavaScript	47.21%	6.87%	6.53%	60.61%
Python	41.98%	4.99%	6.42%	53.39%
Ruby	23.37%	8.37%	8.93%	40.67%
Scala	39.27%	7.40%	7.28%	53.95%

3.5.2 Results and Discussion

What percentage of total tokens are SimpleSyntaxTokens? Stopwords are removed during natural language information retrieval tasks because their high prevalence introduces noise reducing the likelihood of retrieving highvalue information. When applied to our programming corpora, in Table 2, we see that SimpleSyntaxTokens account for a high percentage of total tokens. Across the programming languages, JavaScript has the highest number of SimpleSyntaxTokens at 60% of total tokens, while the smallest percentage is 41% for Ruby. Separators account for the largest proportion of SimpleSyntaxTokens, between 23% and 47% of all tokens.

The main implication from Table 2 is that SimpleSyntaxTokens dominate the tokens in all corpora and when included make code look artificially repetitive.

What is the change in SelfCrossEntropy after removing SimpleSyntaxTokens?

We remove the SimpleSyntaxTokens and recalculate the SelfCrossEntropy. In Table 3 we see that the increase in SelfCrossEntropy, *i.e.* a decrease in repetitiveness, is dramatic. For Java, we see that from 1-grams to 6-grams we need a respective increase of 68%, 67%, 90%, 97%, 98% more bits. After 6-grams we need a nearly constant 100% increase in bits. Clearly more information is required to encode Java programs without the artificially repetitive SimpleSyntaxTokens.

How repetitive is code without SimpleSyntaxTokens compared to English?

We investigate the difference in SelfCrossEntropy between programming languages and English by reporting the number of additional bits necessary to encode English. Hindle *et al.* [26] report a maximum average per-word entropy of approximately 8 bits for English and 2 bits for Java, which means that English requires 4 times as many bits, while for 2-grams and 3-grams, English requires 2 and 2.7 times as many bits. Similarly we find that before removing SimpleSyntaxTokens,

Table 3: Percentage increase in SelfCrossEntropy after the removal of SimpleSyntaxTokens

Language	1-gram	2-gram	3-gram	4-gram	5-gram	6-gram	7-gram	8-gram	9-gram	10-gram
Java	60.18	67.40	90.17	94.70	97.49	98.75	99.67	100.55	100.75	101.00
C#	56.73	48.26	77.66	84.34	87.52	89.42	90.04	90.65	90.94	91.16
C	46.75	64.50	81.33	84.95	87.85	90.30	91.80	92.26	92.66	92.85
JavaScript	42.48	35.72	34.61	31.47	32.58	33.03	33.43	33.60	33.66	33.69
Python	57.67	62.81	82.20	86.07	89.45	90.11	90.55	90.53	90.65	90.70
Scala	18.48	15.29	12.74	11.60	11.34	11.22	11.13	10.75	10.78	10.85
Ruby	31.18	36.82	42.91	44.65	45.55	45.87	45.99	46.02	46.05	46.07

we need 1.7, 2.3, 2.7, 2.8, 2.9, more bits for 1-grams to 5-grams for Java. After 5-grams the increase is constant at 2.9 times.

However, without SimpleSyntaxTokens the number of additional bits required is substantially less for Java: 1.0, 1.4, 1.4 additional bits for 1-gram to 3-grams and remains constant at 1.5 from 4-grams to 10-grams. This provides further evidence that SimpleSyntaxTokens clearly account for a large proportion of the repetitiveness in Java. With slight variation in the actual number, this result generalizes to the other programming languages in Figure 2b.

As we discussed in the data section, the Gutenberg corpus contains a wide range of English writing styles, topics, and authors. In contrast, the programming corpora used in our work and that of Hindle *et al.*'s are for single programming languages. To provide a more comparable English corpora we processed StackOverflow posts related to each programming language. We find that SelfCrossEntropy of English on StackOverflow is highly similar to that of code. For example, Java requires .9 times as many bits as StackOverflow English to encode 1-grams. Clearly the vocabulary on StackOverflow is very limited. For 2-grams, 1.1 times as many bits are required and this number remains constant at 1.2 for 3-grams to 10-grams. After 2-grams we see that sequences of token usages are larger in StackOverflow. This is likely because classes and methods tend to be used together in Java. However, compared to the originally reported 4 times as many bits, or 300% more bits the removal of SimpleSyntaxTokens shows a 1.1 to 1.2 times as many bits or 10 to 20% more bits. This result is consistent across programming languages.

3.5.3 Concluding discussion on SimpleSyntaxTokens

Hindle *et al.* were "worried" by the questions that we ask in this section [26]. They asked "is the increased regularity we are capturing in software merely a difference between the English and Java languages themselves? Java is certainly a much simpler language than English, with a far more structured syntax." To answer this question, they conducted an experiment were they compared the SelfCrossEntropy of a single program with the cross entropy of predicting the tokens in one Java program with those in other Java programs. They conclude that because the entropy for single programs is lower than the entropy between programs that regularity of software is "not an artifact of the programming language syntax." However, in both cases the programs were written in the same language, Java, using the same syntax. Their experiment clearly does not control for simple syntactical regularities in the Java language. In contrast, in our study we remove SimpleSyntaxTokens and find that the regularity of programs drops dramatically. We conclude that that the syntax of programming languages artificially reduces the entropy of software. Our findings suggest that software engineers should follow the NLP practice of removing stopwords and punctuation, in this case SimpleSyntaxTokens, to reduce the noise they introduce and to make higher value autocompletion suggestions.

3.6 API Usages

RQ3. How repetitive and predictable are Java API usages?

API code is used across multiple projects in the same manner regardless of the domain of the project. We extract Java API tokens and determine how predictable their usage is. We conjecture that sequences of API elements, *i.e.* API usages, should be more repetitive and predictable than general program code.

3.6.1 Background and Methodology

We extract the Java API elements from the Java Platform Library Standard Edition 7 Specification[23]. We remove all tokens from the Java corpus which are not part of Java standard libraries. The set of API elements includes package, class, field, and method names. For the Java corpus, we calculate the SelfCrossEntropy for the API usage of size 1 to 10-grams.

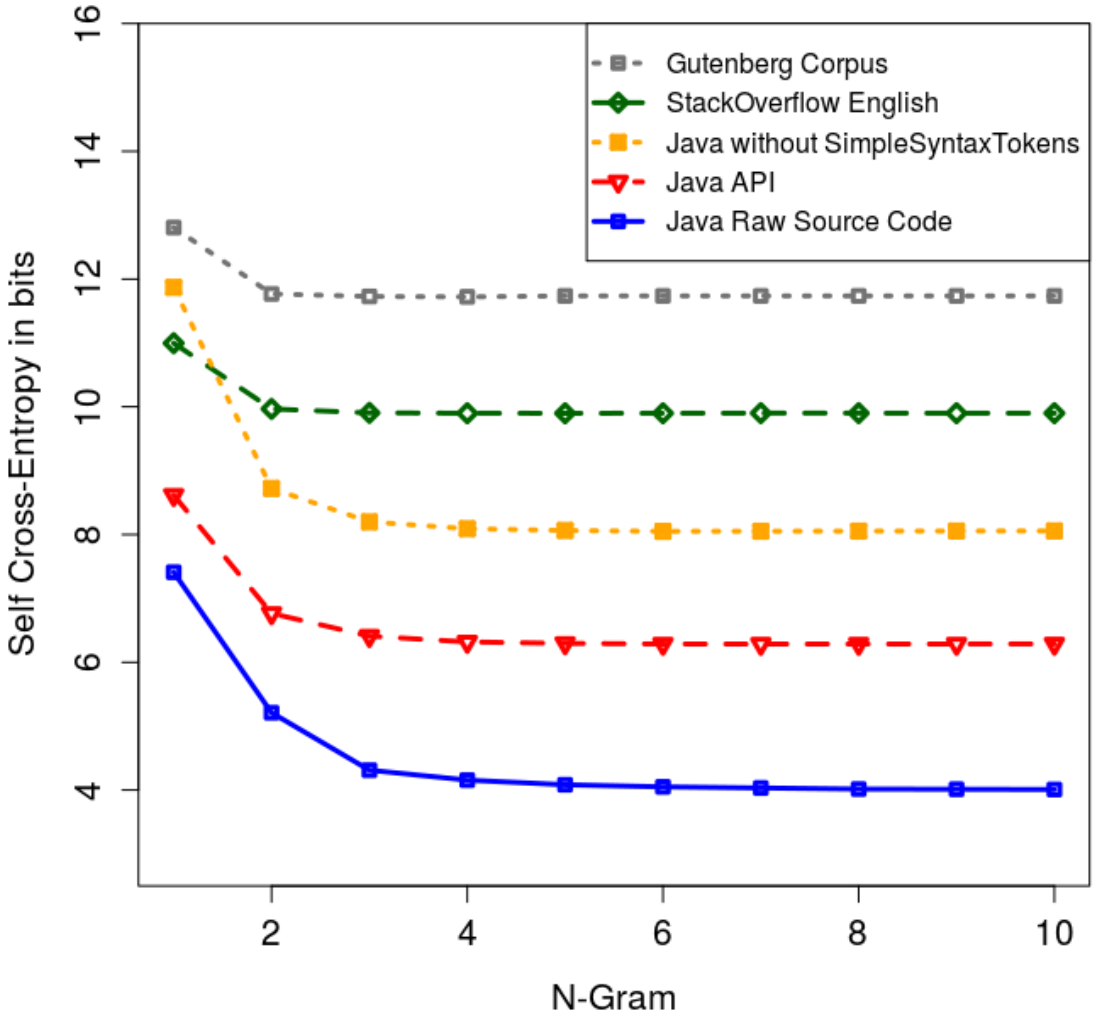


Figure 3: Comparing the Java API SelfCrossEntropy with raw Java source code, Java source code without SimpleSyntaxTokens, and English

3.6.2 Results and Discussion for API Usages

Figure 3 compares the SelfCrossEntropy of n-gram API usages in Java to raw Java, Java without SimpleSyntaxTokens, StackOverflow English, and Gutenberg. We find that the SelfCrossEntropy of the Java API is less repetitive and predictable than the raw corpus which contains SimpleSyntaxTokens. This result derives from the high proportion of SimpleSyntaxToken tokens, *i.e.* 57% of tokens in Java are SimpleSyntaxTokens. Java that excludes SimpleSyntaxTokens but includes internal code, requires 20% more bits for 1-grams and a consistent 30% more for 2 to 10-grams compared with the Java API. This is likely because the domain specific tokens, for example, the "BankAccount" class in a banking application, are used much less repetitively than the API code, such as "String" or "InputStreamReader" classes in standard Java 7 libraries.

The corresponding numbers for English on StackOverflow, are 30% to 60% more bits. For Gutenberg, which includes a diverse set of English texts, 50% to 90% more bits are required. These differences are substantially lower than Gutenberg and raw Java which requires between 70% and 190% more bits to encode the Gutenberg corpus.

We conclude that raw Java code that contains SimpleSyntaxTokens is more repetitive than the Java API usages likely due to the repetitive use of syntax rules. In contrast, we find that Java API is more repetitive than general Java code that does not contain SimpleSyntaxTokens. Our finding that Java API usages are quite repetitive quantifies the truth underlying the large and successful literature on suggesting sophisticated API autocompletions (*e.g.,* [44, 5, 62, 10, 50]).

3.7 Code Graphs

RQ4: how repetitive and predictable are graph representations of Java code?

The assumption made by the n-gram language model is that the current token can be predicted by the sequence of $n-1$ previous tokens. For many natural languages the assumption holds as they are interpreted sequentially from left to right. In contrast, compilers and humans do not usually process programs sequentially. In the case of compilers, parse trees or syntax trees are generated to provide abstract representations. Eyetracking studies of developers reading code show a nonlinear movement along the control and data flow of the program [11] which differs from natural language reading strategies [18], for example, by focusing on method signatures [63] and following beacons [17] in the code. In this section, our goal is to measure how repetitive an abstract graph representation of code is and to understand if it reveals repetitions that cannot be identified with n-grams.

3.7.1 Background and Methodology

In order to determine how repetitive code graphs are, we need a graph extraction technique that is able to satisfy the following requirements:

1. Extract the code graphs from a large number of projects that may not be able to compile due to, for example, external dependencies.

2. Filter out granular information, such as variables and expressions, to include only control and data dependencies among class objects and methods in the code graphs.

3. Identify isomorphic code graphs to determine the occurrence frequency of each graph.

We evaluated the Eclipse AST parser, and found that it had critical limitations:

1. The Java project dependencies must be present for each project.

2. The AST includes lowlevel details, such as variable names, which would artificially reduce graph frequencies.

3. Techniques [34, 60, 28] to identify structural similarities in the code using ASTs are computationally expensive[48].

In summary, the Eclipse AST parser is designed for static analysis, but is not appropriate for statistical based recommendations.

In contrast, GrouMiner [47] was designed to extract GRaph-based Object Usage Models (Groums) and to efficiently calculate isometric graphs. Below we describe the steps necessary to extract the frequency of Java code graphs:

1. Recoder is used to extract an AST without the need to compile the program [41].

2. GrouMiner transforms the AST for each method body into a Groum. The nodes in a Groum represent constructors, method invocations, field accesses, and branching points for control structures. The edges represent temporal, data and control dependencies between nodes.

3. Graph induction is used to generate subgraphs of the Groum for a specified size, in our case 2, 3, and 4-node graphs.

4. GrouMiner computes the occurrence frequencies of each Groum using [48] technique.

3.7.2 Data

We use GrouMiner to capture the occurrence frequency of each Groum in the Java projects used in the previous n-gram sections. In the previous section we found that API code tends to be more

repetitive and predictable across multiple projects. As a result, we capture GROUMS containing API usages from the Java Platform Standard Edition 7 Specification [23]. We include GROUMS that contain at least one Java API node. We eliminate GROUMS which contain only control flow structures or only contain internal code. To perform a fair comparison with n-grams, we use the same inclusion and exclusion criteria to filter the n-grams tokens. Our goal is to study the inherent degree of repetition for the two representations, graphs and n-grams. In the previous sections, we calculated the SELFCROSSENTROPY by predicting the nth token for n-grams in 10-fold cross validation. Since graphs are not sequential, the most appropriate prediction comparison is unclear. To avoid this problem, we examine the underlying frequency distribution for each set of n-grams and n-node graphs on the same set of Java projects. This strategy of examining the distribution has been employed in many previous works examining code structure [40, 74, 8, 16]. The more left skewed the distribution the more repetitive and predictable the representation.

3.7.3 Results and Discussion for Java Code Graphs

We collect GROUMS with 2, 3 and, 4 nodes and the corresponding n-grams. We measure the occurrence frequencies of each GROUM and n-gram across the Java projects. Since graphs represent an abstraction of code, we conjecture, that on the same code, GROUMS will have a stronger Pareto-type distribution than n-grams, *i.e.* graphs will be more repetitive and left skewed. In Figure 4 we plot the top 20% of the n-grams and n-node GROUMS against the percentage of total n-grams and n-node GROUMS, respectively. We see both n-grams and n-node GROUMS are highly left skewed. For example, the top 20% of n-grams account for 76%, 58%, 51% for all instances of 2, 3, and 4-grams, respectively. The corresponding value for the top 20% of n-node GROUMS account for 81%, 73%, 72% of instances of 2, 3, and 4-node graphs, respectively. The top 20% of graphs are 5, 15, 21 percentage points more frequent than the top 20% of n-grams. Furthermore, the drop between 2-nodes and 3-nodes is much less than between 2-grams and 3-grams, indicating that graphs remain highly repetitive with increasing size.

Table 4 shows the complete distribution for the 10 to 90% for graphs and n-grams. The column at 20% is represented in the Figure 4 but for space reasons we cannot show the graphs as this would represent 18 lines. The table shows that the pattern remains clear, with n-nodes being more left skewed than n-grams. We conclude that graph representations are much more repetitive than sequential n-gram representations.

3.7.4 Illustration of Graphs

We have quantitatively determined that GROUMS are more repetitive and predictable than n-grams. In this section, we provide illustrations of why they are more repetitive. For example, an n-gram

sequence will not capture the relationship between `File.open()` and `File.close()`, because there will always be other tokens, such as `File.read()`, between these API calls. Although we removed SimpleSyntaxTokens in this section, if they are included the problem is exacerbated because obvious tokens lie between related API calls. In contrast, Groums will always contain a data dependency edge between `File.open()` and `File.close()` even when internal classes are present. The temporal program flow will still be captured by control edges.

A more complex example from our corpus of Java programs illustrates the transformation of separate program code fragments into a common abstract Groum with 4-nodes. The Groum in Figure 5 represents the API usage pattern of iterating through a `java.util.HashMap` with an enhanced `for` loop. The Groum is an abstract representation of the code in Listings 3.1 to 3.4 as well 23 other classes in the Neo4J project. Specifically, the Groum contains the data and control flow dependencies between `Map.entrySet(), Map.Entry.getKey(), Map.Entry.getValue(),` and an enhanced `for` loop. For example, in Listing 3.2 the code iterates through a hashmap of tracked client sessions and in Listing 3.1 the code iterates through a hashmap of throughput reports.

Below we use the listings to show the important differences between the Groum and n-gram models.

Abstraction: From examining the listings, it is clear that no n-gram model would consider these code fragments as identical. There are many internal classes and SimpleSyntaxTokens between these API elements. Even when only API elements are considered there would be no direct sequence with `Map.entrySet()` preceding `Map.Entry.getValue()`. This relationship is only captured as a data dependency in a graph.

Size: the size of the n-gram necessary to capture each of these code fragments would be much larger than the 4-node Groum. For example, if we include SimpleSyntaxTokens, for the respective listings we need sequences with 34, 32, 30, and 38 tokens to represent the code in each listing. Without SimpleSyntaxTokens the corresponding number of tokens is smaller but still quite large at 14, 15, 13, and 15 tokens per listing.

We conclude that Groums capture information about the control and data flow at a higher level of abstraction which makes them a more repetitive representation of code than n-grams. Graphs are also a more realistic representation of code than sequential n-grams as compilers and humans do not process code sequentially. Graphs are more appropriate for statistical code autocompletion because they can suggest non-sequential relationships that cannot be represented in an n-gram model.

Listing 3.1: TransactionThroughputChecker.java

```
private void printThroughputReports( PrintStream out ) {
out.println( "Throughput␣reports␣(tx/s):" );
for ( Map.Entry<String,Double> entry : reports.entrySet() ) {
```

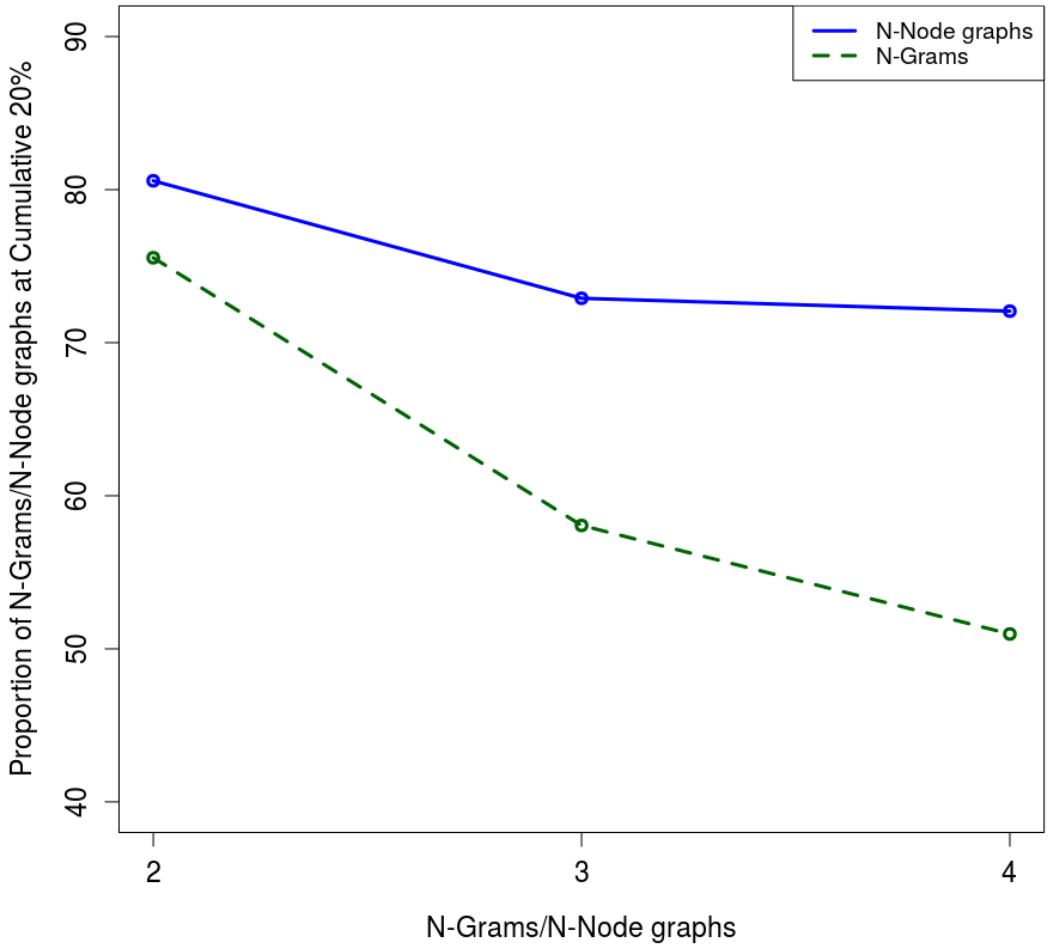


Figure 4: The top 20% of the n-grams and n-node graphs account for the y-axis% of the usages. For example, the top 20% of the n-node graphs account for 80.6% of all usages.

Table 4: The cumulative proportion of n-node graphs and n-grams from 0% to 100% in 10 point increments for all usages. For example, the top 40% of the n-node graphs account for over 89% of all usages. The table shows the left skew of the distributions.

Cumulative Percentage	2-node graph	3-node graph	4-node graph	2-gram	3-gram	4-gram
0	0.00	0.00	0.00	0.00	0.00	0.00
10	71.46	62.18	61.50	66.22	47.34	38.72
20	80.58	72.90	72.06	75.55	58.06	50.97
30	85.82	79.21	78.38	80.95	65.96	57.13
40	89.14	84.11	83.53	85.58	70.82	63.26
50	92.21	87.75	87.14	87.98	75.69	69.38
60	93.76	90.20	89.71	90.38	80.55	75.50
70	95.32	92.65	92.28	92.79	85.41	81.63
80	96.88	95.10	94.86	95.19	90.27	87.75
90	98.44	97.55	97.43	97.60	95.14	93.88
100	100.00	100.00	100.00	100.00	100.00	100.00

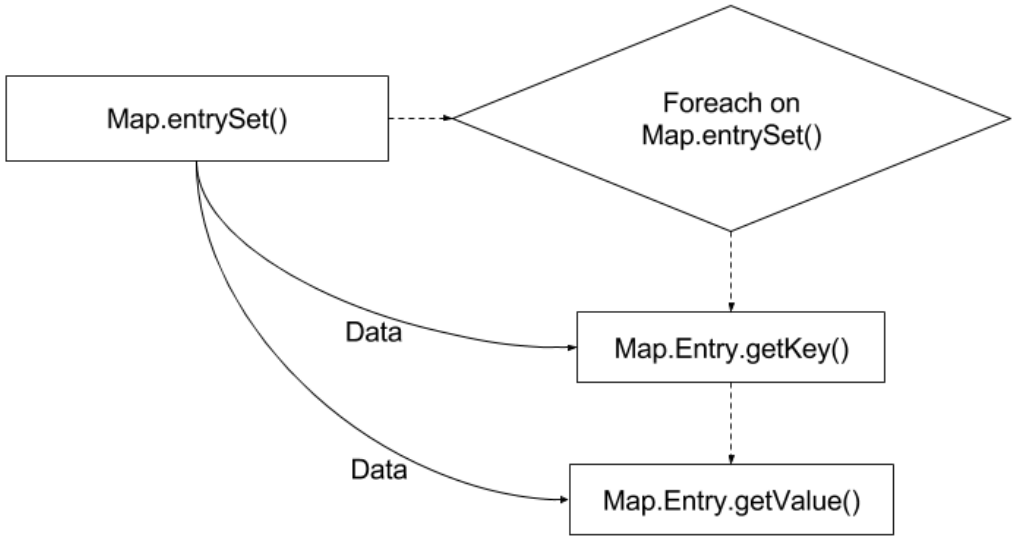


Figure 5: A GROUM representing iteration through a HashMap, which is an abstraction of the code in Listings 1 through 4.

```
out.println( "\t" + entry.getKey() + "␣␣" + entry.getValue() );
}
out.println();
}
```

Listing 3.2: GlobalSessionTrackerState.java

```
public GlobalSessionTrackerState newInstance() {
GlobalSessionTrackerState copy = new GlobalSessionTrackerState();
copy.logIndex = logIndex;
for ( Map.Entry<MemberId,LocalSessionTracker> entry : sessionTrackers.
    entrySet() ) {
copy.sessionTrackers.put( entry.getKey(), entry.getValue().newInstance() );
}
return copy;
}
```

Listing 3.3: ListAccumulatorMigrationProgressMonitor.java

```
public Map<Strin\sectiong,Long> progresses() {
Map<String,Long> result = new HashMap<>();
for ( Map.Entry<String,AtomicLong> entry : events.entrySet() ) {
result.put( entry.getKey(), entry.getValue().longValue() );
}
return result;
}
```

Listing 3.4: ExpectedTransactionData.java

```
private Map<Node,Set<String>> cloneLabelData( Map<Node,Set<String>> map ) {
Map<Node,Set<String>> clone = new HashMap<>();
for ( Map.Entry<Node,Set<String>> entry : map.entrySet() ) {
clone.put( entry.getKey(), new HashSet<>( entry.getValue() ) );
}
return clone;
}
```

3.8 Limitations and Validity

Limitations of graphs: To extract an AST from a large number of projects we used Recoder [41]

because it does not require the project to be compilable. Recoder, like PPA tool [19], has known limitations that lead to unknown nodes in a graph. When a node is unknown we are unable to generate a GROUM. For 2, 3, and 4-node graphs we have 4.5%, 8.0%, 10.6% of graphs that contain an unknown. These percentages are inline with the 90% accuracy of the state-of-the-art partial programs analysis and code snippets analysis tools [19, 41, 61].

A second limitation is that identifying isomorphic graphs using GROUMINER [48] is computationally expensive. In this work, we calculated GROUM sizes up to 4-nodes. Based on our analysis, we have seen that the probability distribution of graphs for 3-node and 4-nodes remain constant indicating that, like n-grams, higher n-node graphs exhibit similar degrees of repetition. Furthermore, since graphs are at a higher degree of abstraction, fewer nodes are necessary to represent the same block of code when compared to sequential n-grams.

Limitations of SELFCROSSENTROPY*:* In terms of entropy calculations, SELFCROSSENTROPY is an extension of cross entropy whereby 10-fold cross validation is used to calculate the per-token average of the probability with which the language model generates the test data [26]. Ideally, we would calculate all possible combinations of the next token, however, as Shannon [68] points out, this is impractical with $O(t^N)$, where t is the number of unique tokens and N is the total number of tokens in the corpus. For each language in our corpus there are over 300k unique tokens and 20 million total tokens. As a result, SELFCROSSENTROPY serves as a good approximation of entropy.

Reliability and External Validity: By examining a diverse set of languages we increase the generalizability of our results. Furthermore, in RQ1 our goal was to replicate previous work and to ensure that our data and scripts produced consistent results. We were successful in this replication, increasing the validity of the data used in the novel work in subsequent research questions. In our replication package [1], we have included all processed n-gram and graph data as well as the scripts used in our processing pipeline to allow other researches to validate and extend our work.

3.9 Related Work

Research into language entropy.

Basic research into understanding redundancy and measuring entropy in languages has a long history. Shannon [68] developed statistical measures of entropy for the English language. Gabel and Su [22] noted high levels of redundancy in code and Hindle *et al.* [26] continued this work demonstrating that software is highly repetitive and predictable. Recent works have replicated these software findings on a giga-token corpus [3] and looked at the entropy in local code contexts [71]. Others have examined repetition at the line level [59] or in other domains such as Android Apps[36, 5]. In each case, code has been found to be repetitive and predictable. In our work, research question 1

replicates Hindle *et al.*'s work expanding it to multiple programming languages. We noted differences among programming languages and conjectured that these differences may be due to syntax. Following NLP practices of removing stopwords and punctuation, we remove operators, separators, and keywords, and find that without these highly repetitive tokens software is much less repetitive and predictable. While we support the general conclusion that code is repetitive and predictable, we find that it is not much more repetitive than English. This conclusion is important because it will reframe the ease with which statistical predictions about software can be made.

Research on code validation and checking.

Most existing tools to find defects and other code faults use static analysis. Recent works have focused on using the statistical properties of the languages to find bugs and to suggest patches. For example, Campbell *et al.* [13] find that syntax errors can be identified using n-gram language models. Ray *et al.* [58] identified bugs and bug fixes in code because buggy code is less natural and has a higher entropy. Santos and Hindle [67] used the n-gram cross entropy of text in commit messages to identify successfully commits that were likely to make a build fail. Our research confirms that statistical code checking will work much better on syntax or APIs than on internal classes as these former types are much more repetitive.

Research into autocompletion and suggestions.

Modern IDEs contain an autocompletion feature that usually uses the structure of the language to make suggestions. Research into code suggestion has long known intuitively that code is repetitive. For example, textual similarity of program code [4], commit messages [14], and API usage patterns [44] have been exploited to guide developers during their engineering activities. Building on this work, Zimmermann *et al.* [76] used association rule mining on CVS data to recommend source code that is potentially relevant to a given change task. Recent work by Azad *et al.* [5] has extended this work to make change rule predictions from a large community of similar Apps and the code discussed in StackOverflow discussions.

Advanced autocompletion techniques have leveraged the history of applications and the repetitive nature of programming to suggest code elements to developers. Robbes and Lanza [62] filtered the suggestions made by code completion algorithms based on, for example, where the developer had been working in the past and the changes he or she had made. Bruch *et al.* [9] suggested appropriate method calls for a variable based on an existing code base that makes similar calls to a library. Buse and Weimer [10] automatically generate code snippets from a large corpus of applications that use an API. Duala-Ekoko and Robillard [21] use structural relationships between API elements, such as the method responsible for creating a class, to suggest related elements to developers. Works by Nguyen *et al.* [50] use statistical language models to autocomplete code accurately. Nguyen and Nguyen [47] expanded this work to graphs in order to create suggestions that are syntactically valid. Much of this

work focuses on suggesting API elements. Our work suggests that API usages are substantially more repetitive and predictable than general code, explaining the success of API suggestion approaches. Furthermore, we show why graphs are a more appropriate representation of code, and we hope this will encourage future researchers to focus on graph abstractions instead of sequential tokens.

Statistical translation.

Recent works have mirrored the success of Statistical Machine Translation in natural languages, *e.g.,* Google Translate, and applied these approaches to translating English to code. For example, SWIM [56] uses a corpus of queries from Bing to align code and English and generates sequences of API usages. DeepAPI [24] uses recurrent neural networks to translate aligned source code comments with code to translate longer sequences of API calls. T2API [49] uses alignments between English and code on StackOverflow to generate a set of API calls. These calls are then rearranged based on their the likelihood of existing program graphs. T2API can generate long graphs of common API usages from English. Our work provides a frame in which to understand these works. For example, the sequences of SWIM and DeepAPI tend to be short and simplistic as they are restricted by a left-to-right processing of tokens. In contrast, T2API which re-orders API elements in a graph can produce more complex usages.

3.10 Conclusion

Our findings confirm previous work that code is repetitive and predictable. However, it is not as repetitive and predictable as Hindle *et al.* [26] suggested. We have found that the repetitive syntax of the program language makes software look artificially much more repetitive than English. For example, language specific SimpleSyntaxTokens account for 59% of the total Java tokens in our corpus. We conclude that the researcher must ensure that the corpus is tuned and cleaned for the prediction task. If the goal is to recommend statistically tokens that are related to complex software engineering tasks, for example, completing a set of API calls, then suggesting SimpleSyntaxTokens, such as semicolons, that are encoded as rules in a compiler, will simply distract from more interesting recommendations.

We make our scripts, n-grams, and graphs available in our replication package [1] and hope that our work will be used by researchers to select appropriate corpora with sufficient repetition. For example, we conducted a failed experiment to suggest patches based on past fixes using an n-gram language model. Had we had our current analysis there would have been little need to conduct the experiment as it would be obvious that internal class tokens and usages were too infrequent to be used successfully in a statistical model. Future work to complement static analysis with statistical models could allow for appropriate recommendations even when a class is used infrequently.

The success of API usage recommendations flows naturally from our findings. By tuning the vocabulary to API code tokens and examining the usage of these APIs element across many programs there is sufficient repetition to make accurate recommendations.

Software recommender tools are moving from simple single element autocompletions to multi-element, non-sequential recommendations of code blocks. Our work shows that different representations of code have different degrees of repetition. Graph representations, such as GROUMS, allow for a higher degree of abstraction and the data and control flow allow for non-sequential relationships. Furthermore, the abstract nature of graphs allows for a more concise representation that reduces the number of noise tokens in code predictions. We hope that future work will focus on new code representations that are tailored to statistical code suggestion allowing for complex and useful recommendations.

Chapter 4

Cleaning StackOverflow for use in Machine Translation

Note: This chapter has been submitted to a conference and has been included verbatim in this manuscript thesis.

4.1 Abstract

Generating source code API sequences from an English query using Machine Translation (MT) has gained much interest in recent years. For any kind of MT, the model needs to be trained on a parallel corpus. In this paper we clean StackOverflow, one of the most popular online discussion forums for programmers, to generate a parallel English-Code corpus. We contrast three data cleaning approaches: standard NLP, title only, and software task. We evaluate the quality of the each corpus for MT. We measure the corpus size, percentage of unique tokens, self-cross entropy, and per-word maximum likelihood alignment entropy. While many works have shown that code is repetitive and predictable, we find that English discussions of code are also repetitive. Creating a maximum likelihood MT model, we find that English words map to a small number of specific code elements which partially explains the success of using StackOverflow for search and other tasks in the software engineering literature and paves the way for MT. Our scripts and corpora are publicly available.

4.2 Introduction

The process of translating between two languages automatically is known as Machine Translation (MT). Recent advances and computational power have increased the popularity of MT. MT techniques can be broadly classified into three classes: Statistical Machine Translation (SMT)[33, 2, 38], Example-based Machine Translation (EBMT)[69], and Neural Machine Translation (NMT)[6, 46]. Although MT approaches differ in terms of theory, algorithms, and efficiently, all approaches require a high volume and low noise *parallel corpus*[31, 7] or *bitext*[25]. Application of MT algorithms is not limited to translation between natural languages. In recent years, MT techniques have been used to translate from natural language to programming languages[24, 49, 56].

StackOverflow, a developer question and answer forum, can be seen as a bilingual corpus because it discusses programming in both English and code. For example, in Figure 6 we see a post the answers in both English and code the question "How can I refresh the cursor from a CursorLoader?" English words in the post, such as "data will be discarded" can be aligned with code elements such as `restartLoader()`. These alignments can then be used in machine translation. Unfortunately, StackOverflow posts are noisy because people write posts in an informal manner. Examples of noise in StackOverflow includes incorrect spelling, inappropriate use of punctuation, use of acronyms without elaboration, and grammatical mistakes. This results in a degradation of the quality of corpus texts. We found while applying existing SMT such as Phrase based MT and Recurrent Neural Network (RNN) based MT that the noise reduced the quality of translation. We clean StackOverflow so that it can be used in MT. Removing the noise from the StackOverflow data without any significant loss of relevant information is challenging. In this paper, our goal is to experiment with different data cleaning techniques ranging from general Natural Language Processing (NLP)[29] to Software Engineering specific techniques[70], and determine which techniques yield a corpus that can be used for various statistical tasks, such as SMT.

4.2.1 Data Cleaning Approaches

We prepare StackOverflow using the following cleaning approaches:

C0: Raw data approach

We prepare this corpus extracting raw text and source code from StackOverflow posts. We do not perform any kind of processing on the English text so this serves as a baseline corpus for comparison purposes. We extract code elements using *ACE*[61], which was built for extracting code elements from freeform StackOverflow text. The alignment is at the post level with the English being aligned with the extracted code elements.

C1: Thread title approach

Many previous researchers report that StackOverflow thread titles contain valuable information in a very brief yet precise manner[65, 54, 12]. We prepare the English corpus with StackOverflow titles only. We remove stopwords and stem the text. For the code corpus we extract code from only the accepted answer posts using *ACE*.

C2: Standard NLP approach

For this corpus we use NLP processing including keyword extraction, stopword removal, and stemming. For code extraction we use *ACE*.

C3: Software engineering task approach

Treude *et al.* developed *TaskNavigator*[70] for extracting development tasks for software documentation. We use this tool to extract tasks from StackOverflow posts and use these tasks as our English corpus after stemming and removing stopwords. We extract code elements from the posts using *ACE*.

4.2.2 Evaluation Metrics and Criteria

After preparing the corpora we evaluate each corpus individually using the following two metrics.

EvalSize: Size of corpus

We measure the size of each corpus in terms of number of posts, number of English words, number of code elements, and frequency of code usage.

EvalAlign: Per-Word Maximum likelihood Alignment Entropy

We create MT maximum likelihood alignment model for each English word to code elements. For each corpus we measure the distribution of per-word alignment entropy between the English and code. Low entropy indicates that the expectation maximization algorithm has converged to a limited, precise set of mappings [33].

The rest of the paper is structured as follows. In Section 4.3 we detail our data and data cleaning approaches. In Section 4.4 we evaluate each corpus for MT. Finally we conclude the paper by summarizing our contribution and briefly discussing some potential future works in Section 4.5.

4.3 Corpus Cleaning

StackOverflow is the most popular Q&A forum where people discuss programming related issues ranging from how to solve a problem in a particular language to the best programming practices [43]. One of the key features of StackOverflow which makes it suitable for bilingual parallel corpus is the presence of both the verbal description of how to solve a problem and example code snippet showing how to implement the solution.

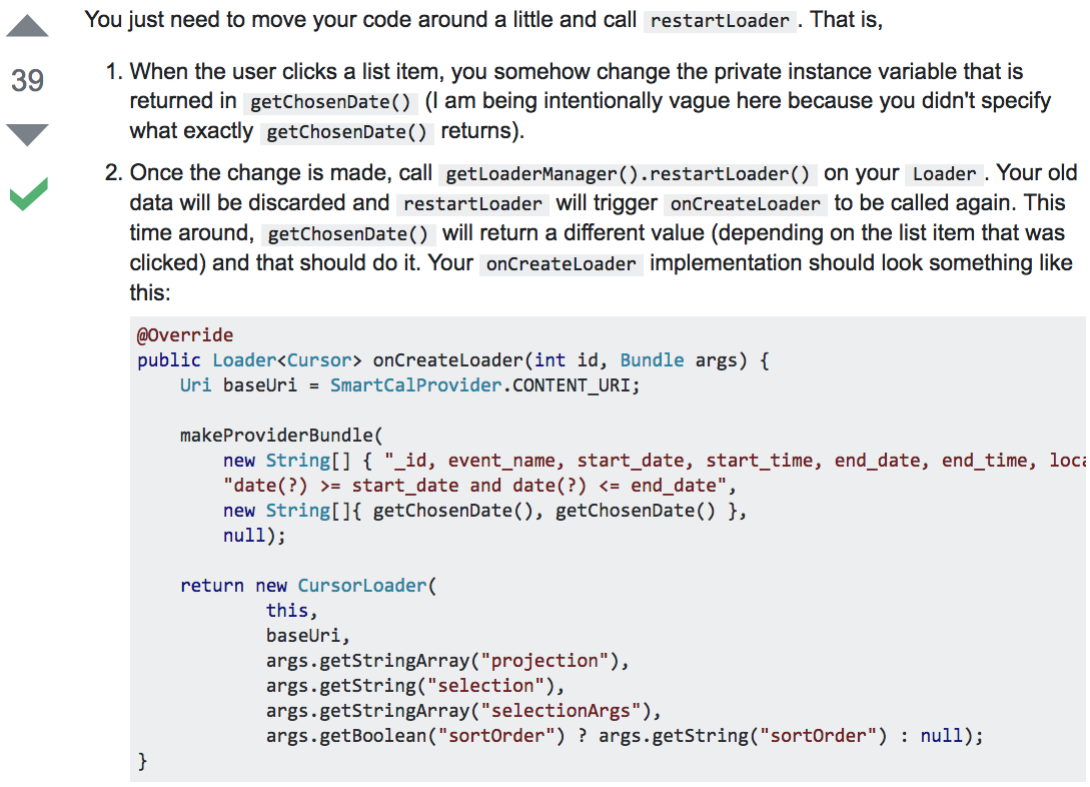


Figure 6: An example of a StackOverflow post that answers in both English and code the question, "How can I refresh the cursor from a CursorLoader?" English words, such as "discarded", can be aligned with code elements, such as `restartLoader()`, for use in MT.
Post available at: `https://stackoverflow.com/a/11092861/1055441`

Figure 6 shows an example of a typical StackOverflow post. The is about refreshing the cursor from CursorLoader in an Android application. A step-by-step solution has been described in English which is followed by a sample code snippet showing how to implement the solution. As a result one can get the solution to a programming problem both in English and source code from a StackOverflow post making it a suitable for bilingual English-code parallel corpus.

In this work, we process posts that are tagged with "android " between September 2011 to September 2016. We perform corpus preparation in multiple experimental settings. The outcome of each setting is then processed by an n-gram language model and a word-based aligner to determine the self cross-entropy and the per-word alignment entropy respectively for each corpus.

4.3.1 C0: Raw data approach

We create a corpus with raw data extracted from StackOverflow. For each post in our English corpus we take the title of the thread and the text from the post body. We remove the code elements

and code snippets from the body. We extract the code element for the code using *ACE*. After filtering out posts that do not contain both English and code we have a corpus of 149,476 Android posts.

4.3.2 C1: Thread title approach

Rosen *et al.* [65] state that "titles summarize and identify the main concepts being asked in the post." The title is brief and in contrast to the entire post contains little noise. In this experimental setting we align the English in the title with the code elements that are contained in positively voted answer posts. We apply the follow additional criteria:

- Stopwords are removed from the title.
- The title text is stemmed.
- There are at least three and at most twenty code elements extracted from the answer post.

For example, the title of the post in Figure 6 is "How can I refresh the cursor from a CursorLoader?". For this post extracted English and code will be as follows:

- English: refresh cursor CursorLoader
- Code: Bundle.getString, Uri.baseUri, SmartCallProvider.CONTENT_URI, Bundle.getString, Bundle.getBoolean, String.getLoaderManager, String.getLoaderManager.restartLoader, LoaderCallbacks.onCreateLoader

The final corpus comprises of 106,359 question titles and corresponding posts.

4.3.3 C2: Standard NLP approach

For this corpus we extract keywords from free-form English of the StackOverflow post body for our English corpus. The English is processed by RAKE[64] which is a general purpose keyword extractor. For our code corpus we use ACE[61] to extract the *unique* code elements present in the post body. Code elements can be either inside a code snippet or can be embedded within free-form English. We add the title of the thread at the top of each post due to a reason discussed in section 4.3.2.

Rapid Automatic Keyword Extraction (RAKE) RAKE splits a text into word groups separated by sentence separators or words from a provided list of stopwords. For our implementation we use the list of English stopwords provided in Python's NLTK[37] library. Each of the resulting word groups is a keyword candidate. In the next step the algorithm scores each keyword according to the word co-occurrence graph. The word co-occurrence graph assigns the frequency of unique word-pairs

across the set of candidate keywords. The details of the algorithm can be found in[64]. The stopword list plays an important role in the effectiveness of the RAKE. Some works use manually curated stopwords list for domain specific application. An example is [30] in which the authors initially carried out their experiments with standard information retrieval stopwords list and report that this yielded poor results for the domain specific case of Polish legal texts. They further report that this approach generated a lot of very long keywords, containing many words that are not very informative. To deal with this issue they create their own list of stopwords and achieved better result. However, in our experiment we deal with this problem in a different way. As mentioned before we do not curate any stopwords lists rather we go with the stopwords provided by NLTK. Besides observing the issue of long keywords containing 'uninformative' words, we also observe other issues associated with the software engineering keywords extracted by RAKE. We briefly discuss these issues here.

- Keywords with long sequence of tokens containing 'uninformative' words.
- Presence of stopwords in the multi-token keywords.
- Association of high scores with keywords having very long sequence of tokens.

In order to deal with these issues we add an additional level of filtering after the keywords are returned by RAKE. This filtering includes constraining the length of keywords and the associated score. It also includes a stopwords removal step performed on the extracted keywords to remove any stopwords that may be embedded within the keywords. On the code side of the parallel corpus we put one constraint which is on the number of extracted code elements by ACE[61]. Steps/constraints related to the final level of filtering are as follows:

1. Keywords must be of length between 1 and 4 in terms of number of tokens to be accepted as a valid keyword.
2. Score associated with a keyword must be greater than 5 and less than 50.
3. There has to be at least 3 code elements extracted from the post.
4. Perform stopwords removal on the extracted keywords.
5. Perform stemming with Porter Stemmer[55].

After performing all these five steps (first three steps for constraints checking and last two for post-processing) the extracted keywords and extracted *unique* code elements become a part of our *Keyword-Code* parallel corpus. For the same example in Figure 6 the following English (without stemming) and code pair is returned.

- English: move code, private instance variable, user clicks, list item, old data, different value, discarded, trigger, cursor, refresh
- Code: Bundle.getString, Uri.baseUri, SmartCallProvider.CONTENT_URI, Bundle.getString, Bundle.getBoolean, String.getLoaderManager, String.getLoaderManager.restartLoader, LoaderCallbacks.onCreateLoader

This corpus contains 130,988 Android posts.

4.3.4 C3: Software engineering task approach

The previous approaches do not consider software engineering specific keywords and concepts. We use Treude *et al.*'s [70] software engineering task extractor to provide concise descriptions of the tasks contained in StackOverflow posts. The *TaskNavigator* [70] extracts development tasks from a documentation based on syntactic dependencies in the sentences. Verbs associated with a direct object and/or a prepositional phrase are identified as tasks. They consider four syntactic dependencies for extracting tasks: **direct object**, **prepositional modifier**, **passive nominal subject**, and **relative clause modifier**. Definitions of the these grammatical dependencies are available in [20] and the implementation details along with the performance of *TaskNavigator* are available in [70].

TaskNavigator was designed to extract development tasks from *formal documentation.* However, StackOverflow is an informal and noisy discussion forum. Unlike formal documentation where the text is grammatically correct with precise descriptions, StackOverflow posts contain additional grammatical errors and other erroneous task information. For example, posters tend to give context before giving the precise step by step solution. This results in *TaskNavigator* identifying many unrelated or irrelevant tasks because it only examines sentence structure and the associated dependencies.

In examining these false tasks, we observed true tasks tend to contain a code element in the sentence. For example, short tasks with fewer than three tokens are usually false positives unless one of the tokens is a code element. Therefore, we remove extracted tasks if they contain fewer than three tokens and no code element. The following five steps are an additional filtering on *TaskNavigator*:

For the example shown in Figure 6 the following English-code pair is returned by this approach.

- English: refresh cursor, move code around, change private instance variable, discard old data, return different value.
- Code: Bundle.getString, Uri.baseUri, SmartCallProvider.CONTENT_URI, Bundle.getString, Bundle.getBoolean, String.getLoaderManager, String.getLoaderManager.restartLoader, LoaderCallbacks.onCreateLoader

The final corpus contains 110,009 posts.

4.4 Evaluation

One of the limitations of MT is that there is no straightforward mechanism to judge the quality of the corpus. In order to determine how well a corpus performs one has to train a model on the corpus and test its performance. However, for any MT system training the model takes considerable amount of time. For example, when we trained an RNN based MT model on our raw corpus on a Google cloud virtual machine with 4 Nvidia GPUs each having 12GB of memory and 2496 processor cores, it took us about 6 days to complete the training and see that the translation quality was poor. Therefore it is not a convenient way to go through all the steps of the MT pipeline to check whether a corpus can perform well for a given task or not. In this section we describe two evaluation criteria, which are computationally less expensive to perform, and present our result for each corpus.

4.4.1 EvalSize: Size of corpus

Our first evaluation criterion is size of the corpora. Statistical learning requires a large number of aligned English and code posts. Table 5 shows that number of posts, unique English tokens, unique code elements, and the number of times each code element has been used. For our calculation we consider tokens that occur more than once. We see that the Raw corpus has the largest size and the largest number of unique tokens. However, to do prediction, we need repetitive use of tokens and we can see that most code elements are only used once. In contrast, the other approaches contain similar numbers of tokens and the median usage of code is two. From simple size measures, we can conclude that the Raw corpus likely still contains too much noise, however, the other corpora are difficult to differentiate purely based on size.

4.4.2 EvalAlign: Per-Word Alignment Entropy

We create simple maximum-likelihood machine translation models. Each English word is aligned to one or more code elements using a simple maximum-likelihood model [33]. The lower the alignment entropy for each English word the stronger the mapping to the specific code elements and the better the model. We plot the distribution of per-word alignment entropy in Figure 7. The raw corpus has the highest median entropy indicating that each English word maps imprecisely to many code elements. In contrast, the 75th percentile for the the title corpus, SE task corpus, and Standard NLP corpus 0.8390, 0.6250, 1.060 respectively. In all cases we see that most English words map with a high probability to few code elements. Although all the distribution are highly left skewed, there are some outlier English words that map to a large number of possible elements. The SE tasks extracted

Table 5: Simple size measures of the corpora

Corpus	Posts	Unique English	Unique code	Median code usage
Raw	149,476	46,067	30,188	3
Title	106,359	10,477	12,227	6
Standard NLP	130,988	15,266	15,949	6
Task	110,009	9,738	14,971	6

from *TaskNavigator* shows the most strict mapping. However, the tool requires substantial processing time and in comparison with the simple approach of using titles, the entropy values are similar.

4.5 Conclusion

In this paper, we show the potential of StackOverflow as a bilingual machine translation corpus. We extract the English and code from each post to create aligned corpora. We show that the Raw StackOverflow posts contain substantial noise and do not lead to good MT models. In contrast, the three cleaned corpora: Standard NLP, including keyword extraction, using the StackOverflow title, and extracting SE tasks lead to low entropy alignments between English and code. This maximum likelihood entropy implies that each English word maps to specific code elements that can be used for translations between English and code.

Our work is empirical in demonstrating the steps necessary to prepare SE corpora. Like Hindle *et al.*'s [26] our results provide an empirical basis on which to understand the literature. For example, many researchers have used StackOverflow posts for performing experiments on issues related to empirical software engineering. Wong *et al.* mined StackOverflow data to autogenerate source code comments [73]. Pinto *et al.* studied software energy consumption from StackOverflow discussion in [53]. Wong *et al.* in [73] studied how developers interact in StackOverflow discussion. In a similar work, Chowdhury *et al.* filtered out off-topic posts from online discussion forums [15]. Finally, Nguyen *et al.* [49] created a tool demonstration using StackOverflow to translate from English into code template graphs. We make our data and corpus preparation approaches available in the hope that other researches will used these StackOverflow alignments to help software engineers search for code and translate from English tasks to working code.

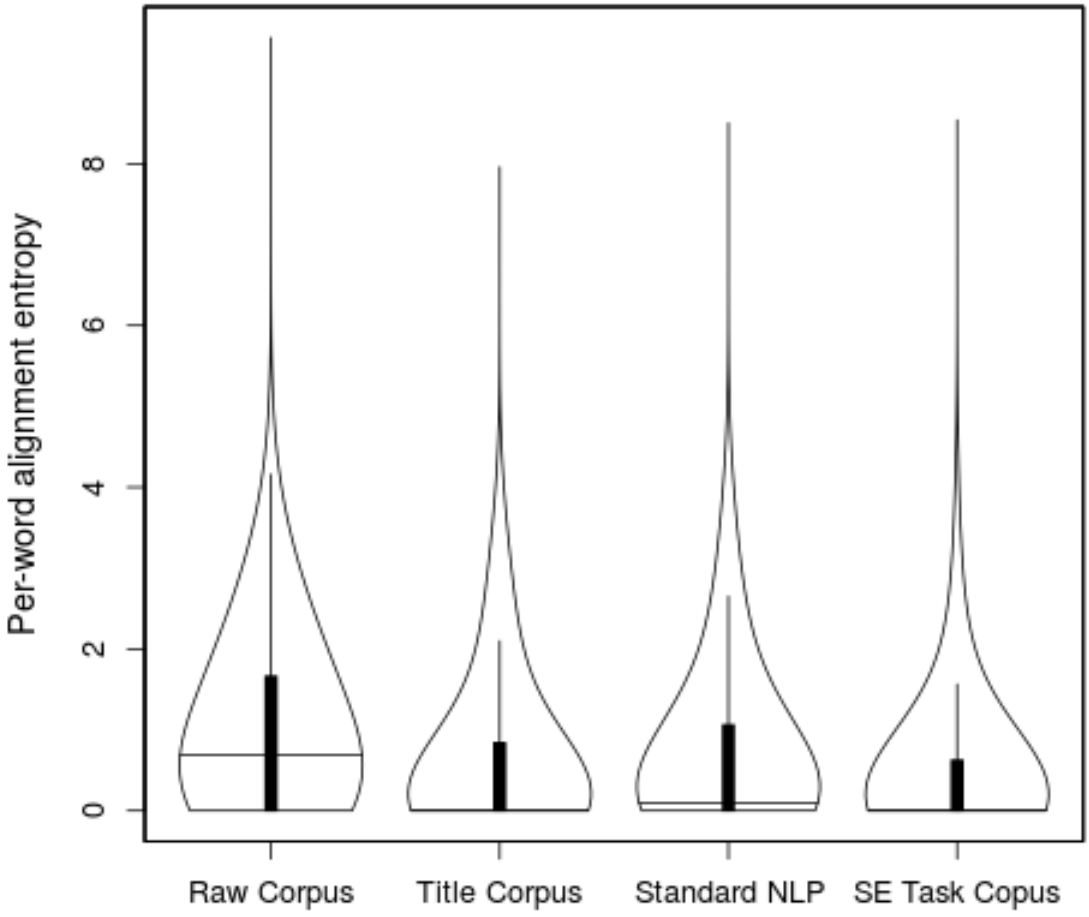


Figure 7: Per-word maximum likelihood alignment entropy

Chapter 5

Conclusions and Future Work

In this chapter we conclude the thesis by briefly summarizing our contributions and suggest potential future work based on the findings in the thesis.

Our findings from the first part of the thesis (Chapter 3) confirm previous work that code is repetitive and predictable. However, it is not as repetitive and predictable as Hindle *et al.* [26] suggested. We have found that the repetitive syntax of the program language makes software look artificially much more repetitive than English. For example, language specific SimpleSyntaxTokens account for 59% of the total Java tokens in our corpus. If the goal is to suggest tokens that are related to software engineering tasks, then SimpleSyntaxTokens will reduce the quality of suggested task tokens.

We hope our work will be used by researchers to select appropriate corpora with sufficient repetition. For example, we conducted a failed experiment to suggest patches based on past fixes using an n-gram language model. Had we had our current analysis there would have been little need to conduct the experiment as it would be obvious that internal class tokens and usages are too infrequent to be used successfully in a statistical model.

The success of API usage suggestions flows naturally from our findings. By reducing the vocabulary of unique tokens to API code elements and examining the usage of these APIs element across many programs there is sufficient repetition to make accurate suggestions.

Finally, our work shows that different representations of code have different degrees of repetition. Graph representations, such as Groums, allow for a higher degree of abstraction and the data and control flow allow for non-sequential relationships. Furthermore, the abstract nature of graphs allows for a more concise representation. Compared to n-grams, graphs reduce the number of noise tokens and increase a researcher's ability to make useful code suggestions and other types of predictions that are interesting to developers.

In the last part of the thesis (Chapter 4), we show the potential of StackOverflow as a bilingual machine translation corpus. We extract the English and code from each post to create aligned corpora. We show that the Raw StackOverflow posts contain substantial noise and do not lead to good MT models. In contrast, the three cleaned corpora: Standard NLP, including keyword extraction, using the StackOverflow title, and extracting SE tasks lead to low entropy alignments between English and code. This maximum likelihood entropy implies that each English word maps to specific code elements that can be used for translations between English and code.

Our work is empirical in demonstrating the steps necessary to prepare SE corpora. We make our data and corpus preparation approaches available in the hope that other researches will used these StackOverflow alignments to help software engineers search for code and translate from English tasks to working code.

Bibliography

[1] Replication package: scripts, n-gram, and graph data., 2017. Available at `https://github.com/CESEL/CodeEntropyReplication`.

[2] Y. Al-Onaizan, J. Curin, M. Jahr, K. Knight, J. Lafferty, D. Melamed, F.-J. Och, D. Purdy, N. A. Smith, and D. Yarowsky. Statistical machine translation. In *Final Report, JHU Summer Workshop*, volume 30, 1999.

[3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[4] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In B. Magnusson, editor, *SCM*, volume 1439 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 1998.

[5] S. Azad, P. C. Rigby, and L. Guerrouj. Generating api call rules from version history and stack overflow posts. *ACM Trans. Softw. Eng. Methodol.*, 25(4):29:1–29:22, Jan. 2017.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] R. Barzilay and K. R. McKeown. Extracting paraphrases from a parallel corpus. In *Proceedings of the 39th annual meeting on Association for Computational Linguistics*, pages 50–57. Association for Computational Linguistics, 2001.

[8] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 397–412, New York, NY, USA, 2006. ACM.

[9] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[10] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 20th International Conference on Software Engineering*, pages 782–792, 2012.

[11] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265, May 2015.

[12] B. A. Campbell and C. Treude. Nlp2code: Code snippet content assist via natural language tasks. *arXiv preprint arXiv:1701.05648*, 2017.

[13] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 252–261, New York, NY, USA, 2014. ACM.

[14] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. Cvssearch: Searching through source code using cvs comments. In *ICSM*, pages 364–, 2001.

[15] S. A. Chowdhury and A. Hindle. Mining stackoverflow to filter out off-topic irc discussion. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 422–425. IEEE Press, 2015.

[16] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, Oct 2007.

[17] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[18] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, Jan 1990.

[19] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. *SIGPLAN Not.*, 43(10):313–328, Oct. 2008.

[20] M.-C. De Marneffe and C. D. Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

[21] E. Duala-Ekoko and M. Robillard. Using structure-based recommendations to facilitate discoverability in apis. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 79–104. Springer Berlin Heidelberg, 2011.

[22] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[23] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification: Java se7 edition, 2013.

[24] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[25] B. Harris. Interlinear bitext. *Language Technology*, page 12, 1988.

[26] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, 2012.

[27] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, pages 1–34, 2017.

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[29] K. S. Jones. Natural language processing: a historical review. In *Current issues in computational linguistics: in honour of Don Walker*, pages 3–16. Springer, 1994.

[30] M. Jungiewicz and M. Łopuszyński. *Unsupervised Keyword Extraction from Polish Legal Texts*, pages 65–70. Springer International Publishing, Cham, 2014.

[31] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86, 2005.

[32] P. Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

[33] P. Koehn and K. Knight. Statistical machine translation, Nov. 24 2009. US Patent 7,624,005.

[34] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[35] S. Lahiri. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter*

of the Association for Computational Linguistics, pages 96–105, Gothenburg, Sweden, April 2014. Association for Computational Linguistics.

[36] Y. Lamba, M. Khattar, and A. Sureka. Pravaaha: Mining android applications for discovering api call usage patterns and trends. In *Proceedings of the 8th India Software Engineering Conference*, pages 10–19. ACM, 2015.

[37] E. Loper and S. Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[38] A. Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3):8, 2008.

[39] M. Lormans and A. Van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10–pp. IEEE, 2006.

[40] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, Oct. 2008.

[41] A. Ludwig. Recoder Technical Manual, 2001.

[42] E. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering (TSE)*, PP(99):To Appear, 2017.

[43] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[44] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, pages 24–, Washington, DC, USA, 1999. IEEE Computer Society.

[45] E. Millar, D. Shen, J. Liu, and C. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *Journal of Digital Information*, 1(5), 2006.

[46] G. Neubig. Neural machine translation. 2016.

[47] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[48] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE*, volume 9, pages 440–455. Springer, 2009.

[49] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen. T2api: Synthesizing api code usage templates from english texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017. ACM, 2016.

[50] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[51] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

[52] J. R. Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[53] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[54] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[55] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[56] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: synthesizing what i mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 357–367. ACM, 2016.

[57] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

[58] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.

[59] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: Characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 34–44, Piscataway, NJ, USA, 2015. IEEE Press.

[60] R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[61] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841. IEEE Press, 2013.

[62] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[63] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 390–401, New York, NY, USA, 2014. ACM.

[64] S. Rose, D. Engel, N. Cramer, and W. Cowley. Automatic keyword extraction from individual documents. *Text Mining: Applications and Theory*, pages 1–20, 2010.

[65] C. Rosen and E. Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.

[66] G. Salton and M. J. McGill. Readings in information retrieval. chapter The SMART and SIRE Experimental Retrieval Systems, pages 381–399. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[67] E. A. Santos and A. Hindle. Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 504–507, New York, NY, USA, 2016. ACM.

[68] C. E. Shannon. Prediction and entropy of printed english. *Bell Labs Technical Journal*, 30(1):50–64, 1951.

[69] H. Somers. Example-based machine translation. *Machine Translation*, 14(2):113–157, 1999.

[70] C. Treude, M. P. Robillard, and B. Dagenais. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering*, 41(6):565–581, 2015.

[71] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[72] S. Wang, D. Lo, and L. Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024. ACM, 2013.

[73] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 562–567. IEEE Press, 2013.

[74] H. Zhang. Exploring regularity in source code: Software science and zipf's law. In *2008 15th Working Conference on Reverse Engineering*, pages 101–110, Oct 2008.

[75] Y. F. Zhang, Q. F. Zhang, and R. H. Yu. Markov property of markov chains and its test. In *2010 International Conference on Machine Learning and Cybernetics*, volume 4, pages 1864–1867, July 2010.

[76] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.