Natural Software Revisited

Musfiqur Rahman, Dharani Palani, and Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University
Montréal, Canada
peter.rigby@concordia.ca

Accepted to ICSE'19. Camera-ready version will be available in February.

***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. When we remove SyntaxTokens 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, which contains a historically large range of styles and subjects (e.g. Saint Augustine to Oscar Wilde). We perform an additional comparison of technical StackOverflow English discussions with source code and find that this restricted English is similarly repetitive to code.

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 irrespective of domain. When we restrict our n-grams to those contained in the Java API we find that the entropy is significantly lower than the English corpora.

Previous works have focused on sequential sequences of 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 sequential representations of the same code. This suggests that future work should focus on *statistical* graph models that go beyond linear sequences of tokens.

Our anonymous replication package makes our scripts and data available to future researchers and reviewers [1].

***Index Terms*—Basic Science; Entropy; Language Models; Statistical Code Graphs; StackOverflow**

I. Introduction

Language modelling is a popular approach in the field of *Statistical Machine Translation (SMT)* [24] and *Natural Language Processing (NLP)* [22]. 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 [5, 9, 11, 20, 28, 36, 40, 51]. Although the term *naturalness* is vague, it has been expressed mathematically with statistical language models [20]. 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.* [9] 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, different programming tasks 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 SyntaxTokens. In contrast, if the goal is to recommend multi-element API usages, then SyntaxTokens will dilute predictions. For example, Hindle *et al.* [20] did not remove SyntaxTokens and in their autocompletion model they suggest a SyntaxToken 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 SyntaxTokens 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.* [20]. 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, Repetitive Syntax: how repetitive and predictable is code once we remove SyntaxTokens?

In NLP, it is standard practice to remove punctuation and stopwords [31, 46]. We examine the contribution of three types of SyntaxTokens 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.

TABLE I: Corpus size in tokens. The total of tokens must be constant across languages.

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%)

RQ4, Statistical 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 [8]. We extract the *Graph-based Object Usage Model (Groum)*[37] 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 II, we describe our data. In Sections III, IV, V, and VI, 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 VII, we discuss limitations of our work and threats to validity. In Section VIII, we position our work in the context of the literature. In Section IX, 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.

II. 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 [49]. 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 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 when we answer each 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 I.

[1]Top GitHub projects per language: https://github.com/trending/

English and StackOverflow text: Following Hindle *et al.* [20] we process the Gutenberg corpus. We use a subset of the Gutenberg corpus which includes over 3.4k English works [26]. The corpus represents a range of styles, topics, and time periods making Gutenberg a diverse corpus. To make a comparable, technical 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. 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 of English text and exclude 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, about the configuration of an IDE.

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

III. Replication

RQ1: How repetitive and predictable is source code?

We replicate the work of Hindle *et al.* [20] 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.

A. Theoretical background and methodology

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

1) 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, ... 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 a series of conditional probabilities [23, Ch 4]:

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

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 [52]. 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 [38]. 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. These conditional probabilities are calculated from the n-gram frequencies. We use these n-grams to determine the entropy of a language corpus including source code.

2) SelfCrossEntropy: Hindle *et al.*'s [20] calculate the average number of bits (entropy), required to predict the nth token of the n-grams in a document. They use the standard formula for cross-entropy, which is also the log-transformation of perplexity. 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.

3) Extracting n-grams: We replicate Hindle *et al.* [20] 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[2] 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 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[3] 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.* [20] 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.

B. Replication Result

How repetitive and predictable is software?

Figure 2a shows the replication of Hindle *et al.*'s [20] 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 results, we are confident that our dataset is large and diverse enough to test the "naturalness" hypothesis in new contexts.

While the pattern is the same, the values of SelfCrossEntropy are substantially different for each language. With Scala being much less repetitive than C#. The difference among languages leads us to conjecture that the syntax of the language artificially reduces its SelfCrossEntropy.

The pattern of decreasing SelfCrossEntropy across n-grams holds from Hindle *et al.*'s [20] work. However, the value SelfCrossEntropy differs among languages.

IV. Repetitive Syntax

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

Standard preprocessing steps in NLP involve the removal of stopwords and punctuation [31, 46]. 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 (unless one is creating a grammar checker). In our work, we examine the impact of three types of SyntaxTokens: separators such as brackets and semi-colons; keywords, such as `if` and `else`; and operators, such as plus and minus signs. Our replication package contains the full list of SyntaxTokens for each language [1]. By including syntax in their analysis, we suspect that Hindle *et al.* artificially inflate the naturalness of source code. Without these syntax tokens we hypothesize that raw source code will not be especially repetitive. In this section, we examine the impact of each type of SyntaxToken on the repetitiveness of code.

[2]ANTLR4 http://www.antlr.org/

[3]MITLM https://github.com/mitlm/mitlm

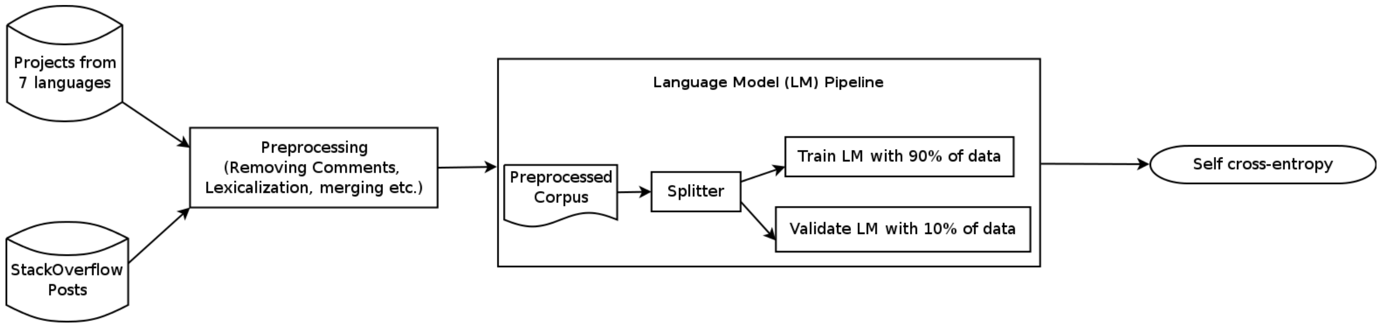
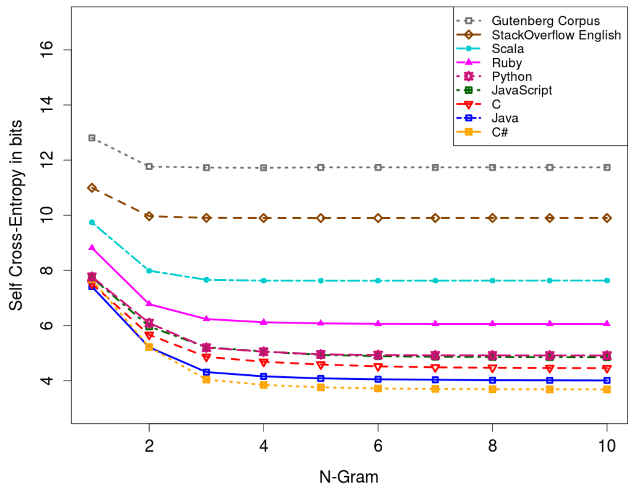
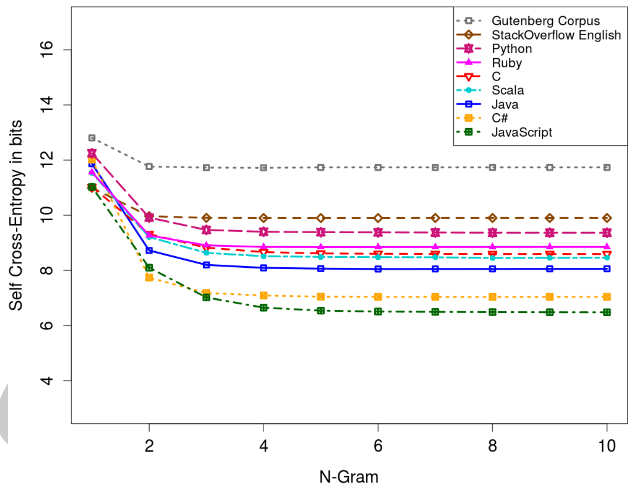

Projects from 7 languages
StackOverflow Posts
Preprocessing (Removing Comments, Lexicalization, merging etc.)
Language Model (LM) Pipeline
Preprocessed Corpus
Splitter
Train LM with 90% of data
Validate LM with 10% of data
Self cross-entropy

Fig. 1: Pipeline for experiments performed in this study



(a) SelfCrossEntropy with SyntaxTokens.



(b) SelfCrossEntropy *without* SyntaxTokens.

Fig. 2: The left figure is a direct replication of Hindle *et al.* [20]. The right figure does not include SyntaxTokens and the number of bits required to encode the programming languages is reduced and the variation among them is substantially smaller. Also the number of bits required to encode technical English on StackOverflow is similar to code.

A. Background and Methodology

For each programming language, we examined the language specification to identify the keywords, separators, and operators. We calculate the percentage of these SyntaxTokens in each programming language. Then we remove SyntaxTokens 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 SyntaxTokens?
2) What is the change in SelfCrossEntropy after removing SyntaxTokens?
3) How repetitive is code without SyntaxTokens compared to general English and to technical English on StackOverflow?[4]

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

B. Results and Discussion

What percentage of total tokens are SyntaxTokens? 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 II, we see that SyntaxTokens account for a high percentage of total tokens. Across the programming languages, JavaScript has the highest number of SyntaxTokens at 60% of total tokens, while the smallest percentage is 41% for Ruby. Separators account for the largest proportion of SyntaxTokens, between 23% and 47% of all tokens. The corresponding values for keywords are 5% and 11%, and for operators, 6% and 15%. **The main conclusion is that SyntaxTokens dominate the tokens in all programming languages and when included make code look artificially repetitive.** However, we are not suggesting that researchers or developers remove all SyntaxTokens as there may be cases where it is interesting to predict, for example,

TABLE II: Percentage of language syntax token. SyntaxTokens dominate the tokens in all programming languages and when included make code look artificially repetitive.

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%

the control structure through the "if" keywords. We do feel, as we discuss later, that statistical graph representations of code that include control flow may be better abstract representations than simply removing SyntaxTokens from a corpus.

What is the change in SelfCrossEntropy after removing SyntaxTokens?

When we remove the SyntaxTokens and recalculate the SelfCrossEntropy in Table III, we see a dramatic increase in SelfCrossEntropy and a corresponding decrease in repetitiveness. 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 SyntaxTokens.**

How repetitive is code without SyntaxTokens 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.* [20] 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 SyntaxTokens, 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 SyntaxTokens 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 SyntaxTokens 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.

How repetitive is StackOverflow English?
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 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 SyntaxTokens shows a 1.1 to 1.2 times as many bits or 10 to 20% more bits. This result is consistent across programming languages. **Technical discussion in English on StackOverflow have a similar degree of repetition to code**

C. *Concluding discussion on SyntaxTokens*

Hindle *et al.* were "worried" by the questions that we ask in this section [20]. 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 does not control for simple syntactical regularities in the Java language. In contrast, in our study we remove SyntaxTokens 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 SyntaxTokens, to reduce the noise they introduce and to make higher value recommendations.

SyntaxTokens make code look artificially repetitive and when removed the SelfCrossEntropy drops and the programming languages require a similar number of bits to encode as technical English on StackOverflow.

V. 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. For example reading from an input stream would have the following Java API sequence:
```
FileInputStream.FileInputStream()
FileInputStream.read()
FileInputStream.close()
```
We conjecture that sequences of API elements, *i.e.* API usages, should be more repetitive and predictable than general program code.

TABLE III: Percentage increase in SelfCrossEntropy after the removal of SyntaxTokens. For example, the SelfCrossEntropy for Java doubles after 3-grams indicating a substantial drop in its repetitive nature.

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

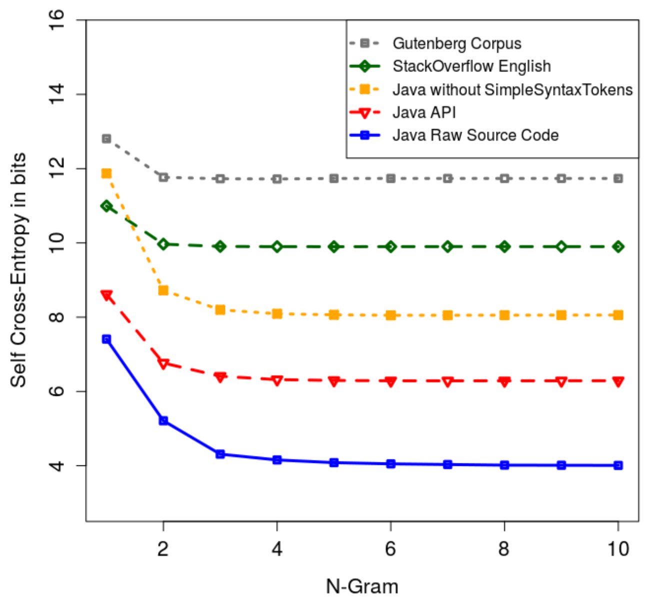


Fig. 3: Comparing the Java API SelfCrossEntropy with raw Java source code, Java source code without SyntaxTokens, and English. The use of the Java API is highly repetitive.

A. Background and Methodology

We extract the Java API elements from the Java Platform Library Standard Edition 7 Specification[17]. 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 (the full list and processed corpus can be found in our replication package [1]). For the Java corpus, we calculate the SelfCrossEntropy for the API usage of size 1 to 10-grams.

B. Results and Discussion for API Usages

Figure 3 compares the SelfCrossEntropy of n-gram API usages in Java to raw Java, Java without SyntaxTokens, StackOverflow English, and Gutenberg. We find that the SelfCrossEntropy of the Java API is less repetitive and predictable than the raw corpus which contains SyntaxTokens. This result derives from the high proportion of SyntaxToken tokens, *i.e.* 57% of tokens in Java are SyntaxTokens. Java that excludes SyntaxTokens 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 technical 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 SyntaxTokens 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 SyntaxTokens.

> We find that Java API usages are quite repetitive which quantifies the truth underlying the large and successful literature on sophisticated API recommendations (*e.g.,* [4, 7, 30, 36, 44]).

VI. Statistical Code Graphs

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

Most natural language models assume a sequential left-to-right reading order. 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 [8] which differs from natural language reading strategies [13]. For example, developers focus on method signatures [45] and following beacons [12] 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 has repetitions that cannot be identified with n-grams.

A. Background and Methodology

In order to determine how repetitive code graphs are, we need a statistical 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 be compiled 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 to create *statistical* graph models of code.

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) Statistics on structurally similar ASTs is not built in. Techniques [25][42][21] to identify structural similarities in the code using ASTs are computationally expensive[33].

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

In contrast, GROUMINER [32] was designed to extract GRaph-based Object Usage Models (GROUMS) and to calculate efficiently 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 [29].
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 [50].

B. Data

We use GROUMINER to capture the occurrence frequency of each GROUM in the Java projects used in the previous 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 [17]. 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 (see our replication package for the graphs and n-grams [1]). 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.

TABLE IV: 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

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 [5, 11, 28, 51]. The more left skewed the distribution the more repetitive and predictable the representation.

C. Results and Discussion for Statistical 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 IV shows the complete distribution from 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 more repetitive than sequential representations.**

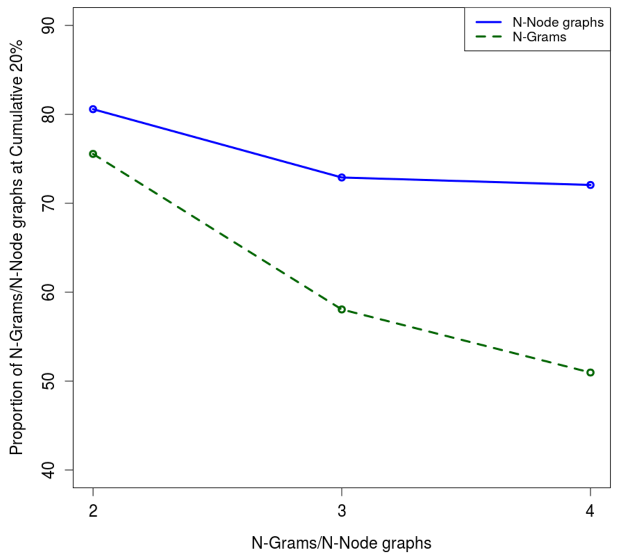


Fig. 4: The top 20% of n-grams and n-node graphs account for the y-axis% of the usages. For example, the top 20% of the n-node graphs account for 80.6% of all usages. Graphs are more repetitive than n-gram sequences.

D. *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 SyntaxTokens in this section, if they had been included the problem would be exacerbated because SyntaxTokens lie between all 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 1 to 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 2 the code iterates through a hashmap of tracked client sessions and in Listing 1 the code iterates through a hashmap of throughput reports.

Below we use the listings to show the important differences

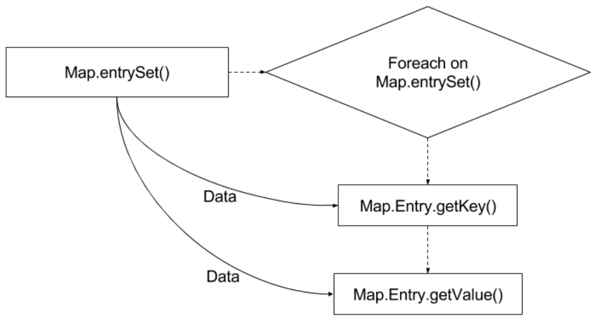


Fig. 5: A GROUM representing iteration through a HashMap. This graph is statistically common and is an abstraction of the code in Listings 1 through 4.

between the GROUM and n-gram models.

Abstraction: From examining the listings, it is clear that no sequential model would consider these code fragments as identical. There are many internal classes and SyntaxTokens 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 SyntaxTokens, for the respective listings we need sequences with 34, 32, 30, and 38 tokens to represent the code in the listing. Without SyntaxTokens the corresponding number of tokens is smaller but still quite large at 14, 15, 13, and 15 tokens, respectively.

Graphs are also a more realistic representation of code than sequential n-grams because compilers and humans do not process code sequentially. Graphs are more appropriate for statistical code recommendation because they can recommend non-sequential relationships that cannot be represented in a sequential model.

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 sequences of tokens.

Listing 1: TransactionThroughputChecker.java

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

Listing 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: 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 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;
}
```

VII. Limitations and Validity

Limitations of graphs: To extract an AST from a large number of projects we used Recoder [29]. Recoder, like the PPA tool [14], 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 [14, 29, 43].

A second limitation is the computational expense of identifying isomorphic graphs using GROUMINER [33]. 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.

Model Limitations: There are many natural language models that deal with sequences of tokens: n-grams, skip-grams, RNN, LSTM [16, 19, 49]. The goal of the basic research in this paper is to understand the distribution of tokens in code corpora and to understand if there is enough repetition to make interesting statistical predictions. The model is less relevant to this work than the repetition in the corpus as neural networks depend upon this repetition as much as linear models. Since models designed for natural language assume a sequential sequence of tokens, they are not appropriate for code. We need to modify the underlying unit for these models and modify them for statistically-based abstract graph representations.

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.

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 [20]. Ideally, we would calculate all possible combinations of the next token, however, as Shannon [48] 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.

VIII. Related Work

Research into language entropy.

Basic research into understanding redundancy and measuring entropy in languages has a long history. Shannon [48] developed statistical measures of entropy for the English language. Gabel and Su [16] noted high levels of redundancy in code. Hindle *et al.* [20] continued this work demonstrating that software is highly repetitive and predictable. Recent works have replicated these software findings on a giga-token corpus [2], looked at the entropy in local code contexts [49], and applied neural network models [19]. Others have examined repetition at the line level [41] and in other domains such as Android Apps[4, 27]. 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 (especially without separators). 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 for finding 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.* [9] find that syntax errors can be identified using n-gram language models. Ray *et al.* [40] identified bugs and bug fixes in code because buggy code is less natural and has a higher entropy. Santos and Hindle [47] 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 because these former types are much more repetitive.

Research into recommenders and autocompletions.

Modern IDEs contain an autocompletion feature that usually use the structure of the language to make suggestions. Researchers working on code suggestion have long known intuitively that code is repetitive. For example, textual similarity of program code [3], commit messages [10], and API usage patterns [30] have been exploited to guide developers during their engineering activities. Building on this work, Zimmermann *et al.* [53] 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.* [4] has extended this work to make change rule predictions from a large community of similar Apps and the code discussed on StackOverflow.

Advanced recommendation techniques have used the history of applications and the repetitive nature of programming to recommend code elements to developers. Robbes and Lanza [44] filtered the suggestions made by code completion algorithms, for example, based on where the developer had been working in the past and the changes he or she had made. Bruch *et al.* [6] recommend appropriate method calls for a variable based on an existing code base that makes similar calls to a library. Buse and Weimer [7] automatically generate code snippets from a large corpus of applications that use an API. Duala-Ekoko and Robillard [15] use structural relationships between API elements, such as the method responsible for creating a class, to recommend related elements to developers. Works by Nguyen *et al.* [36] use statistical language models to recommend code accurately. Nguyen and Nguyen [32] expanded this work to graphs in order to create recommendations that are syntactically valid. Much of this work focuses on recommending API elements. Our work suggests that API usages are substantially more repetitive and predictable than general code, which explains the success of API recommendation approaches. Furthermore, we show why graphs are a more appropriate representation of code, and we hope this will encourage future researchers to focus on *statistical* graph abstractions instead of sequential tokens.

Research on 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 [39] uses a corpus of queries from Bing to align code and English and generates sequences of API usages. DeepAPI [18] uses recurrent neural networks to translate aligned source code comments with code to translate longer sequences of API calls. T2API [34, 35] uses alignments between English and code on StackOverflow to generate a set of API calls. These calls are then rearranged based on their usage likelihood in 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.

IX. Conclusion

Our findings confirm previous work that code is repetitive and predictable. However, it is not as repetitive and predictable as Hindle *et al.* [20] 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 SyntaxTokens 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 SyntaxTokens, 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 any 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 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. Most abstract representations of code are based on rule oriented ASTs. Although this work is basic science and does not involve developing new tools and techniques, we suggest that future work should focus on new code representations that are tailored to *statistical* code suggestion allowing for complex and useful recommendations.

References

[1] Replication package: scripts, n-grams, and graph data. https://www.dropbox.com/s/o5016gelg8tx5yx/README.md?dl=0, 2018.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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*, FSE 2016, pages 631–642. ACM, 2016.

[19] V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773. ACM, 2017.

[20] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[21] 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.

[22] 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.

[23] D. Jurafsky and J. H. Martin. *Speech and language processing*, volume 3. Pearson London, 2014.

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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Dharani, Palani ano Karanfil, and T. N. Nguyen. Statistical Translation of English Texts to API Code Templates. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, ICSME '18. IEEE Computer Society, 2018.

[35] 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 (Formal Demostration Track)*, FSE 2016, pages 1013–1017. ACM, 2016.

[36] 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.

[37] 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.

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

[39] 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*, ICSE '16, pages 357–367, New York, NY, USA, 2016. ACM.

[40] 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.

[41] 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.

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

[43] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841, 2013.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] J. R. Ullmann. An algorithm for subgraph isomorphism.

Journal of the ACM (JACM), 23(1):31–42, 1976.

[51] 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.

[52] 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.

[53] 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.