THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Mining source code repositories at massive scale using language modeling

Citation for published version:
Allamanis, M & Sutton, C 2013, Mining source code repositories at massive scale using language modeling. in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on.* Institute of Electrical and Electronics Engineers (IEEE), pp. 207-216. https://doi.org/10.1109/MSR.2013.6624029

Digital Object Identifier (DOI):
10.1109/MSR.2013.6624029

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 19. Apr. 2024

Allamanis, M., & Sutton, C. (2013). Mining source code repositories at massive scale using language modeling. In Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. (pp. 207-216). IEEE Computer Society. doi: 10.1109/MSR.2013.6624029

Mining Source Code Repositories at Massive Scale using Language Modeling

Miltiadis Allamanis, Charles Sutton
School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, UK
Email: m.allamanis@ed.ac.uk, csutton@inf.ed.ac.uk

***Abstract*—The tens of thousands of high-quality open source software projects on the Internet raise the exciting possibility of studying software development by finding patterns across truly large source code repositories. This could enable new tools for developing code, encouraging reuse, and navigating large projects. In this paper, we build the first giga-token probabilistic language model of source code, based on 352 million lines of Java. This is 100 times the scale of the pioneering work by Hindle et al. The giga-token model is significantly better at the code suggestion task than previous models. More broadly, our approach provides a new "lens" for analyzing software projects, enabling new complexity metrics based on statistical analysis of large corpora. We call these metrics *data-driven complexity metrics*. We propose new metrics that measure the complexity of a code module and the topical centrality of a module to a software project. In particular, it is possible to distinguish reusable utility classes from classes that are part of a program's core logic based solely on general information theoretic criteria.**

I. Introduction

An important aspect of mining software repositories is the analysis of source code itself. There are several billion lines of open source code online, much of which is of professional quality. This raises an exciting possibility: If we can find patterns that recur throughout the source code of many different projects, then it is likely that these patterns encapsulate knowledge about good software engineering practice. This could enable a large variety of *data-driven software engineering tools*, for example, tools that recommend which classes are most likely to be reusable, that prioritize potential bugs that have been identified based on static analysis results, and that aid program navigation and visualization.

A powerful set of tools for finding patterns in large corpora of text is provided by statistical machine learning and natural language processing. Recently, Hindle et al. [1] presented pioneering work in learning *language models* over source code, that represent broad statistical characteristics of coding style. Language models (LMs) are simply probability distributions over strings. However, the language models from that work were always trained on projects within a single domain, with a maximum size of 135 software projects of code in their training sets. Previous experience in natural language processing has shown that n-gram models are "data hungry", that is, adding more data almost always improves their performance. This raises the question of how much the performance of an LM would improve with more data, and in particular, whether it is possible to build a *cross-domain LM*, that is, a single LM that is effective across different project domains.

In this paper, we present a new curated corpus of 14,807 open source Java projects from GitHub, comprising over 350 million lines of code (LOC) (Section III). We use this new resource to explore coding practice across projects. In particular, we study the extent to which programming language text is *productive*, constantly introducing original identifier names that have never appeared in other projects (Section IV). For example, although the names `i` or `str` are common across projects, others like `requiredSnaphotScheduling` are specific to a single project; we call these *original identifiers*. We examine the rate at which original identifiers occur, finding that most of them are variable names, with an average project introducing 56 original identifiers per kLOC.

Second, we train a n-gram language model on the GitHub Java corpus (Section V). This is the first *giga-token* LM over source code, i.e., that is trained on over *one billion* tokens. We find that the giga-token model is much better at capturing the statistical properties of code than smaller scale models. We also examine what aspects of source code are most difficult to model. Interestingly, we find that given enough data the n-gram learns all that it can about the syntactic structure of code, and that further gains in performance are due to learning more about patterns of identifier usage.

The giga-token model enables a new set of tools for analyzing code. First, we examine which tokens are the most predictable according to the LM, e.g., for identifying code regions that are important to the program logic. We find that *method names are more predictable than type and variable names*, perhaps because API calls are easy to predict. Extending this insight, we combine the probabilistic model with concepts from information theory to introduce a number of new metrics for code complexity, which we call *data-driven complexity metrics*. Data-driven metrics, unlike traditional metrics, are fine-tuned by statistical analysis of large source code corpora. We propose using the n-gram log probability (NGLP) as a complexity metric, showing that it trades off between simpler metrics such as LOC and cyclomatic complexity. Additionally, we introduce a new metric that measures how *domain specific* a source file is, which we call the *identifier information metric* (IIM). *This has direct applications to code reuse*, because code that is less domain specific is more likely to be reused.

Finally, we present a detailed case study applying these new data-driven complexity metrics to the popular rhino

JavaScript compiler. On rhino the identifier information metric is successful at differentiating utility classes from those that implement core logic, despite *never having seen* any code from the project in its training set.

II. Language Models for Programming Languages

Source code has two related but competing purposes. First, it is necessary to provide unambiguous executable instructions. But it also acts a means of communication among programmers. For this reason, it is reasonable to wonder if statistical techniques that have proven successful for analyzing natural language will also be useful for analyzing source code.

In this paper, we use *language models* (LM), which are probability distributions over strings. An LM is *trained* on a corpus of strings from the language, with the goal of assigning high probability to strings that a human user of a language is likely to write, and low probability to strings that are awkward or unnatural. For n-gram language models (Section II-A), training is *programming language independent* since the learning algorithms do not change when we develop a model for a new programming language.

Because they are grounded in probability theory, LMs afford the construction of a variety of tools with exciting potential uses throughout software engineering. First, LMs are naturally suited to predicting new tokens in a file, such as would be used by an autocompletion feature in an IDE, by computing the model's conditional probability over the next token. Second, LMs provide a metric for assessing whether source code has been written in a natural, idiomatic style, because code that is more natural is expected to have higher probability under the model. Finally, we can use tools from information theory to measure how much information (in the Shannon sense) each token provides about a source file (Section II-B).

LMs have seen wide use in natural language processing, especially in machine translation and speech recognition [2]. In those areas, they are used for ranking candidate sentences, such as candidate translations of a foreign language sentence, based on how natural they are in the target language. To our knowledge, Hindle et al [1] were the first to apply language models to source code.

A. n-Gram Language Models

Language models are generative probabilistic models. By the term *generative* we mean that they define a probability distribution from which we can sample and generate new code. Given a string of tokens $t_0, t_1 \dots t_M$, an LM assigns a probability $P(t_0, t_1 \dots t_M)$ over all possible token sequences. As there is an infinite number of possible strings, obviously we cannot store a probability value for every one. Instead we need simplifying assumptions to make the modelling tractable. One such simplification, which has proven to be effective is practice, forms the basis of n-gram models.

The n-gram language model makes the assumption that the next token can be predicted using only the previous $n-1$ tokens. In mathematical terms, the probability of a token t_m conditioned on all of the previous tokens $t_0 \dots t_M$ is a function only of the previous $n-1$ tokens. This assumption implies that we can write the joint probability of an entire string as

$$P(t_0 \dots t_M) = \prod_{m=0}^{M} P(t_m | t_{m-1} \dots t_{m-n+1}). \quad (1)$$

To use this equation we need to know the conditional probabilities $P(t_m | t_{m-1} \dots t_{m-n+1})$ for each possible n-gram. This is a table of V^n numbers, where V is the number of unique words in the language. These are the *parameters* of the model that we will learn from a *training corpus* of text that is written in the language of interest. The simplest way to estimate these parameters is to use the empirical frequency of the n-gram in the training corpus, that is,

$$P(t_m | t_{m-1} \dots t_{m-n+1}) = \frac{c(t_m \dots t_{m-n+1})}{c(t_{m-1} \dots t_{m-n+1})}, \quad (2)$$

where $c(\cdot)$ is the count of the given n-gram in the training corpus. However, in practice this simple estimator does not work well, because it assumes that n-grams that do not occur in the training data have zero probability. This is unreasonable even if the training corpus is massive, because languages constantly use sentences that have never been uttered before. Instead, n-gram models are trained using *smoothing* methods that reserve a small amount of probability to n-grams that have never occurred before. For detailed information on smoothing methods, see Chen and Goodman [3].

B. Information Theory & Language

Since n-grams assign probabilities to sequences of tokens we can interpret code both from a probabilistic and an information theoretic point of view. Given a probability distribution $P(\cdot)$

$$Q(t_0, t_1 \dots t_M) = -\log_2(P(t_0, t_1 \dots t_M)) \quad (3)$$

is the *log probability* of the token sequence $t_0 \dots t_M$. We refer to this measure as the *n-gram log probability* (NGLP). Intuitively this is a measure of "surprise" of a (fictional) receiver when she receives the token sequence. For example, in Java, tokens such as `;` or { can be predicted easily, and thus are not as surprising as variable names.

The standard way to evaluate a language model is to collect a test corpus $t'_0 \dots t'_M$ that was not used to train the model, and to measure how surprised the model is by the new corpus. The average amount of surprise per token is given by

$$H(t'_0 \dots t'_M) = \frac{Q}{M} = -\frac{1}{M} \sum_{m=0}^{M} \log_2 P(t'_m | t'_{m-1} \dots t'_{m-n+1}), \quad (4)$$

and is called the *cross entropy*. Cross entropy measures how well a language model P predicts each of the tokens in turn, while *perplexity*, equivalent to $2^{H(\cdot)}$, shows the average number of alternative tokens at each position. A low cross entropy means that the string is easily predictable using P. If the model predicts every token t'_m perfectly, i.e., with a probability of 1, then the cross entropy is 0. A LM that assigns low cross entropy to a test sequence is better, since

TABLE I: Top Projects by Number of Forks in Corpus

Name	# forks	Description
hudson	438	Continuous Integration - Software Engineering
grails-core	211	Web Application Framework
Spout	204	Game - Minicraft client
voldemort	182	Distributed key-value storage system
hector	166	High level Client for Cassandra (key-value store)

TABLE II: Top Projects by LOC in Corpus

Name	kLOC	Description
openjdk-fontfix	4367	OpenJDK fork, fixing fonts
liferay-portal	4034	Enterprise Web Platform
intellij-community	3293	IDE for Java
xtext	3110	Eclipse for DSLs
platform	3084	WSO_2 entrerprise middleware platform

it is on average more confident when predicting each token. Cross entropy also has a natural interpretation. It measures the average number of bits needed to transfer the string $t'_0 \dots t'_M$ using an optimal code derived from the distribution P. For this reason, cross entropy is measured in "bits".

III. The GitHub Java Corpus

To our knowledge, research on software repositories, with the exception of Gabel and Su [4] and Gruska et al. [5], has focused on small curated corpora of projects. In this section we describe a new corpus that contains thousands of projects. Large corpora present unique challenges. First at that scale it is impractical to compile, or fully resolve code information (e.g. fully resolved AST) since most of the projects depend on a large and potentially unknown set of libraries and configurations. Therefore, there is a need for methods that can perform statistical analysis from large amounts of code while being generic or even programming language independent. To achieve this we turn to probablistic language models (Section II). A second challenge with large corpora is the need to automatically exclude low-quality projects. GitHub's[1] social fork system allows us to approximate that because low quality projects are more rarely forked.

Although Java is the 3rd most popular language in GitHub it is only the 19th when considering the average forks per project (1.5 forks per project). Among the top languages and across the projects that have been forked at least once, Java comes 23rd with an average of 20 forks per project. This is partially due to the fact that some projects, such as Eclipse, split their codebase into smaller projects that are more rarely forked. It may also be attributed to Java being more difficult compared to other popular languages, such as JavaScript.

We processed GitHub's event stream, provided by the GitHub Archive[2] and filtered individual projects that were forked[3] at least once. For each of these projects we collected the URL, the project's name and its language. Filtering by forks aims to create a quality corpus: Since we filter away the majority of projects that have a below-average "reputation", we hope to obtain code of better quality. This criterion is not clear-cut and we could have chosen another method (e.g. GitHub's star system), but we found this preprocessing sufficient.

We then downloaded (`clone` in Git's terms) the full repositories for all remaining Java projects and removed any duplicates: We found about 1,000 projects that shared common commit SHAs indicating that they were most likely forks of the same project (but not declared on GitHub) and we manually picked those projects that seemed to be the original source. Project duplicates need to be removed to avoid a "leak" of data from the test to the train projects and also to create a more representative corpus of repositories. Many of the duplicate projects were Android subsystems and that would have skewed our analysis towards those projects.

TABLE III: Java Corpus Characteristics

	Train	Test
Number of Projects	10,968	3,817
LOC	264,225,189	88,087,507
Tokens	1,116,195,158	385,419,678

We ended up with 14,807 projects across a wide variety of domains amounting to 352,312,696 lines of code in 2,130,264 files. Tables I and II present some projects in our corpus. Only files with the `.java` extension were considered. We split the repository 75%-25% based on the lines of code assigning projects into either the training or test set. The characteristics of the corpus are shown in Table III and the corpus can be found online[4]. We tokenized and retrieved the AST of the code, using Eclipse JDT. Any tokens representing comments were removed. From the ASTs we are able to determine if identifiers represent variables, methods or types.

A potential limitation of the resulting corpus is that it focuses on open source projects, which limits the type of domains included. For example, it is more probable to find projects about programming tools than banking applications. Second, since GitHub has gained popularity relative recently some mature and low-activity projects may not be present.

IV. Properties of a Large Source Code Corpus

In this section we explore coding practice at scale using the GitHub Java corpus (Section III). By examining the training corpus tokens we observe (Figure 1) Zipf's law, usually found in natural languages. Zipf's law states that the frequency of a token is inversely proportional to its rank in a hypothetical (sorted) frequency table.

We are now interested in studying how original identifiers are introduced across projects. Some identifiers, such as `i` or `str` are far more common, but projects also introduce original identifiers such as `requiredSnapshotScheduling`. Source code identifiers, the most diverse type of tokens in code, have strong semantic connotations that allow an intuitive

[1] http://www.github.com

[2] http://www.githubarchive.org

[3] In GitHub's terminology a fork is when a project is duplicated from its source to allow variations that may later be merged back to the original project

[4] http://groups.inf.ed.ac.uk/cup/javaGithub/

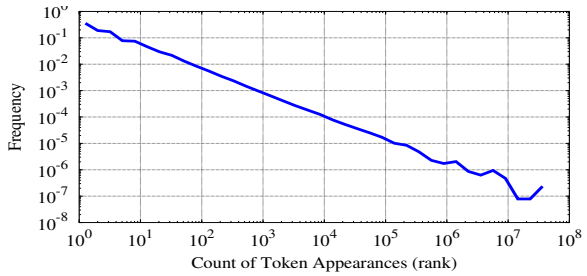


Fig. 1: Zipf's law for tokens (log-log axis) in the train corpus. The x-axis shows the number of times a single token appears. The y-axis shows the number (frequency) of tokens that have a specific count. The train corpus contains 12,737,433 distinct tokens. The slope is $a = -0.853$.

TABLE IV: Original Identifiers Introduced per kLoC for All Test Projects (statistics across projects)

	Method	Type	Variable	Total
Mean	20.98	14.49	21.02	56.49
Median	17.93	12.07	15.13	51.72
St.Dev.	17.93	13.11	22.16	38.80

understanding of the code [6]. We study three types of identifiers:

- variable and class field name identifiers;
- type name identifiers such as class and interface names;
- method (function) name identifiers.

In this analysis, we ignore literals, identifiers in package declarations and Java annotations for simplicity. The three types of identifiers studied represent different aspects of code.

First, we look at the test corpus and compute the number of original identifiers (not seen in the training set) per kLOC for each project (Table IV). We observe that type identifiers are more rarely introduced compared to method or variable identifiers, which are introduced at about the same rate. Interestingly, variable identifiers have a larger standard deviation, indicating that the actual rate varies more across projects compared to other identifiers. As we will discuss later, this is probably due to the multiple domains in our corpus. Surprisingly, the results indicate that although test projects are not related to the train projects they tend to use the same identifier "vocabulary".

Figure 2 shows the rate at which original identifiers are introduced in the training corpus as we increase the amount of code seen. We randomize the order of the source code files and average over a set of five random sequences. According to the data the identifier introduction rate decreases as we observe more lines of code. According to Figure 2, initially, variable and method identifiers are introduced at about the same proportion with type identifiers prevailing. However, as we observe just one order of magnitude more code, variable and method identifiers prevail. This transitional effect can be interpreted as the introduction of the basic type vocabulary commonly reused across projects before reaching a steady state. Finally, after having observed about a million lines

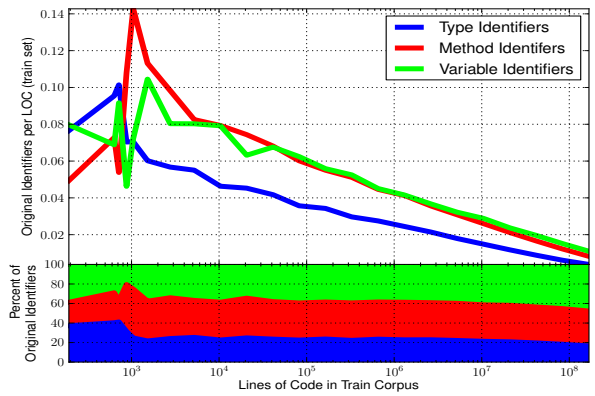


Fig. 2: The median number of original identifiers introduced per line of code for each identifier type as more data is observed (i) as an absolute value (top) and (ii) as a percentage of the total identifiers introduced (bottom)

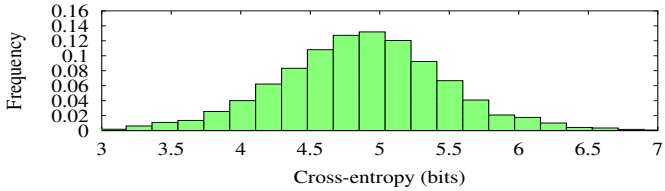


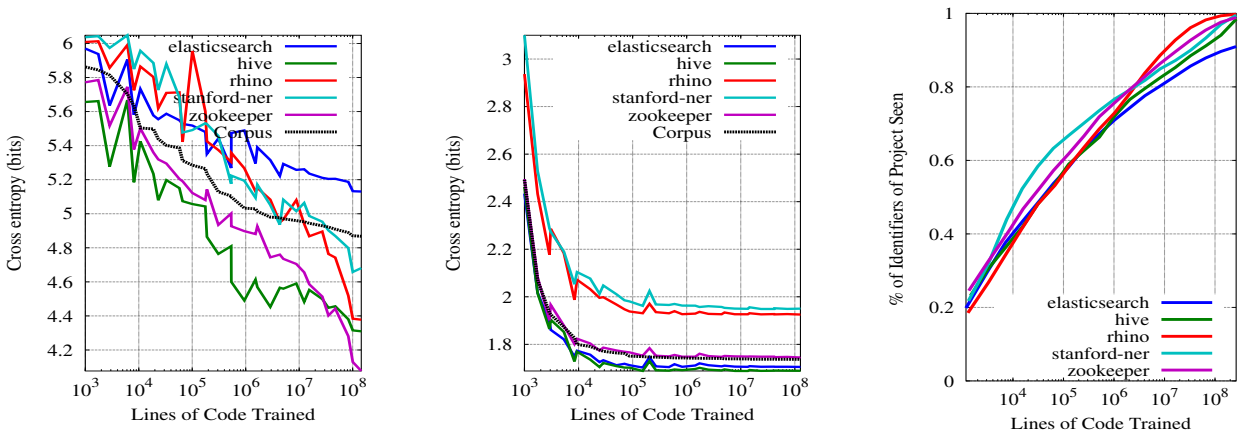
Fig. 3: Distribution of project cross entropy (bits) in 3-gram for test projects. $\mu = 4.86$, $\sigma = 0.64$ bits.

of code variable identifiers become the predominant type of identifiers accounting for almost 50% of the original identifiers introduced. This effect can be attributed to the *reuse* of classes and interfaces and to the fact that variable names are those that bridge the class (type name) concepts to the domain specific objects that a piece of code instantiates.

V. Giga-Scale Language Models of Code

Now, we describe our model, a cross domain trigram model that is trained on over a billion tokens of code. We train trigram model on the training portion of our Java corpus and evaluate it on the testing portion. As the corpus is divided at a project level, we are evaluating the model's ability to predict across projects. The giga-token model achieves a cross entropy of 4.9 bits on average across the test corpus (Figure 3). Comparing to the cross entropies of 6 bits reported by Hindle et al. [1], the new model decreases the perplexity by an order of magnitude. This result shows that cross-project prediction is possible at this scale, overcoming problems with smaller scale corpora.

To better understand how n-gram models learn from source code, in this section we study the learning process of these models as more data is observed. We plot the learning curve (Figure 4a) of five sample test projects as a function of the number of lines of code used to train the LM. The cross entropy of the sample projects decreases in a log-linear fashion as more projects are used to train the model, as one would expect because there is a diminishing returns effect to enlarging the training corpus.



(a) Full Model (b) Collapsed Model (c) Identifier Coverage

Fig. 4: Learning curves and identifier coverage for 3-gram LM using 5 sample test projects and for the whole test corpus.

To obtain insight into the model, we can differentiate between what it learns about the syntactic structure of code as opposed to identifier usage: we define a new model, which we call the *collapsed model*, based on a small change to the tokenization process. Whenever the tokenizer reads an identifier or a literal it outputs a special `IDENTIFIER` or `LITERAL` token instead of the actual token. The rest of the model remains the same. Now our tokenized corpus contains only tokens such as keywords (e.g. `if`, `for`, `throws`, `and`) and symbols (e.g. {, +) that represent the structural side of code. This means that the model is expected to predict only that an identifier will be used, but not which one. Using this tokenization, we now plot the learning curves for five sample test projects as the n-gram learns from more training data (Figure 4b). The cross entropy achieved is much lower than the one achieved by the full model. This signifies the importance of the identifiers when learning and understanding code: Identifiers and literals are mostly responsible for the "*unpredictability*" of the code.

Interestingly, the learning curve in Figure 4b also shows that the model has learned all that it can about syntax. After only about 100,000 lines of code in the training set, the performance of the collapsed model levels off, indicating that the n-gram model cannot learn any more information about the structure of the code. This contrasts to the fact that the original n-gram model (Figure 4a) continues to improve its performance even after having seen millions of lines of code, suggesting that the continuing decrease in cross entropy in the full model is due to enhanced learning of identifiers and literals.

Additional evidence is given by our study in Section IV. In that section, we saw that original identifiers are constantly introduced but at a constantly smaller rate. To verify this result, Figure 4c plots the percentage (coverage) of identifiers of each of the sample projects as we increase the size of the training set, averaged over ten random sequences of training files. When trained on the full training corpus, the n-gram model sees about 95% of the identifiers present in the sample test projects. This provides additional support for the explanation that the n-gram model (Figure 4a) continues to become better as it is trained on more data because it becomes able to better predict the identifiers.

A. Predicting Identifiers

Figure 5 shows a sample elasticsearch snippet tagged with the probabilities assigned by two n-gram models: the n-gram model using all tokens (left) and the collapsed version. According to Figure 5 the collapsed n-gram model is much more confident at predicting the type of the tokens that are following, especially when the type of the token is an identifier. However, when we try to predict the actual identifiers (Fig. 5 left) the task is much harder. For example, it is hard for the model to predict that a `ForkJoinTask` object is about to be created (line 4; $p = 5.5 \cdot 10^{-7}$) or that `doSubmit` will be called (line 9; $p = 1.4 \cdot 10^{-7}$) although it is about 45.5% probable that an identifier will follow.

Nevertheless, the model has managed to capture some interesting patterns with identifiers. For example, the n-gram assigns a reasonably high probability ($p = 0.028$) to `NullPointerException` when the previous tokens are `throw new`. It also considers almost definite ($p = 0.86$) that when an identifier named `task` is tested for equality, the second predicate will be `null` (line 2).

B. Learnability of Identifiers

In the previous sections our results indicate that learning to predict code from a n-gram perspective is difficult mainly because of the identifiers. It seems that identifiers are responsible for the greatest part of code cross entropy while they are those that allow the expressiveness in programming languages. Using the findings in Section IV, we now examine how different types of identifiers are learned.

Figure 6 depicts the learning curves for two sample test projects we previously used. For each project we plot the learning curve of a n-gram model trained collapsing all identifiers except from a specific type. Figure 6c shows the number of extra bits per token required to predict each identifier type letting us infer the difficulty of predicting each type of identifier. As expected the fully collapsed token sequences are

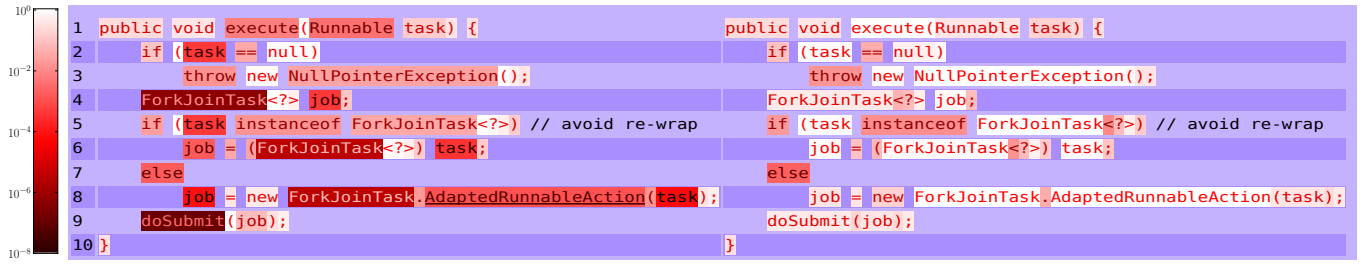
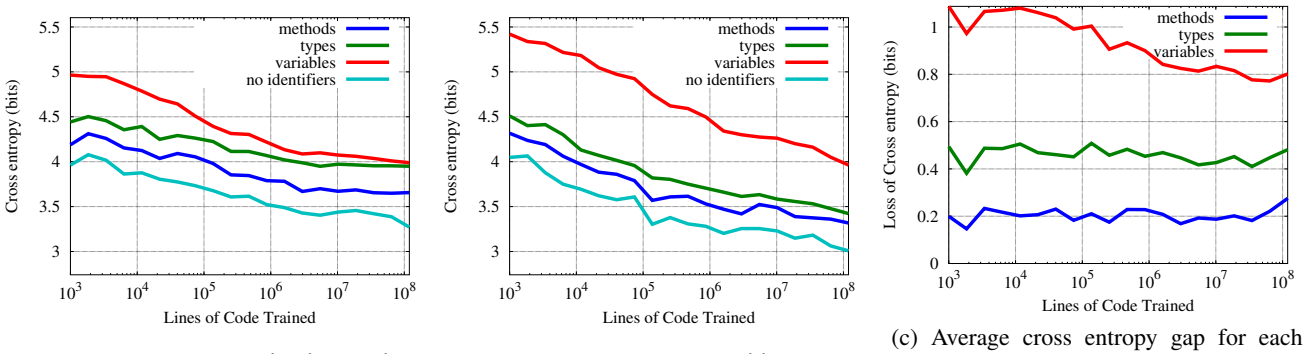


Fig. 5: Token heatmap of a `ForkJoinPool` snippet on the elasticsearch project. The background color of each token represents the probability (color scale—left) assigned by two 3-gram models. The left (right) snippet shows the token probabilities of the full (collapsed) 3-gram. The underlined tokens are those that the LM returns the probability for a previously unseen (UNK) token.



(a) Token learning in elasticsearch

(b) Token learning in rhino

(c) Average cross entropy gap for each token type on five sample projects

Fig. 6: Learning Curve for n-gram per token type cross entropy

the easiest to predict and have consistently the lowest cross entropy across projects.

The collapsed n-gram using method name identifiers has the second best cross entropy. Predicting method names adds on average only 0.2 bits per token. This means that *API calls are significantly more predictable compared to types and variable names*. Predicting type identifiers comes second in difficulty after method names. Thus type identifiers are harder to predict in code increasing the cross entropy on average by about 0.5 bits per token.

Finally, variable name identifiers increase the cross entropy by about 0.9 bits, reducing the token probabilities by almost an order of magnitude compared to the full collapsed n-gram LM. Again this shows the significance of the variable names to the domain of each project and the (obvious) importance they have when appearing in a sequence of code tokens.

The most surprising result (Figure 6c) is the average number of extra bits of cross entropy required for each type of token: the model performance on methods and types does not significantly improve, irrespectively of the amount of lines of code we train it. In contrast, the n-gram model gradually becomes better at predicting variable name identifiers, although they are harder to learn. This is surprising given our prior knowledge for variable names: *Although in general they are harder to predict, they are more easily learned* in large corpora. This implies that from the large number of distinct variable names few of them are frequently reused and are more context dependent. Looking back at Figure 5 we can now better explain why in line 1 the variable name `task` is fairly easy to predict compared to the `Runnable` type name. *Thus when creating code completion systems variable name prediction will only improve as more data is used.*

VI. Code Analysis using Giga-Scale Models

In this section, we apply the giga-token language model to gain new insights into software projects. First, we consider whether cross entropy could be used as a code complexity metric (Section VI-A), as a complement to existing metrics such as cyclomatic complexity. Then we examine how cross entropy varies across files (Section VI-B) and (Section VI-C) across projects. We find that on average interfaces appear simpler than classes that implement core logic, and projects that most define APIs or provide examples have lower cross entropy than research code. Finally, we define a new metric, called the identifier information metric, that attempts to measure how domain specific a file is based on how unpredictable its identifiers are to a language model (Section VI-D3). We find that the identifier information metric can successfully differentiate utility classes from core logic, even on projects that do not occur in the LM's training set.

A. n-gram Log Probability as a Data-driven Code Complexity Metric

Code complexity metrics quantify how understandable a piece of code is and are useful in software quality assurance. A multitude of metrics have been proposed [7], including McCabe's cyclomatic complexity (CC) and lines of code (LOC).

TABLE V: Test Corpus Method Complexity Statistics

	μ	$p_{50\%}$	$p_{75\%}$	$p_{90\%}$	$p_{95\%}$	$p_{99\%}$
LOC	8.65	3	9	19	29	67
Cyclomatic	2.13	1	2	4	6	15
Cross Entropy	5.62	5.42	6.56	7.74	8.59	10.44
Collapsed Cross Entropy	2.67	2.48	2.99	3.59	4.68	6.17

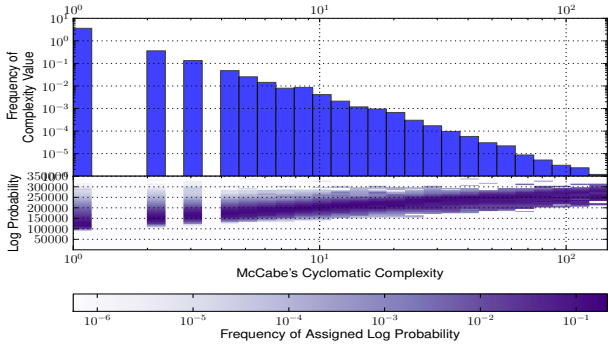
In this section, we consider whether n-gram log probability (NGLP) can be used as a complementary complexity metric. The intuition is that code that is more difficult to predict is more complex. First it is interesting to examine the distribution of the existing complexity metrics across a large test corpus. This is shown in Figures 7a and 7b. We see that both CC and LOC follow a power law, in which most methods have low values of complexity but there is a long tail of high complexity methods. For this idea to be credible, we would expect NGLP to be correlated with existing complexity metrics but is distinct enough to add information to the existing metrics.

Nevertheless, NGLP differs from the other measures: CC focuses on the number of linearly independent paths in the code, ignoring their length. Conversely, LOC solely considers the length of a method ignoring control flow. In the remainder of this section we will see that NGLP combines both lines of code and independent linear paths of source code, providing a complementary indicator of code complexity.

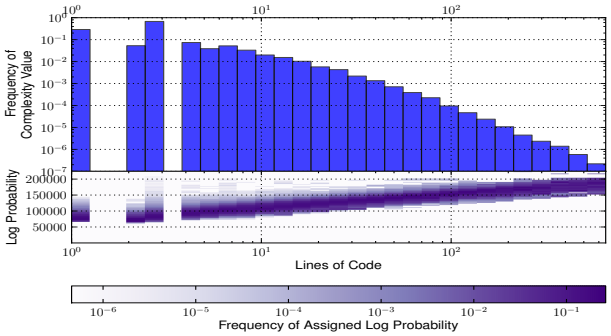
First, we examine the correlation between the NGLP and the existing complexity metrics on the methods in the test corpus. Using Spearman's rank correlation we find that NGLP and the metrics are related ($\rho = .799$ and $\rho = .853$ for CC and LOC respectively). This is higher than the one between LOC and CC ($\rho = .778$). The lower panels of Figures 7a and 7b present a histogram of NGLP and the other methods. Overall NGLP seems to be a reasonable estimator of method complexity.

Table V shows the mean and some of the percentiles of the complexity metrics across the methods in the test corpus. One interesting observation is that 50% of Java methods are 3 lines or less. Manually inspecting these methods we find accessors (setters and getters) or empty methods (e.g. constructors). We also examine methods for which LOC and CC disagree. Unsurprisingly, methods with high LOC but low CC tend to be test and benchmark methods. For example, the elasticsearch project's class `AbstractSimpleIndexGatewayTests` contains the `testSnapshotOperations` method with CC of 1 and LOC 116. Similarly, the `main` method of `PureCompressionBenchmark` consists of 40 lines of code but CC is only 2. On the other end of the spectrum we find methods that act as lookup tables containing multiple branches. For example the `stats` method of the `NodeService` class (LOC: 3, CC: 11) or the `create` method of the `WordDelimiterTokenFactory` class (LOC: 3, CC: 10) contain a series of multiple conditionals. Neither of these methods are necessarily very complex.

When LOC and CC disagree, what does NGLP do? Figure 8 attempts to answer this question comparing the complexity of test corpus methods. The x-axis (Figure 8) shows the



(a) Cyclomatic Complexity Distribution and NGLP per Method. CC fits a power law with slope $a = -1.871$.



(b) Lines of Code Distribution and NGLP per Method. LOC fit a power law with slope $a = -1.624$.

Fig. 7: Histograms (log-scale) of 5,277,844 Project Methods on the test corpus for McCabe's CC and LOC. The distribution of NGLP (collapsing identifiers) for these methods is shown below each histogram. Methods with complexity exceeding the 99.99 percentile are not shown.

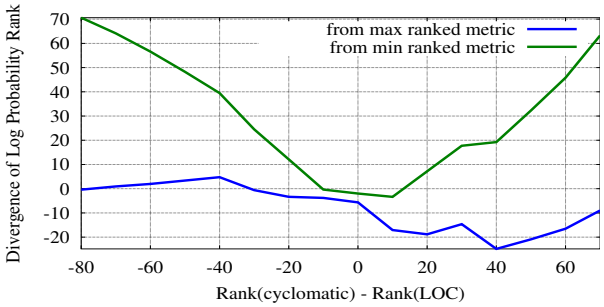


Fig. 8: From the test methods we sample 10,000 sets of 100 and rank them using the three metrics. On the y-axis we plot the average disagreement intensity between NGLP rank and the min and max of either CC or LOC rank.

"*disagreement intensity*" (DI) i.e., difference between the CC and LOC rank of a method. We bin the methods based on the CC vs LOC DI and plot the average DI between NGLP and the maximum and minimum rank of the method by either CC or LOC. We see that, on average, when LOC ranks a method higher, NGLP follows that metric since DI between NGLP and LOC tends to zero. As CC considers a piece of code more complex compared to lines of code, NGLP ranks

the snippet's complexity slightly lower but never lower than 25 positions from that metric (DI between NGLP and CC is never less than -25). Therefore, NGLP seems to be a trade-off between these two metrics, avoiding cases in which either LOC or CC underestimate the complexity of a method.

Finally, we can gain insight by examining the cross entropy using the collapsed model. Methods with high (collapsed) cross entropy tend to contain highly specific and complex code. For example, in the elasticsearch project we find methods that are responsible for parsing queries. When looking at cross entropy, most methods have a similar (rank-wise) complexity. Nevertheless, there are minor differences: methods with relatively lower cross entropy compared to CC and LOC tend to excessively use structures such as multiple nested ifs (e.g. elasticsearch unwraps some loops for performance). On the other hand methods with relatively higher cross entropy are lengthy methods performing multiple tasks that do not have very complicated execution paths but include series of code fragments closely coupled with common variables. For example, the `PluginManager.downloadAndExtract` method of elasticsearch is 163 lines of code long with CC 28 and collapsed cross entropy 1.75 bits, and is placed by cross entropy 10 positions higher compared to the other metrics. These methods can probably be converted into helper objects and their variables to its fields.

In summary, from this analysis we observed that NGLP and cross entropy offer a different view of code complexity, quantifying the trade-off between branches and LOC and subsuming both metrics. *Compared to other complexity metrics, such as LOC, McCabe's and Halstead complexity, the NGLP score rather than solely depending on intuition, also depends on data*. This means that NGLP adapts its value based on the training corpus and is not "hard-wired". An anonymous reviewer argued that code complexity may be solely related to structural complexity, this would indicate that we should used the collapsed n-gram model. However, we argue that complexity should measure the difficulty of predicting code. This means that a piece of code with simple structure but many domain specific terms is more complex, compared to one that contains more common identifiers, since it would be harder for a human to understand. We also believe that this is a better approach because it is able to capture subtle patterns and anti-patterns that were previously too complicated to factor in complexity metrics. Given the promising results in this section, we propose that *practitioners replace hard-coded complexity metrics with data-driven ones*.

B. Log Probabilities at a Project Level

Previously, we studied n-gram model log probability and cross entropy as complexity metrics. In this section, we present how NGLP varies across files. Figure 9 shows the distribution of NGLP across files in a subset of test projects. Each project has its own distribution but there are some common aspects. Most of the files have NGLP constrained on a small range, while there is a small number of files with very large or very small values.

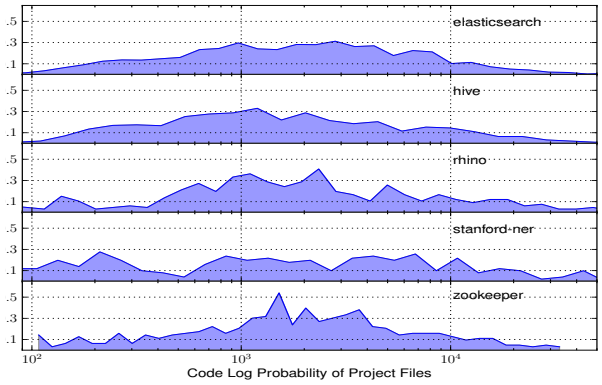


Fig. 9: Distribution of the n-gram log probability of files (in semilogx scale) of five sample test projects.

Since in Java every file is a class or an interface, this allows us to extend our observations to the design and coding of classes and get insight into object-oriented software engineering practice. From manual inspection of the sample projects, the files with the lowest NGLP are those that contain interfaces. In the rhino project the average NGLP of interfaces is 215 while classes have an average of 7153. The same is the case with cross entropy. This observation is interesting and shows the significance of interfaces to the simplification of software packages both from an architectural viewpoint and the perspective of code readability. Interfaces (APIs) are the simplest part of each project, abstracting and hiding implementation details. For practitioners, this translates that *when designing and supporting APIs their NGLP has to be kept low*.

In the middle of the NGLP spectrum lie classes and tests with well defined responsibilities implementing "helper" components. Finally, the most complex (high NGLP) classes are those responsible for the core functionality of the system. They provide complex logic responsible for integrating the rest of the components and core system functionality.

C. n-gram Cross Entropies Scores across Projects

The previous section discussed how NGLP differs across files in a single project. We now focus to the cross entropy variance across projects. Figure 3 shows the distribution of cross entropies of all test projects. Manually inspecting the projects, we make rough categorizations. At the lower cross entropy range we find two types of projects:

- Projects that define APIs (such as jboss-jaxrs-api_spec with cross entropy of 3.2 bits per token). This coincides with our observation in Section VI-D1 that interfaces have low NGLP.
- Projects containing either example code (such as maven-mmp-example and jboss-db-example with cross entropy 3.23 and 2.97 bits per token respectively) or template code (such as Pedometer3 with 2.74 bits per token —a small project containing the template of an Android application with minor changes). Intuitively example projects are sim-

ple and contain code frequently replicated in the corpus.

Projects with the high cross entropy vary widely. We recognize two dominant types:

- Projects that define a large number of constants or act as "databases". For example CountryCode (cross entropy 9.88 bits per token) contains one unique enumeration of all the country codes according to ISO 3166-1. Similarly, the text project (7.19 bits per token) dynamically creates a map of HTML special characters.
- Research code, representing new or rare concepts and domains. Such projects include hac (Hierarchical Agglomerative Clustering) with cross entropy of 7.84 bits per token and rlpark, a project for "Reinforcement Learning and Robotics in Java" (cross entropy 6.37 bits per token).

For both of these categories, as discussed in Section V, the identifiers and literals pose the biggest hurdle for modeling and predicting their code. Code files of the first kind are usually unwanted in codebases and can be converted to more efficient and maintainable structures. The latter type of projects represent code that contains new "terms" (identifiers) representing domains not in the train corpus. Thus cross-entropy can translate to actionable information by detecting new concepts and code smells.

D. Entropy and the Rhino Project: A Case Study

The entropy "lens" that n-grams offer allows for a new way of examining source code. In this section, we present a empirical case study of a single project using this perspective. From the test projects we chose rhino to perform a closer analysis of the characteristics of its codebase. rhino[5] is an open-source implementation of JavaScript in Java maintained by the Mozilla Foundation. It contains code to interpret, debug and compile JavaScript to Java. rhino also contains APIs to allow other applications to reuse its capabilities.

1) File Cross Entropy: Examining how cross entropy varies across files, we notice that files that have low cross entropy are mostly interfaces such as `Callable` (2.75 bits), `Function` (3 bits) and `Script` (3.18 bits). In the middle of the spectrum we find classes such as `ForLoop` (3.94 bits), `InterpreterData` (4.5 bits). Finally, the most complex classes are those implementing the core rhino functionality such as `Parser` (5.06 bits), `ClassCompiler` (6.06 bits) and `ByteCode` (7.86 bits) responsible for parsing Javascript and creating JVM code for execution. Empirically, we observe that classes with the highest cross entropy combine interfaces and the various components of the system to provide core system functionality (i.e. a JavaScript compiler for JVM in rhino's case).

2) Learning from files: In this section, we ask whether all files are learned at the same rate. Figure 10 shows rhino's learning curve plotting the distribution that cross entropy follows. The darkest colored area represents the middle 10% of the cross entropies, while as the color fades we increasingly

[5]https://www.mozilla.org/rhino/

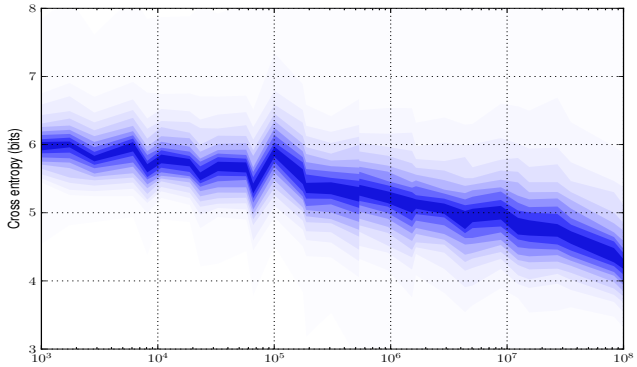


Fig. 10: rhino 3-gram learning curve. The graph shows the variance of cross entropies across the project files.

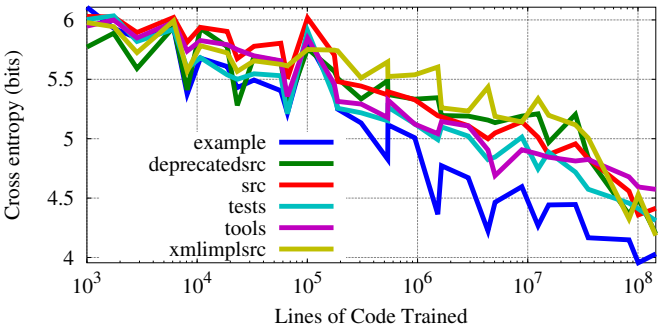


Fig. 11: rhino 3-gram learning per project folder showing how cross entropy as the LM is trained on more data.

add 10% of the files. We observe (Figure 10) that not all files in the project have similar learning behavior. Although the median of the cross entropy is decreasing, the variance across the files increases. This suggests that classes differ in their learnability: The most predictable class (`Callable`) decreases its cross entropy by 58.22%, on the other hand, classes with high density of identifiers such as (`ByteCode`)—an enum of bytecodes—increases its cross entropy by almost 72%.

Finally, we study cross entropies and learnability across rhino's project folders. According to Figure 11 all folders are learned at a similar rate with the only exception of the `example` folder. This confirms the observation in Section VI-C that projects with example code have low cross entropy.

3) Understanding Domain Specificity: Previously we found that identifiers are responsible for the largest part of the cross entropy of source code. We take advantage of this effect combined with the massive cross-domain nature of our corpus to get an indication of the domain specificity of files.

For rhino's files we compute the difference of the cross entropy under the collapsed and full n-gram LMs. Subtracting these entropies returns the "gap" between structure and identifiers which we name *Identifier Information Metric* (IIM). IIM is an approximate measure of the domain specificity of each file, representing the per token predictability of the identifier "vocabulary" used. The files with small IIM are the least specific to rhino's domain. For example, we find classes

such as `UintMap` (1.42 bits), `NativeMath` (1.18 bits) and `NativeDate` (1.47 bits) and interfaces such as `Callable` (0.76 bits), `JSFunction` (0.84 bits) and `Wrapper` (1.57 bits).

On the other hand classes with large IIM are highly specific to the project. Here we find classes such as `ByteCode` (8.68 bits), `CompilerEnvirons` (4.55 bits) and tests like `ParserTest` (4.52 bits) that are specific to rhino's domain. Thus, scoring classes by IIM can help build recommendation systems allowing practitioners to identify reusable code. Overall, this observation is interesting since it implies that *by looking at the identifiers and how common they are in a large corpus, we can evaluate their reusability* for other projects or find those that can be replaced by existing components. IIM may also be helpful in code navigation tools, allowing IDEs to highlight the methods and classes that are most important to the program logic.

VII. Related Work

Software repository mining at a large scale has been studied by Gabel and Su [4] who focused on semantic code clones, while Gruska et al. [5] focus on cross-project anomaly detection. However, this work does not quantify how identifier properties vary, since it ignores variable and type names. Search of code at "internet-scale" was introduced by Linstead et al [8]. Another GitHub dataset, GHTorrent [9] has a different goal compared to our corpus, excluding source code and focusing on users, pull requests and all the issues surrounding social coding.

From a LM perspective, the pioneering work of Hindle et al [1] was the first to apply language models to programming languages. They showed that the entropy of source code was much lower than that of natural language, and that language models could be applied to improve upon the existing completion functionality in Eclipse. We build on this work by showing that much larger language models are much better at prediction and are effective at capturing regularities in style across project domains. We also present new information theoretic tools for analyzing code complexity, explored in a detailed case study.

A different way of machine learning applied to source code classification has been used by Choi et al [10] for detecting malicious code. Apart from autocomplete, code LMs could be applied in programming by voice applications [11], API pattern mining [12] and clone search [4].

Finally, the role of identifiers has been studied by Liblit et al [13] from a cognitive perspective, while their importance in code understanding has been identified by Kuhn et al. [14].

VIII. Conclusions & Future Work

In this paper, we present a giga-token corpus of Java code from a wide variety of domains. Using this corpus, we trained a n-gram model that allowed us to successfully deal with token prediction across different project domains. Our experiments found that using *a large corpus for training these models on code can increase their predictive capabilities*. We further explored how the most difficult class of tokens—namely identifiers—affect the training procedure and quantified the effect of three types of identifiers. *The identifiers seem to have a significant role when mining code*.

Using the trained n-gram model we explored useful information theoretic tools and metrics to explain and understand source code repositories, thanks to the corpus' scale. Using rhino we explored how log probability and cross entropy help us identify different various aspects of the project.

In the future, from the pure token prediction perspective we can extend and combine n-gram LMs with other probabilistic models that can help us achieve better prediction, especially of identifiers. The various metrics presented can implemented in large codebases to assist code reuse, evaluate code complexity and assist code navigation and code base visualization.

Acknowledgments

The authors would like to thank Prof. Premkumar Devanbu and Prof. Andrew D. Gordon for their insightful comments and suggestions. This work was supported by Microsoft Research through its PhD Scholarship Programme.

References

[1] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[2] J. Martin and D. Jurafsky, *Speech and language processing*. Prentice Hall, 2000.

[3] S. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 1996, pp. 310–318.

[4] 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*. ACM, 2010, pp. 147–156.

[5] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *ISSTA*, vol. 10, 2010, pp. 119–130.

[6] P. Bourque and R. Dupuis, *Guide to the Software Engineering Body of Knowledge, SWEBOK*. IEEE, 2004.

[7] S. H. Kan, *Metrics and models in software quality engineering*. Addison-Wesley (Reading, Mass.), 2002.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[9] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 12–21.

[10] J. Choi, H. Kim, C. Choi, and P. Kim, "Efficient malicious code detection using n-gram analysis and SVM," in *Network-Based Information Systems (NBiS), 2011 14th International Conference on*. IEEE, 2011, pp. 618–621.

[11] S. C. Arnold, L. Mark, and J. Goldthwaite, "Programming by Voice, VocalProgramming," in *Assets: International ACM Conference on Assistive Technologies*. Association for Computing Machinery, 2000, p. 149.

[12] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 48–61.

[13] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Psychology of Programming Workshop*. Citeseer, 2006.

[14] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.