On the Impact of Tokenizer and Parameters on N-Gram Based Code Analysis

Matthieu Jimenez
University of Luxembourg
Luxembourg, Luxembourg
matthieu.jimenez@uni.Lu

Maxime Cordy
University of Namur
Namur, Belgium
maxime.cordy@unamur.be

Yves Le Traon, Mike Papadakis
University of Luxembourg
Luxembourg, Luxembourg
yves.letraon@uni.lu, michail.papadakis@uni.lu

***Abstract*—Recent research shows that language models, such as n-gram models, are useful at a wide variety of software engineering tasks, e.g., code completion, bug identification, code summarisation, etc. However, such models require the appropriate set of numerous parameters. Moreover, the different ways one can read code essentially yield different models (based on the different sequences of tokens). In this paper, we focus on n-gram models and evaluate how the use of tokenizers, smoothing, unknown threshold and n values impact the predicting ability of these models. Thus, we compare the use of multiple tokenizers and sets of different parameters (smoothing, unknown threshold and n values) with the aim of identifying the most appropriate combinations. Our results show that the Modified Kneser-Ney smoothing technique performs best, while n values are depended on the choice of the tokenizer, with values 4 or 5 offering a good trade-off between entropy and computation time. Interestingly, we find that tokenizers treating the code as simple text are the most robust ones. Finally, we demonstrate that the differences between the tokenizers are of practical importance and have the potential of changing the conclusions of a given experiment.**

I. Introduction

Natural Language Processing (NLP) [1] techniques realise the assumption that humans exploit partially the complexity of the language by following particular norms. Thus, natural language is composed of small snippets that are repetitive and follow predictable patterns. This phenomenon is called by researchers as the *naturalness* of the language. Recently, the study of Hindle et al. [2] showed that source code follows the same trend. This means that code (small code snippets) is also repetitive and predictable.

This observation paves the way for using statistical language models for code analysis. Such models can provide suggestions for completing code [2], complementing static bug finders [3], [4], detect and correct syntax errors [5], [6], automatically generating code explanations [7], synthesizing code from natural-language specifications [8] and many others [9].

N-gram models operate by *tokenizing* documents (i.e. breaking these into words) and calculating the number of times every sequence of n words appear in a given document corpus. Based on that they estimate the likelihood that a given sequence appears. Their application requires setting parameters such as the length n of the sequences, the unknown threshold (ignoring tokens that appear fewer times than the threshold) and the smoothing technique (scoring unknown sequences).

In the case of code, the appropriate way of tokenizing documents (e.g. source files) is not evident as code can be processed in many ways. Naturally, one can read code as any text document, that is, typically from left to right and top to bottom. However, developers tend to follow the flow of the program (that is not necessarily sequential) by taking advantage of the code characteristics [10], e.g., the grammar (programming language) used, while automated tools like compilers rely on program abstractions such as flow graphs and Abstract Syntax Trees (AST).

Overall, the prominent use of n-grams for source code analysis requires setting a number of parameters. Previous work set them as in the case of natural language. However, given the differences between code and natural language, it is imperative to re-validate and tune the application of the language models in the context of code. This is because there is no empirical evidence related to the generalization of the existing results, from the natural language field to the source code analysis field. For instance, choosing the most appropriate smoothing technique (way of treating unseen sequences), is not evident due to the vocabulary and structural differences between code and text. Moreover, as there is a plethora of parameter possibilities, there is a need for checking the sensitivity of the models w.r.t. these choices and the overall impact of the untuned parameter selection.

We therefore, investigate the effect of code representation when used in language models. We use the 8 tokenizers of Table I. The first two correspond to "an outsider's point of view", i.e., the reader does not have any knowledge about the code structure. The next two correspond to "a developer's point of view", i.e., the reader knows the grammar of the written language. Whereas the last four correspond to "the automated tools point of view", i.e., the reader is a parser that transforms code to a representation like AST.

The differences of 'UTF' with 'UTF woc' and 'Java Parser' with 'Java Parser woc' are due to the way comments are handled. This differentiation is useful as comments can generate noise in our models. The last four tokenizers differ from the way an AST is processed (typically in depth-first or in breadth-first order), and whether or not the AST is pruned of redundant nodes. This is important as language models work with sequence of tokens, which in this case are the different orders that one can visit the tree representations.

In this paper, we address the problem of tuning n-gram models to a given purpose by evaluating 120 different configurations of n-gram models (6 n values, 4 smoothing techniques, and 5 unknown thresholds) combined with the above-mentioned tokenizers. We implemented these configurations in a tool that is available online.[1] In the first part of the evaluation, we assess the capability of the configurations to capture regularities within 20 open-source Java projects.

For each project and configuration, we compute the cross entropy of the project. The cross entropy is a measure commonly used in NLP to assess the efficiency of a language model. Intuitively, it represents how "surprised" a model trained on a given set of documents is when it encounters an unseen one. Therefore, the best configurations should give the lowest entropy, given that one can find local regularities within a given project [2], [11]. This allows us to check the sensitivity of the approach with respect to the studied configurations.

Comparing tokenizers is tricky as each involves its own specific building blocks. Thus, entropy values cannot be compared directly. We bypass this problem by comparing the tokenizers according to the relative entropy differences and the entropy-based rankings of source code files, i.e., we measure whether the models judge and select the unlikeliest or likeliest files similarly.

To further strengthen our study, in the third part of the evaluation we consider a particular experimental scenario and demonstrate that the use of different tokenizers leads to contradictory conclusions. We thus investigate whether buggy files are more likely to have higher cross-entropy values than non-buggy ones, and whether fixing bugs results in a reduced file cross-entropy. These objectives were inspired by the study of Ray et al. [3] and represent a concrete example of research that can be influenced by n-gram model tuning.

Overall, our study involves 20 large open source Java projects and a dataset of 3,800 bugs. Our results show that the Modified Kneser-Ney smoothing technique performs best. Choosing an n value equal to 4 or 5 seems to be the most appropriate choice for all tokenizers. Perhaps more importantly, we find a large disagreement between the tokenizers and show that not all of them are appropriate for particular problems. We further demonstrate this by investigating the link between entropy and bugginess, showing that only 2 out of our 8 tokenizers are capable of exploiting this link. Interestingly, our results show that the closer, to human perspective (unprocessed code), the used tokenizer is, the better the model is at detecting the effects of bug fixes. In this regard, tokenizers treating code as pure text are thus the winning ones.

In summary our paper makes the following contributions:

1) It identifies and explores the impact of different parameters on the predictability of the n-gram models for code.
2) It demonstrates large disagreements between the predictions of models that use different tokenizers.
3) It provides evidence that untuned n-gram models have the potential of biassing research conclusions.

[1]URL is anonymised for the purpose of double-blind review

TABLE I
STUDIED TOKENIZERS

Tokenizer	Representation	Delimiter	Technology	Specificities
UTF	Raw	Non-Alphanumeric	Terrier	-
UTF woc (UTFw)	Raw	Non-Alphanumeric	Terrier	without comments
Java Parser (JP)	Raw	Java Grammar	Java Parser	-
Java Parser woc (JPw)	Raw	Java Grammar	Java Parser	without comments
AST Depth First (DF)	AST	Node	Java Parser	depth first
AST Breadth First (BF)	AST	Node	Java Parser	breadth first
Pruned AST Depth First (PDF)	AST	Node	Java Parser	Pruned, depth first
Pruned AST Breadth First (PBF)	AST	Node	Java Parser	Pruned, breadth first

II. N-GRAM MODELS

Language models operate on sequences of words and compute their probability distribution. In code analysis, such sequences are the code fragments such as source files, Java classes or specific code lines. Words are the constituent tokens of the code fragments. Let $s = t_1, \ldots, t_m$ be a finite sequence of tokens. We denote by $P(t_1, \ldots, t_m)$ the non-zero probability that can be estimated for s by a given language model. The model is first *trained* on a set of sequences, named the *training corpus*. The training process determines the probability distribution of the known sequences, which in essence compose our model. The distribution typically results from the computation of the maximum likelihood estimates, that is, the probability of a (sub-)sequence is given by the number of times it appears in the training set divided by the number of (sub-)sequences in the set.

N-gram models are a particular type of language models that are fast to train and easy to use. Their origin can be traced back to Shanon's work [12] that presented the task of guessing the next letter in a text. Such models statistically estimate the probability that a token follows a given preceding sequence. Accordingly, the probability of a sequence is defined as the product of the probability of each token to follow its prefix. Thus, $P(s) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2)\ldots P(t_m \mid t_1\ldots t_{m-1})$. N-grams also assume a Markov property of order $n-1$. Thus, the probability of occurrence of a token in a sequence depends on the $n-1$ previous tokens, i.e., $P(t_i \mid t_1\ldots t_{i-1}) = P(t_i \mid t_{i-n+1}\ldots t_{i-1})$. Then, the probability of a sequence becomes a product of n-sized conditional probabilities. For example, for $n = 3$ the probability of s is given by $P(s) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2)\ldots P(t_m \mid t_{m-2}t_{m-1})$. Following the above equation, an estimate of the probability of s is the product of estimates for its constituent conditional probabilities (based on the training corpus). A maximum likelihood estimate for $P(t_i \mid t_{i-n+1}\ldots t_{i-1})$ is obtained by dividing the number of occurrences of $t_{i-n+1}\ldots t_i$ by the number of occurrences of the prefix $t_{i-n+1}\ldots t_{i-1}$.

Interestingly, the training corpus is not the only one that can impact the utility of an n-gram model. There are multiple parameters that can influence these results, with the most obvious one being the size n. To evaluate alternative models, one can carry out intrinsic evaluations to measure the performance of the models on some unseen data. In our case, this *test corpus* consists of code fragments that were not part of the training corpus. Then a model m_1 has a higher utility than a model m_2 if it can better predict the sequences of the test corpus. In other words, m_1 assigns a higher probability to the test corpus.

In practice, one does not use the raw probability but rather rely on a derived measure named *cross entropy*. It is given by $H(s) = -\frac{1}{m} \log P(s)$ which, for an n-gram model of size n, is equivalent to $H(s) = -\frac{1}{m} \sum_{i=1}^{m} \log P(t_i \mid t_{i-n+1} \dots t_{i-1})$. A lower cross-entropy thus means a better model. Intuitively, the cross-entropy indicates how "surprised" the model is when confronted to s. More formally it describes the average number of bits required to encode the data from the test set that have a distribution P using the code that is optimal for a distribution Q (the model built using the training set).

The choice of the n-gram size n can have a major impact on the model utility. Indeed, a higher n allows the model to better discriminate the sequences of tokens. However, it takes a longer time and more memory to train since more sequences have to be considered when computing the conditional probabilities.

Another point that can influence the model is the way it deals with *unknown words*. It may indeed happen that the model encounters some tokens (in the test corpus) that never appeared in the training corpus. The probability of this token is thus zero according to the model, which leads to an infinite cross entropy. In source code, this problem typically arises when new variable names are introduced. Of course, it is unrealistic to consider all potential variable names. The *vocabulary* of our model is thus *not closed*.

A common way to deal with this issue is to replace all words with less than k occurrences in the training corpus with a special token `<UNK>` (where $k > 0$). Since `<UNK>` occurs in the training corpus, the model estimates and assigns some probability values for this token. Then, each time an unknown word appears in the test corpus, the model interprets it as `<UNK>` and assigns it a non-zero probability. The aforementioned parameter k, named the *unknown threshold*, obviously affects the quality of the model since it modifies the estimated probability value of every token.

A similar problem occurs when dealing with *data sparsity*. As it is rather unlikely to observe every possible sequence of tokens in a training corpus, it might happen that sequences absent from the training corpus appear for the first time in the test corpus. This is even more common than unknown words, especially for higher-sized n-grams that work with long sequences. To prevent the model from assigning zero probabilities to these sequences, several *smoothing* techniques have been proposed. Intuitively, smoothing reserves a part of the probability mass for the unseen sequences, and estimates a probability for known sequences based on the rest of the probability mass. Smoothing has the effect of improving the accuracy of the models, especially in the case of probability estimated from few counts.

There are many smoothing techniques but we only focus on the four most popular ones. For additional details on the subject please refer to the comprehensive survey of Chen and Goodman [13]. We study the following four techniques: Witten Bell [14], [15], Absolute Discounting [16], Kneser Ney [17] and Modified Kneser Ney [13].

Witten Bell was first introduced for text compression, but it can be used for smoothing language models as well. It is an instance of another smoothing technique called Jelinek Mercer [18] where the n-th order smoothed model is defined recursively as a linear interpolation of the maximum likelihood for n-th order and the (n-1)th order smoothed models. This technique uses as λ the probability of observing an n-gram for the first time, i.e., the number of n-grams appearing more than once over this number plus the total count of n-gram.

Absolute Discounting involves an interpolation between higher order and lower order. Instead of multiplying the higher order by a computed λ, a fixed discount is subtracted from it.

Kneser Ney is an extension of Absolute Discounting with a cleverer way of computing the discount, based on the idea that lower-order models are significant only when the number of occurrences is small or zero in the higher-order model.

Modified Kneser Ney is a further improvement that uses three different discounts depending on the number of occurrences of the considered n-gram.

Like the unknown threshold, the choice of a specific technique is important as it impacts the cross entropy returned by the model. Earlier work on software naturalness [2] argue that *Kneser-Ney* is the most appropriate, but have not presented detailed experiments confirming this claim. As we will see, our experiments fill this gap and empirically evaluate the different techniques w.r.t. other parameter values.

III. Related Work

The application of machine learning to software engineering has received a growing interest in the recent years [9]. In particular, the study of software naturalness [2] has given birth to many approaches for generating source code (e.g., code completion [2], synthesis [19], review [20], obfuscation [21] and repair [22]) and performing static analyses [23]–[25].

According to Allamanis et al.'s survey [9], n-grams are among the most popular language models. They have been used mainly for code completion [2], [11], [26]–[28], program analysis [23], bug detection [3], code review [20], and information extraction [29], [30]. Despite being popular we are unaware of any systematic and empirical evaluations that analyse the sensitivity of these models w.r.t. their parameters, although many papers give a few insights.

In their seminal work on code naturalness, Hindle et al. [2] already inform us that Modified Kneser-Ney smoothing gives good results for software corpora. However, they state that *"these are very early efforts in this area"*, which motivated our systematic evaluation of other smoothing techniques. According to their experiments, the reduction of cross entropy with higher-order n-grams saturates around 3- or 4-grams, whereas our evaluation shows that the reduction from 4-grams to 5-grams remains statistically significant in many cases. Finally, they tokenize code just like any English text and do not consider alternatives like AST-based tokenization.

In [26], Nguyen et al. add semantic information, e.g., the data type of a variable, to lexical tokenization in order to improve code suggestion. Their approach inherently considers n-grams of multiple sizes; thus we do not know how a fixed

n-gram size would affect their results. Also they used only additive smoothing, which is the simplest but arguably the less efficient technique [31], [32].

A subsequent work [27] tackle the problem of suggesting API calls. With this objective in mind, the authors argue that graph-based representations (e.g., AST and control flow graph) are more appropriate than n-grams computed from lexical tokenization. Based on such representations, they implemented API suggestion algorithms that outperform 8-gram models equipped with additive smoothing. These results motivate our interest towards AST tokenization, although we found that lexikal (UTF) tokenizers are better than AST tokenizers at detecting bugs. Similarly, Hsiao et al. [23] tokenize a program using a dependency graph (representing data-flows between the program statements). Their tokenizer hardly scale from 5-gram onwards, but even with smaller n-gram sizes it outperforms 7-gram models obtained from lexical tokenization. The authors do not mention the use of any smoothing technique.

Tu et al. [11] propose a cache model that captures local regularities. Their evaluation shows that the best results (in terms of cross-entropy reduction) are obtained by combining the cache with standard n-gram models, as these capture different regularities. They also show that the size of the cache has a significant impact on cross entropy, and suggest that trigrams are sufficient. They do not explicitly mention what smoothing technique they use. As before, we argue that 4-grams and even 5-grams can yield significant improvements in some cases, and that the choice of a smoothing technique is not without impact. On the other side we did not consider integrating a cache model, which is an interesting direction for future work.

Raychev et al. [28] focus on suggesting API calls. The originality of their approach lies in that it combines n-grams with recurrent neural networks. Their experiments show a substantial improvement in effectiveness over standard n-grams, yielding 90% of relevant suggestions in top 3 candidates. They rely on trigrams and Witten-Bell smoothing, but did not study how these choices affect their results.

In a peripheral work, Hellendoorn et al. [20] correlate the naturalness of pull requests in GitHub (computed by n-gram models) to their acceptance rate and the degree to which the requests are debated. They acknowledge the importance of choosing an appropriate size and smoothing technique, although they do not report on the sensitivity of their approach w.r.t. these parameters.

Sharma et al. [29] propose an approach to identify tweets related to the software industry. More precisely, they use n-gram models to compute the cross entropy of tweets, and rank these accordingly. They evaluate the effectiveness of their approach with different n-gram sizes, and discover that 4-grams still offer an interesting marginal gain. As for smoothing, they assess only the Katz backoff model [33] and do not consider the other alternatives.

Saraiva et al. [34] perform a study on n-gram model specificities for source code, but focused on different research questions than the present paper. They first attempt to determine whether building language models specific to an application or specific to a developer can lead to better results. Then they investigate the importance of the temporality of language models. They found out that developer- and application-specific models were indeed performing better than general models, while temporality has little to no effect.

Finally, Yadid and Yahav [30] make use of n-gram models to correct and complete code fragments that were extracted from video tutorials. They use unigrams and bigrams conjointly, but have not investigated other parameter settings. Also, they do not mention the smoothing technique they use.

The above discussion highlights that n-grams is a frequently used statistical model. Many of the previous studies recognise the importance of the chosen parameters, but paradoxically evaluate only a few configurations. Moreover, none of the previous approaches considers alternative representations and their impact on the experimental conclusions one can draw. Thus, our paper raises the awareness of what can go wrong and what should be tuned in order to draw reliable experimental conclusions.

IV. Research Questions

Our aim is to investigate how the different parameters involved when building n-gram models impact the source code analysis tasks. In particular we seek to investigate the parameters related to the size n, the smoothing technique and the way code is tokenized. This is important as the factual differences between natural language and source code have not been exploited. For instance, natural language almost always flows sequentially, while source code includes many conditional jumps, which may necessitate a different analysis. Therefore, the use of different tokenizers should play a major role on the model's utility. Thus, our first research question is:

RQ1: What is the impact of the different parameters on n-gram models when used for source code analysis? Is there an optimal configuration for all tokenizers?

We answer this question by computing, for 8 different tokenizers, the average cross entropy of 20 Java projects using 24 sets of parameter configurations, i.e., 6 n values * 4 smoothing techniques. Then we check whether optimal parameters stand out across all the tokenisers.

Measuring the cross entropy provides an insight into the relative performance of studied configurations. However, this information, i.e., low or high entropy values, says nothing about the use of different tokenizers for distinguishing source code files with respect to naturalness of code. Indeed, cross entropy measures the distance of the test corpus from the generated model with respect to the involved building blocks of the model trained on the training corpus. Different tokenizers rely on different views of the code and hence result in models trained on different building blocks. Since the building blocks differ, we cannot directly compare the tokenizers with cross entropy.

To bypass this problem, we compare the tokenizers with respect to the relevant information they provide, i.e., their ability to distinguish and rank source code files (natural and

unnatural ones). Hence our second research question regards the information that different tokenizers learn:

RQ2: Do models built based on different tokenizers learn the same information for the same code?

To answer RQ2 we consider specific configurations that return good results (in terms of lower cross entropy). Then, we measure the relative agreement between the tokenizers, by computing the correlation between the cross-entropy values they provide on the source code files we study. We deem this comparison as valid as it measures the relative volume of agreement between the entropy value differences and their relative rankings with respect to a set of source code files. We also consider the correlation between these entropy values and the number of lines of code in order to check the effects of size on our results. Strong correlations indicate a large agreement between the tokenizers, while weak correlations indicate a disagreement.

As we will see later, our results show a large disagreement between tokenizers. Yet, it is unclear whether these differences are capable of impacting the findings of research studies. Thus we ask:

RQ3: Does the use of different tokenizers has the potential of impacting research results?

To answer this question, we set a simple experiment to investigate whether buggy files are more likely to have higher entropy values than non-buggy ones and whether fixing bugs results in a reduced file entropy. These objectives were inspired by the study of Ray et al. [3], which investigates the link between buggy lines of code and naturalness (and the impact of bug fixes on it).

Finally, we investigate the use of a 'special' parameter of the n-gram models. This is the unknown threshold k, which determines the confidence on the estimations made by the models. This is a special parameter as it involves a trade-off between the accuracy of the model and the information it captures. Thus, by setting the threshold at higher values we get more accurate but also more coarse-grained models. Therefore we investigate:

RQ4: What is the impact of setting the unknown threshold at different levels?

We examine this issue by using 5 unknown thresholds k and measuring their impact on both entropy and results of RQ3.

V. Methodology

A. Test Subjects

To answer our research questions, we rely on data gathered from 20 open source software from the *Apache Commons* project [35]. Apache Commons comprises reusable open source Java software projects which are intensively developed and maintained. At the time of writing this paper, a query about "org.apache.commons" on Github returned close to 7,000,000 different Java files. This indicates that the selected projects are popular.

Building our experiments around Apache Commons projects has many benefits. First, the projects follow strict development guidelines. This fact has a potential effect of improving the performance of language models (as repetitiveness is encouraged). We deemed this as an advantage as it reflects industrial settings where code conventions and implicit coding rules are followed throughout whole companies. Second, each one of the selected projects has its own usage context and implements different functionalities. This fact challenges our models, whose performance should generalize over all projects. This counterbalances the facility offered by coding conventions while further increasing the transferability of the results to real-world or industrial projects (indeed, a company typically develops software for slightly different application domains). Third, every Apache Commons project reports its bugs on the same platform with similar reporting guidelines. This facilitates the creation of a bug dataset, which is required to address RQ3.

TABLE II
Dataset Statistics

Project	Latest	Files	kLoC	Versions	Bugs
BCEL	6.1	488	75	5	93
BeansUtils	1.9.3	257	72	18	155
CLI	1.4	50	12	6	91
Collections	4.1	525	118	12	186
Compress	1.15	329	70	18	309
Configuration	2.2	457	125	15	325
CSV	1.4	28	8.4	5	67
DBUtils	1.7	92	15	8	23
EMail	1.4	47	12	8	51
FileUpload	1.3.3	54	10	10	67
IO	2.5	227	55	14	213
JCS	2.2.1	562	102	6	102
Jexl	3.1	108	23	8	126
Lang	3.6	318	141	20	567
Math	3.6.1	970	218	16	830
Net	3.6	270	59	20	246
Pool	2.4.2	79	24	22	154
Rng	1.0	124	14	1	3
Text	1.1	104	25	2	38
VFS	2.2	382	52	4	214
Total	-	5,471	1,230	218	3860

Apache Commons involve 41 projects. We selected 20 of them based on the following three criteria:

- **Date of the last update**. A recent update indicates that the project is still active and of interest. It also means that developers continue to fix bugs.
- **Size of the project**. A larger project increases the size of the training corpus, and thereby reduces the risk of overfitting for our models.
- **Length of project history**. A long history usually implies a higher number of bugs to study and the possibility to observe whether results generalize over releases.

From the data of the 20 projects we select (recorded in Table II), we analyse their Java files. Everything else was not considered as it contains irrelevant information for our study. In our first two and our last research questions we analyse the source code of the latest project release. For the third research question, which involves the analysis of bugs, we had to go back in the project history in order to identify and collect a large set of bugs. As a consequence, we had

to gather multiple releases of the projects and identify the versions containing the studied bugs. We also had to identify the versions where these bugs were removed (fixed). To collect our data we implement the following procedure, which was added as part of our toolchain:

1) We crawl the full commit history of the projects and identify all the commits that mention an issue ID.
2) For each issue ID we check whether the issue is mentioned on the issue tracker. We then check whether it refers to a bug and if so, we retrieve the affected version.
3) For each issue ID referring to a bug, we go back to the corresponding commits and get the list of files modified by these commits. Then we store those files and flag them as fixing the previous buggy version.

Table II presents the characteristics of our dataset. `Latest` is the latest version of the project at the time of writing, which is also the version we consider for RQs 1, 2 and 4. `Files` is the number of Java files in this version. `kLoC` is equal to the number of lines of code of the version (in thousands). `Versions` refer to the total number of versions of the project we studied in RQ3. Finally, `Bugs` refers to the number of unique bug-related issues retrieved by the above procedure.

B. N-Gram Model Configurations

1) Tokenizers: We build n-gram models from source code using 8 tokenizers. Details about the tokenizers we use are presented in Table I. These can be categorized in three main groups. The distinction between the groups regards the representation of the source code. Thus, the first group (first two rows in Table I) comprises tokenizers that treat code as text and directly use it as input. In these cases a sequence of words is separated by delimiters. The second group (rows three and four in Table I) comprises tokenizers that delimit code based on the language grammar. Tokenizers of the third group, (the last four rows in Table I), are defined based on the AST representation of the code. Thus, these tokenizers perform the tokenization based on a serialized representation of the AST.

The tokenizers of the first group are standard UTF tokenizers. These are similar to the one used by the open-source search engine Terrier [36]. They split the text into groups of alphanumeric tokens while still keeping the delimiters in the sequence. The only difference between the two is that in the first considers the complete file, whereas the second ignores comments. This should give us insights about the sensitivity of the models wrt. code comments. Note that previous works tend to completely ignore code comments, e.g., [2].

UTF tokenizers take all non-alphanumeric characters as delimiters. This fact may yield inappropriate tokenizations for specific cases. For example, variable names using an underscore are separated in three. A similar case is the float numbers (also separated in two words). This motivated the need for the second group of tokenizers. Thus, the `Java Parser` and `Java Parser woc` tokenize code according to the language (Java) grammar. This implies a correct identification of the Java tokens. Another effect of using the grammar is that the `Java Parser` considers a block of comments as a single token. Thus, ignoring comments should have a minor impact on our models. In our implementation we perform the parsing based on the Java Parser tool [37]. The reason behind this choice is that this tool facilitates the treatment of ASTs by providing specific data structures (i.e., following the *visitor* design pattern), which was useful for implementing the remaining tokenizers.

The four tokenizers of the third group differ in the way they process the program's AST. The first two of them, serialize the complete AST (they print the type of every node as well as package, method, and variable names) in a specific order that depends on the visit strategy, i.e. breadth first or depth first. The last two tokenizers of this group implement a pruned version of the serialization process (only the text of non-redundant nodes is considered). We consider a node as redundant when it does not directly correspond to a string in the source code. In other words, this node serves a structural purpose, e.g., every variable name is preceded by a node of type `NameExpr`. Studying all these alternative tokenizers helps us understand whether the visit strategy and the redundancies in the ASTs have a significant impact on our results.

2) Language Modeling: To compute the cross entropy, we need to use some code parts as a training corpus and some others as a test corpus. We thus, define an n-gram model as a stateful service interface with two methods: (1) `train`, which takes as input a corpus and trains the model accordingly; (2) `entropy`, which returns the cross entropy of an input sequence of tokens based on the trained model. Our implementation uses the Kyoto Language Modeling Toolkit (Kylm) [38]. Kylm is an established tool developed in Java that provides all the functionalities needed for our experiments. Indeed, it allows one to specify the size n of an n-gram model, its unknown threshold, and the associated smoothing technique. Following the principle of interface segregation [39], one can easily switch to another language-modelling tool by developing an alternative implementation of the n-gram interface defined above.

C. Research Protocol

1) RQ1: To address RQ1, we consider 24 configurations, which are the combinations of n values 2 to 7 with 4 smoothing techniques (absolute discounting, Kneser-Ney, modified Kneser-Ney, and Witten-Bell). For each tokenizer t, project p and configuration c, we build an n-gram model parameterized by c, and compute the average cross-entropy over 10-fold cross validation of p's source code tokenized by t, where only one of 10 successive files (according to the file system ordering) belongs to the test set at each iteration. This leads to a total of 3,840 cross validations.

Every iteration of a given 10-fold cross validation involves 90% of the source files for training the models, whereas the remaining ones compose the test corpus. We operate on a file-level granularity as it is common in defect prediction, and for simplicity when using the AST tokenizers. Indeed, in Java,

ASTs are built class by class, and a (public) class is commonly contained within one file. It is noted that for now, we set the unknown threshold k to 1, as the influence of the unknown threshold k is studied in RQ4.

2) RQ2: To address RQ2, we build an n-gram model m based on the configuration that is the most representative (identified in RQ1) and, for every tokenizer t, we make m compute the cross entropy of all source files tokenized by t. Then, we check whether there is a correlation between the cross-entropy values across each pair of tokenizers. This allows us to check whether tokenizers agree between them when comparing the source code files. We also verify the existence of a correlation between the number of Lines of Code (LoC) and the entropy values associated to each tokenizer to check whether our observations are influenced by the code size.

To perform the comparisons, we carry out two correlation tests. First, we compute the Pearson correlation coefficients to formally assess whether there is a strong linear relationship between the tokenizers (i.e., the entropy values change similarly among the files when using different tokenizers). We also check this relation with the LoC. Second, we measure the ordinal association between the same variables using the Kendall's tau coefficients. Ordinal relations differ from the linear relations as they do not consider the size of the differences between the values. This allows determining whether the code files are ranked differently according to their entropy and LoC.

3) RQ3: To address RQ3 we investigate the influence of tokenizers on the findings of a research experiment we design. Thus, we investigate whether (1) buggy files tend to be more unnatural than non-buggy ones and (2) fixing a bug makes a file more natural.

For (1), we compute the entropy of each file successively in the release using all other files for training. This process ensures a common training and evaluation ground that is deterministic and reproducible. This way we avoid using large training corpus and focus on the relative differences between the files under analysis and the rest of the project. The idea is that the more improbable a file, w.r.t. the others of the same project, the more likely it is to be problematic. Future work includes the use of cross-project training or past-release project training.

Based on the entropy values, we can observe whether files flagged as buggy have indeed a higher cross entropy. Then for (2), we compute, for each buggy file, its cross entropy in the release just before the patch and after the patch. We use a model built on the last affected release – excluding the assessed file – and analyse the percentage of difference between the two cross-entropy values.

To see the actual impact of the tokenizers, we compare the conclusions that one can draw for the above experiments when using one tokenizer instead of another. In case of contradictions, we can conclude that the tokenizer choice is important (as a different choice may imply a different conclusion for a given task).

4) RQ4: To study, the impact of the unknown threshold, we repeat the analysis followed in RQ1, but for different thresholds. Thus, the following values of k are studied, 1, 2, 4, 8 and 16. This adds up 15,360 new 10-fold cross validation to the one done for RQ1. Then we study the impact of this parameter on the findings of RQ3. To do so, we measure the differences between buggy and non-buggy code and the impact on entropy when fixing a file (using a k equal to 8). Finally, we compare these results with those obtained in RQ3.

5) Statistical Comparison: To judge the significance of the observed differences we use standard statistical tests. We used the Wilcoxon signed-rank test to measure the arbitrariness of our results. We choose the Wilcoxon hypothesis test as it is non-parametric and thus, it does not make any normality assumptions. As it is typically performed, we adopt a significance level of 0.05, below of which we consider the differences statistically significant. To measure the size of differences we used the Vargha Delaney effect size $\hat{\mathrm{A}}_{12}$, which quantifies the size of the differences (statistical effect size). The $\hat{\mathrm{A}}_{12}$ evaluates the number of times that the data of one method are higher than those of the other. $\hat{\mathrm{A}}_{12} = 0.5$ suggests that the two value sets are more or less the same, while, $\hat{\mathrm{A}} > 0.5$ suggest that the first set has higher values than the second one. Values of $\hat{\mathrm{A}}_{12} < 0.5$ suggest that the second set has higher values.

VI. Results

A. RQ1: Optimal Configuration

We start our analysis by identifying the impact of the smoothing techniques on the entropy of the source code files. Our analysis is based on the principle that a smoothing technique giving lower entropy values than another one for the same files, tokenizers and n-values is preferable. We therefore computed all the combinations of n-values, smoothing techniques and tokenizers with the aim of identifying the most appropriate configurations.

Our results are consistent across all tokenizers and n-values: they show that the Modified Kneser Ney smoothing is the most appropriate which is in line with what was found by the NLP community [13]. Although the difference with Kneser Ney is thin, it is statistically significant (using Wilcoxon signed-rank test) and has $\hat{A}_{12}$ values in the range from 0.50 to 0.53. Figure 1 presents an example of our data for the case of the AST breadth-first tokenizer and n equal to 4.

Having shown that the Modified Kneser Ney smoothing technique is the best one, we turn to see the impact of choosing an appropriate n-value. Again, we have similar trends for the n-values across the different tokenizers. However, we observe that some tokenizers do converge faster than others. For instance, when increasing n from 5 to 6, we observe that the reduction of entropy is way smaller for the UTF tokenizer than for the depth-first tokenizer; the former thus converges faster than the latter. Figures 2(a) and 2(b) demonstrate these results. Although we also observe that benefits of using n values higher than 4 is small, all the n-values result in statistically significant differences. In particular, all the tokenizes have $\hat{A}_{12}$ values in the range from 0.54 to 0.65 when comparing $n = 4$ with $n = 5$. These drop to the range 0.53 to 0.57 when comparing

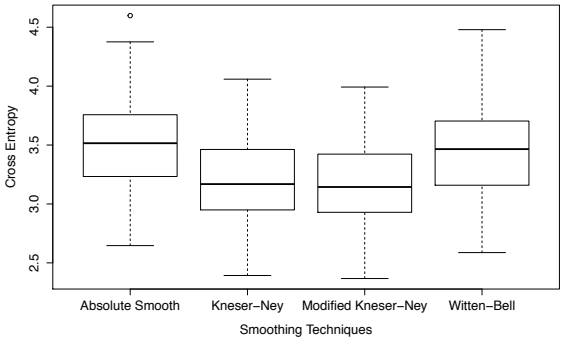
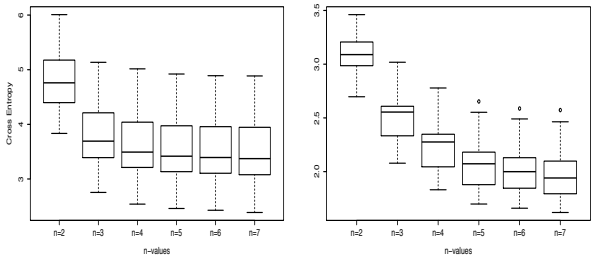


Fig. 1. Cross entropy of source code files when using different smoothing techniques for AST breadth-first tokenizer and $n = 4$.



(a) UTF tokenizer (b) AST depth-first tokenizer

Fig. 2. Cross entropy of source code files when using different tokenizers with Modified Kneser Ney smoothing for n-values in the range n=2..7.

the $n = 5$ with $n = 6$. Therefore, the general most appropriate choices are the $n = 4$ or $n = 5$. It is noted that the AST depth first and the "Java Parser" tokenizers are the only ones that continue to improve (slightly) beyond $n = 6$.

B. RQ2: Tokenizer Correlations

Our second research question examines the correlation between the cross-entropy values returned by the 8 tokenizers for n size equal to 5. We also consider the correlation between these values and the number of lines of code (LoC). We computed the correlation coefficients for all files by considering every pair of tokenisers (and LoC). Table III summarizes the results. It gives the median of the coefficients over all projects for each pair. The upper triangle of the table records the Pearson coefficients, whereas the lower one is about the Kendall's tau coefficients. A higher coefficient means stronger correlation. All coefficients are statistically significant with a p-value lower than 0.05 in every case.

TABLE III
CORRELATIONS BETWEEN TOKENIZERS AND NUMBER OF LINES OF CODE. UPPER (RESP. LOWER) DIAGONAL GIVES, FOR EACH PAIR, THE MEDIAN OF THE PEARSON CORRELATION COEFFICIENTS (RESP. KENDALL'S TAU COEFFICIENTS) OVER ALL PROJECTS.

	LoC	UTF	UTFw	JP	JPw	DF	BF	PDF	PBF
LoC		0.52	0.32	0.31	0.29	0.25	0.15	0.24	0.17
UTF	0.52		0.83	0.78	0.77	0.71	0.60	0.74	0.64
UTFw	0.32	0.66		0.90	0.90	0.88	0.79	0.91	0.81
JP	0.29	0.60	0.77		0.99	0.89	0.81	0.90	0.79
JPw	0.29	0.60	0.77	0.93		0.91	0.82	0.92	0.79
DF	0.30	0.57	0.73	0.76	0.79		0.85	0.96	0.78
BF	0.14	0.47	0.62	0.65	0.66	0.70		0.85	0.94
PDF	0.28	0.58	0.76	0.77	0.78	0.84	0.69		0.85
PBF	0.15	0.48	0.62	0.64	0.64	0.64	0.80	0.70	

A first observation is that Pearson coefficients are higher than Kendall's coefficients for each pair of tokenizers. This indicates that large differences in cross-entropy are more likely to lead to an agreement between the tokenizers than smaller differences. The strongest correlations are in the case of JP with the JPw. If we exclude this case, the Pearson correlations are in the range of 0.60 to 0.96, while the Kendall ones are in the range of 0.47 and 0.84.

Another general observation is that the correlation between LoC and the cross-entropy values is generally weak (< 0.33). The only exception is the UTF tokenizer, which has a correlation with LoC of 0.52. This is due to comments. Indeed, UTF is the only tokenizer that tokenizes comments as any English text. The JP tokenizer includes comments as well, but considers a block of comments (e.g., the Javadoc of a method) as a single token. The impact of comments for this tokenizer is thus limited, as witnessed by the strong correlation between JP and JPw (Pearson 0.99; Kendall 0.93). Given the difference in the way code and natural language are written, it is expected that comments significantly increase cross entropy. Moreover, the number of comments is likely to increase with the number of lines of code. We also see that AST tokenization further reduces the correlation with LoC (≤ 0.25). Indeed, the number of tokens depends on the number of AST nodes, which is not necessarily proportional to LoC.

Interestingly, we observe that the differences between the ways the AST is visited, in a breadth-first or depth-first manner, plays an important role, as it gives the lowest correlation values. Generally, the breath-first tokenizers give in all cases, lower correlations than their depth-first counterparts. This indicates that breadth-first AST tokenizers capture the most different information than any other tokenizer. In other words, the disagreement with the other tokenizers is higher. This can be explained by the fact that the other tokenizers have inherently different views of the code, i.e., structure-oriented versus sequence-oriented.

We also observe that there are no significant differences between AST tokenizers (regardless of the visit strategy) and their respective pruned variants. Redundant nodes thus have a limited impact on the captured information. More generally, it might imply that how the AST is constructed is unimportant, although this must be confirmed by additional experiments with alternative parsing tools.

Taken together, our results suggest that tokenizers indeed judge code files differently. They tend to agree on the majority of the cases but still they tend to disagree on a significant number of cases.

C. RQ3: Impact on results: Bug Analysis

Having confirmed that tokenizers from different groups learn (largely) different things, our third research question

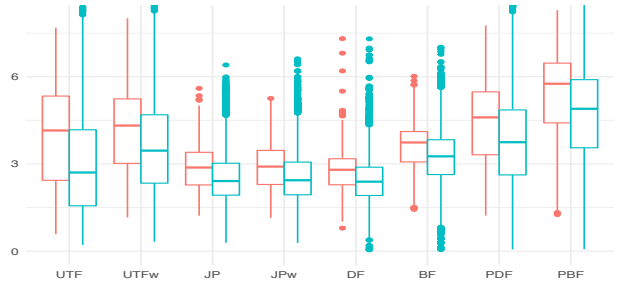


Fig. 3. Entropy over all projects of buggy (red) and non-buggy files (green).

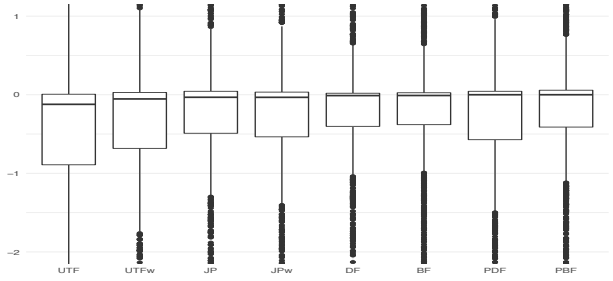


Fig. 4. % of difference in cross entropy between the buggy version of a file and its fixed one per tokenizer.

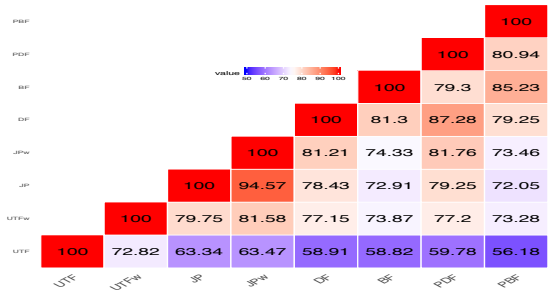


Fig. 5. Percentage of agreement between tokenizers. The values represents the ratios of files judged similarly by the tokenizers (increase or decrease)

regards their possible impact on the findings of an experimental study. To answer this question, we investigate the hypothesis that bugs are linked with naturalness. We do so by checking whether unnatural files are more likely to be buggy than the natural ones. In case we find significant differences, we can conclude that a link between bugs and naturalness exists. However, this link might simply be the result of other (unknown) factors (such as the size of files, the defects' location or others). In other words, we need to show that entropy is linked with both presence and absence of bugs. To control for arbitrary factors, we check whether fixing a file reduces its entropy. In case we observe a reduction in most of the case, then we have strong evidence supporting our hypothesis, while in the opposite case we do not.

Figure 3 reports the results for all considered project releases and tokenizers. From these we can make two main observations. First, buggy files have a higher entropy than their non-buggy counterparts regardless of the tokenizers used. This provides a first indication that our hypothesis might hold (this result is inline with the results of Ray et al. [3]). Second, UTF and pruned AST-based models (PDF and PBF) present the largest variance in entropy which could make them targets of interest when using them as prediction models.

Figure 4 presents the results obtained when studying the entropy differences between the buggy and the fixed versions of our files, following the procedure described in Section V-C3. We observe that in many cases, the cross entropy is indeed slightly reduced (values are below 0) after the fix process. However this does not for a clear majority (approximately 50% of the files have values higher or equal to zero, median values are 0). This means that fixing a file might reduce the entropy or might not, which in turn indicates that bugs appear in files that are unnatural but naturalness is not necessarily linked with the presence of bugs. Considering the starting point of naturalness (i.e., developers tend to write code that is repetitive, hence more natural), this means that bugs are located in files further from developers' usual comfort zone.

Perhaps the most interesting observation is that the differences are more accentuated in the case of UTF tokenizers. These are the only models having median values below 0. As all other tokenizers have their median almost at 0, we can conclude that one can get evidence supporting our hypothesis, only by using the UTF tokenizers. We also statistically examine the differences and find that they are statistically significant with an effect size, close to 42% (when comparing the UTF tokenizers and the others). Interestingly, these cases are the only ones with both statistical significance and effect size differences. The difference between JP, BF and DF are not statistically significant whereas the difference between BF, DF, PBF and PDF are significant (though with negligible effect size). To make these results clear, Figure 5 shows the level of agreement (on the impact of fix) between the tokenizers, i.e., how frequently every pair of tokenizers agree that fixing a file results in reduced or increased entropy. From these results we see that tokenizers largely disagree on their judgements.

The above results imply that the closer, to human perspective (unprocessed code), the used tokenizer is, the more robust the n-gram model is in detecting the effects of a fix. Thus, only UTF tokenizers are robust in this regard. This is also interesting as the UTF tokenizers are not the ones with the lowest entropy.

To summarise, we found that tokenizers have the potential of changing the conclusions of a research study. We demonstrated that only 2 out of the 8 tokenizers are robust at detecting (as they should) the differences between buggy and fixed files. Therefore, researchers need to be cautious that their conclusions may change if they use different tokenizers. Moreover, our data suggest that the most prominent choice of tokenizer for bug identification is the UTF.

D. RQ4: Setting the 'unknown threshold'

In n-gram model related literature, a specific parameter called unknown threshold is often evoked, but has never been examined. Increasing this parameter may make the entropy lower but at the price of a less general model.

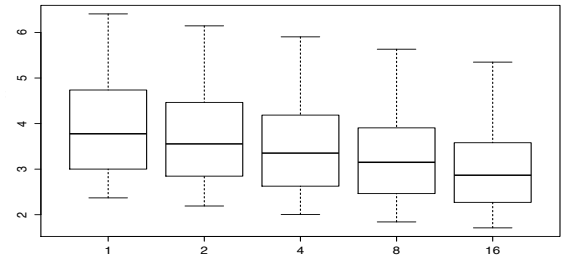
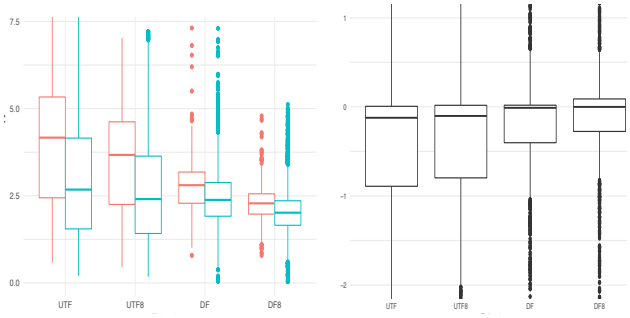


Fig. 6. Effect of threshold k on the cross-entropy



(a) Difference in entropy between buggy file(red) and non-buggy(green)

(b) % of difference in cross entropy between the buggy version of a file and its fixed one per tokenizer

Fig. 7. Effect of the unknown threshold for the UTF and DF tokenizers when k are equal to 1 and 8

Figure 6 presents the results we obtain in the experiment of RQ1 while observing five different values of k. We observe a huge decrease in entropy as k increases. This means that the model copes better with low-count tokens. For source code, this could be explained by the fact that the model is removing variable or function names that are barely used. While this could be interesting in some situations (e.g., when one is interested in general patterns or trends), it can have a negative impact on naturalness-based studies. In Figure 7, we present a comparison of the values obtained with two different k, i.e., $k = 1$ and $k = 8$ for two of our tokenizers.

In Figure 7(a) we observe the reduction in entropy between the two values of k, yet the difference between buggy files and non-buggy ones is still clear. However, Figure 7(b) reveals that the reduction of entropy after a fix is compromised when increasing k. This is more interesting in the case of AST depth-first tokenizer, where for $k = 8$ the entropy increases in more cases than for $k = 1$. This can be explained by the fact that a high entropy is caused by unlikely tokens which are removed when using a higher k, gathering them under a common, more likely one. This, in turn, reduces the opportunity to observe the effect of a fix: If the new token introduced by the fix replace an unknown one but has a lower probability than the unknown the entropy could increase in some case.

VII. Threats to validity

The generalization of our result is a usual threat to validity of experimental studies. We used Java projects from Apache Commons, which may not be representative. Similarly, our results might not hold on other programming languages. We choose the Apache Commons to gather a large variety of projects with different functionalities. Moreover, Apache is a large organization and follows a similar development process with many other organizations.

Similarly, we showed that tokenizer impact the n-gram models in Java. We expect a similar result on other languages as the basic differences between the sequences of tokens and ASTs appear in all languages. However, we still do not know whether n-gram models are similarly sensitive when using other languages.

Another threat to validity regards our toolchain. We integrate different external tools to perform this experiment, hence an error in one of those tools or in our integration would influence the result. To mitigate this, we only rely on tools that are known to be reliable. Terrier, from which we use their UTF tokenizer, is a well-known information retrieval framework. We also carefully tested our tokenizers to ensure of their behaviour. Java Parser is also used by more than 50 libraries and 100 projects on Github, and many companies use it and update it regularly. Nevertheless, as all tokenizers were carefully integrated (using their documentation) and tested, we do not consider this threat as important.

KYLM is widely used for comparing many recent n-gram approaches, e.g., the work of Pickhardt et al. [40]. Since this tool considered as relevant by the NLP community, we believe it is trustworthy. Of course we carefully analysed and tested it before using it. To further reduce these threats we make our toolchain and data available [41].

A threat related to construct validity regards the way we built our bug dataset (used in answering RQ3). The dataset used for this research question is automatically generated using git commit messages and the JIRA Apache bug tracker. Thus, imprecise information or wrongly categorized issues in the tracker or misleading commit messages could generate noise in our data. However, given the strict guidelines used for the development and reporting of bugs in Apache Commons project, we believe that this could only be the case for a small percentage of the files. Therefore, the influence on our results would be relatively small.

Conclusion

Research on naturalness of code is focussing on assisting software engineering tasks using n-gram models. However, the use of such models require setting a number of parameters. We perform a study and show that the choice of smoothing, tokenizer, unknown threshold and n values can impact the predicting ability of the models. We demonstrate that the Modified Kneser-Ney smoothing technique performs best, while n-values equal to 4 or 5 are generally appropriate. We also show that the closer, to human perspective (unprocessed code), the underlying representation is, the more robust the n-gram model is. This suggests that the most prominent choice of tokenizer (wrt to bug identification) is the UTF one. Finally, we demonstrated with an experiment that researcher can come to wrong conclusions if they do not properly tune their models.

REFERENCES

[1] K. S. Jones, *Natural Language Processing: A Historical Review*. Dordrecht: Springer Netherlands, 1994, pp. 3–16.

[2] 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[3] 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*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439.

[4] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 708–719. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970341

[5] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, 2014, pp. 252–261. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597102

[6] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 311–322. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330219

[7] X. Hu, Y. Wei, G. Li, and Z. Jin, "Codesum: Translate program language to natural language," *CoRR*, vol. abs/1708.01837, 2017. [Online]. Available: http://arxiv.org/abs/1708.01837

[8] W. Ling, P. Blunsom, E. Grefenstette, K. Moritz Hermann, T. Koisk, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 03 2016, pp. 599–609.

[9] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017.

[10] S. Simone, L.-V. Mario, O. Rocco, and P. Denys, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958

[11] 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*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[12] C. E. Shannon, "Prediction and entropy of printed english," *Bell System Technical Journal*, vol. 30, pp. 50–64, Jan. 1951. [Online]. Available: http://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf

[13] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Comput. Speech Lang.*, vol. 13, no. 4, pp. 359–394, Oct. 1999. [Online]. Available: http://dx.doi.org/10.1006/csla.1999.0128

[14] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[15] I. H. Witten and T. C. Bell, "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression," *IEEE Trans. Inf. Theor.*, vol. 37, no. 4, pp. 1085–1094, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/18.87000

[16] H. Ney, U. Essen, and R. Kneser, "On structuring probabilistic dependences in stochastic language modelling," *Computer Speech & Language*, vol. 8, pp. 1–38, 1994.

[17] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. I, Detroit, Michigan, May 1995, pp. 181–184.

[18] F. Jelinek and R. L. Mercer, "Interpolated estimation of markov source parameters from sparse data," in *In Proceedings of the Workshop on Pattern Recognition in Practice*, Amsterdam, The Netherlands: North-Holland, May 1980, pp. 381–397.

[19] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *ACL*, 2017.

[20] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli, "Will they like this?: Evaluating code contributions with language models," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 157–167.

[21] H. Liu, "Towards better program obfuscation: Optimization via language models," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 680–682. [Online]. Available: http://doi.acm.org/10.1145/2889160.2891040

[22] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," pp. 39–40, 07 2016.

[23] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 49–65.

[24] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2017. New York, NY, USA: ACM, 2017, pp. 35–42.

[25] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 572–588. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814309

[26] 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*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 532–542.

[27] 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*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[28] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[29] A. Sharma, Y. Tian, and D. Lo, "NIRMAL: automatic identification of software relevant tweets leveraging language model," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 449–458.

[30] S. Yadid and E. Yahav, "Extracting code from programming tutorial videos," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2016. New York, NY, USA: ACM, 2016, pp. 98–111. [Online]. Available: http://doi.acm.org/10.1145/2986012.2986021

[31] W. A. Gale and K. W. Church, "Poor estimates of context are worse than none," in *Proceedings of the Workshop on Speech and Natural Language*, ser. HLT '90. Stroudsburg, PA, USA: Association for Computational Linguistics, 1990, pp. 283–287.

[32] ——, "What's wrong with adding one," in *Corpus-Based Research into Language. Rodolpi*, 1994.

[33] S. M. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer." *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 35, no. 3, pp. 400–401, 1987.

[34] J. Saraiva, C. Bird, and T. Zimmermann, "Products, developers, and milestones: How should i build my n-gram language model," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 998–1001. [Online]. Available: http://doi.acm.org/10.1145/2786805.2804431

[35] A. Foundation. (2017) Apache commons. [Online]. Available: http://commons.apache.org

[36] Terrier. (2017) Terrier open source search engine. [Online]. Available: http://terrier.org

[37] J. Parser. (2017) Java parser github. [Online]. Available: https://github.com/javaparser/javaparser

[38] G. Neubig. (2017) Kyoto language modeling toolkit. [Online]. Available: https://github.com/neubig/kylm

[39] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, ser. Alan Apt series. Pearson Education, 2003. [Online]. Available: https://books.google.be/books?id=0HYhAQAAIAAJ

[40] R. Pickhardt, T. Gottron, S. Staab, P. G. Wagner, T. Speicher, and T. Gbr, "A generalized language model as the combination of skipped n-grams and modified kneser-ney smoothing," in *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

[41] M. Jimenez, M. Cordy, Y. L. Traon, and M. Papadakis, "Tuna: Tuning naturalness-based analysis," in *34th International Conference on Conference on Software Maintenance and Evolution, ICSME 2018, September 23 - 29, 2018, Madrid, Spain*, 2018.