# LEVERAGING NEURAL LANGUAGE MODEL FOR AUTOMATED CODE QUALITY ISSUE IDENTIFICATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

The usefulness of machine learning techniques for understanding source code and assisting with software engineering tasks have been demonstrated by recent progress in the community. More specifically, language models (LMs) on code have been originally developed based on n-grams. One of the limitations of such models is the Out-Of-Vocabulary issue which abounds in code-related applications. Recent advances in deep learning have provided new and more powerful tools for source code modeling. For instance, RNN and transformer based models have been built to learn embeddings of codes for various downstream tasks such as code completion, summarization and translation. However, the power of neural language models has not been well leveraged for improving the quality of code. We develop a single framework based on the unifying concept of neural language modeling to identify four different types of code quality or stylistic issues that may hurt the readability, understandability and maintainability of code. We have built an automated tool to spot such issues in real code repositories from GitHub, and the recommendations by our tool are well accepted by developers. Finally, we show that by fine tuning the language model trained in cross-project setting on a specific repository, the resulting "local" model not only has better model performance but can help reduce false positives.

## 1 INTRODUCTION

It is now well-known that source code exhibits strong statistical regularities known as "naturalness" Hindle et al. (2012), meaning that code written by humans is often repetitive and predictable. As such, there has been emerging research and applications of machine learning techniques in source code and software development. In particular, language models, which are probabilistic distributions over sequences of tokens, have been built to find code related issues such as variable misuse and have enabled new tools for automating software development tasks such as code generation and completion. The initial statistical language models on source code were primarily based on n-grams of code tokens. These models have provided important insights on the "naturalness" of source code, including: 1) Source code is even more repetitive and predictable than natural language. If measured by cross entropy, code has lower cross entropy than natural language; 2) Buggy code is less natural than correct code Ray et al. (2016), i.e. the cross entropy of buggy code is higher. While these insights are important, there are several limitations of these models such as limited corpus size, inability to handle out-of-vocabulary tokens, and limited context.

Recent research has introduced more advanced models on source code such as RNN-based models (White et al. (2015); Dam et al. (2016); Rabinovich et al. (2017)) and transformer-based models (Alon et al. (2019); Hellendoorn et al. (2020); Chirkova & Troshin (2020); Liu et al. (2020)). These models have addressed some of the limitations above and have shown better performance than traditional models. While these recent progress on language modeling and deep learning have brought our understanding of source code to the next stage, there has been little-to-no work on identifying code quality related issues using deep learning and providing *actionable recommendations* on improving code quality leveraging language models. For example, developers cannot easily act on recommendations such as “This segment of code has high cross-entropy”, because it is not specific about the issues associated with this and how to improve the code. On the other hand, recommendations such as “This segment of code is repetitive. Consider refactoring the code” are actionable.

In addition, previous works that utilize machine learning to detect code-related issues often require labeled dataset depending on specific problems to train the models, while our language model based approach identifies code issues in an un-supervised manner that does not need task-related labels but only source code to do training and inference.

At a high level, we build two types of language models that work on different granularities: global model built at corpus level and local model built at repository level. For both cases, we implement two kinds of models: full-model that takes raw code text as input and skeletal model that replaces identifiers and literals with placeholders. Each type of model plays a role in providing code quality recommendations, and we will elaborate more in the paper.

The contributions of the paper are the following.

1. We have built an automated tool to identify several code quality or stylistic related issues with low false positive rate (precision $\approx 66.4\%$), including the following.
   (a) Identification of token-level code quality issues (such as typos and contextually-inappropriate variable names) along with corrections. The use of sub-word tokenization in neural language models helps deal with Out-Of-Vocabulary problem.
   (b) Identification of unnatural (i.e., terse or cryptic) code which is usually difficult to read and understand.
   (c) Identification of highly repetitive code which is prone to errors and difficult to maintain.
   (d) Identification of long and structurally complex code which is difficult to understand and maintain.
2. The model trained on the full corpus that contain a large number of repositories (i.e. cross-project setting) is called *global* model. In addition, we fine-tune the global language model on specific repositories, which results in "local" language model that is a customized version for specific repositories. We show that such local models could provide extra benefits in complementary to the global model such as suppressing false positives.

## 2 BUILDING THE LM

We use GPT-2 based language model. Since GPT-2 models are well-understood, the details of the model and training are provided in Appendix A.

We train two versions of models: a *Full model* and a *Skeletal Model*. The full model takes the original source code as inputs for training, and the only difference for the skeletal model is that it replaces all the identifiers and literals in code with special tokens. As identifiers such as variable/method names and literals such as text strings in code could bring lots of uncertainties and variations, we would expect the skeletal model to purely focus on the structure of code. For some applications such as identifying unnatural code or repetitive code that we will discuss in later sections, we would only want the model to focus on the code structures and exclude the noises brought by identifier names and literals. To implement the skeletal model, we used a Python based parser, called *Javalang*, for Java code to locate all identifiers and literals, and replace all identifiers with token '|IDENTIFIER|' and all literals with token '|LITERAL|'. All the other settings and training schemes are the same for both models except that we added these two special tokens into the vocabulary of the model.

The quality of the model, in terms of cross entropy compares well with published models. See Appendix B for details.

## 3 IDENTIFICATION OF CODE QUALITY ISSUES

Code quality issues are not necessarily bugs that prevent proper functioning of the code, but also refer to any code issues or code smells that can lead to low readability, understandability and maintainability. Low code quality could increase tech debt and harm the health of software development cycle in the long term Mamun et al. (2019). Moreover, it also makes the code prone to errors and severe system failures. Many automated tools such as static analyzers have been

Table 1: Identification of code quality issues

| Issue | Model | Metric | Logic |
|---|---|---|---|
| Token error | Full | Token probability | $p(t) < p_0$ and $p(t') > p'_0$ |
| Unnatural code | Skeletal/Full | Method cross-entropy | > threshold |
| Repetitive code | Skeletal | Method cross-entropy | < threshold |
| Long and complex code | Skeletal/Full | Method log probability | > threshold |

developed to find bugs in code, but much fewer tools are aimed to improve the quality of code, especially in a data-driven way. Here we leverage neural language models we trained to identify several types of code quality issues.

For an input code to be analyzed, we calculate the *token probability* for each sub-token and *cross entropy* at method level. In Table 1, we provide a high-level summary of the model, metric and approach we use to identify various types of code quality issues. We have made use of several outputs from the language model, including token probability, method cross-entropy and log probability. Note that the distributions of these metrics we obtained during training on the large code corpus provide reliable baselines for determining the thresholds that are used for the code quality issue identifications. In the following sections, we will discuss each type of issue in more details.

### 3.1 TOKEN-LEVEL ERRORS

The language model assigns each code token $t$ (i.e. sub-word) a probability $p(t)$. Intuitively, a low $p(t)$ indicates the model infers $t$ is less likely to appear in the particular position, or the token is "surprising". Meanwhile, the model is able to assign a probability for all other tokens in the vocabulary, with the sum of all token probabilities equal to 1. As such, the model may find another token $t'$ with higher probability $p(t')$ at the same position. We seek for the cases that $p(t)$ is lower than a lower-end threshold $p_0$ (e.g. $0.01$) and $p(t')$ is higher than a higher-end threshold $p'_0$ (e.g. $0.95$), indicating the model is very confident that $t'$ instead of $t$ should be used at the position. This logic enables us to identify potential token-level errors in code. We use the full model to identify such issues as we need the specific identifier names and literals.

Using this approach, we have identified many token-level errors in our test packages, mainly on inappropriate variable/method names and typos. The following is an example we identified from Github repositories. For simplicity we have skipped irrelevant codes.

```
public TableCellEditor getCellEditor(int row, int column){
    ...
    }else if(pname.equals("valign")){
       editor = valingEditor;
    }else if(pname.equals("align")){
       editor = alingEditor;
    }
    ...
}
```

In this example, in the variable `valingEditor` the probability for the sub-token `ing` after `val` is only $2.99e^{-6}$, while the model predicts the correct sub-token `ign` with a probability of 0.983. Similar case applies for the variable `alingEditor` as well. In summary, the recommendation is to use `valignEditor` and `alignEditor` instead of the highlighted variable names. Note that such recommendations cannot be provided by using dictionaries.

For this example, we have investigated the original source code from GitHub and the developer has used the inappropriate variable/method names throughout the code for multiple times without being aware of it, and such issues would not be found during compilation. Apart from inappropriate names, we also detected various of typos in log messages.

It is worthwhile to note that the use of sub-word tokenization (BPE in this case) is also necessary for all such findings, which the traditional n-gram model would not be able to capture.

### 3.2 UNNATURAL CODE

A code snippet is "unnatural" means the code is written in an uncommon or weird way compared to what most developers would do. While it may not directly lead to errors, unnatural code could often make the code less readable and understandable. From the language modeling perspective, unnatural code means the language model finds the code sequence surprising and assigns the code sequence a low probability, or high cross-entropy and perplexity. Hence, we determine Java methods with the perplexity higher than a certain threshold as "outliers" and thus unnatural. As we discussed, identifiers such as variable/ method names and literals could often lead to higher perplexity because they are often unpredictable. We have since used the skeletal model in our tool to mainly focus on the unnaturalness and strangeness of code structures. However, one can also use the full model with the same approach to identify unnatural code related to identifier names, such as uncommon and less interpretable variable/method names.

With this approach, we are able to identify diverse types of unnatural codes in our test packages. The interpretation of "unnatural" code has been always a challenge, and here we provide more clarity by manually examining the findings from our model and discussing several typical scenarios that make the code unnatural.

#### COMPLEX LOGIC

Methods that contain statements with complex logic are flagged by the model because they are much less likely to happen and thus more difficult to predict, which result in higher perplexity. For instance, some statements contain a mix of multiple bit operations, comparison, ternary operator and method calls, which makes the code logic difficult to read and understand.

In such cases, our tool will suggest refactoring the code to make it concise, more readable and understandable. For instance, using multiple separate statements for different operations could resolve the code quality issue.

#### FORMATTING ISSUES

Our tool found various formatting issues in code with high method perplexity, including missing or extra spaces, unexpected line breaks and indentations. The high perplexity of such codes often originates from extremely low probability of those special characters.

#### SWAPPED OPERANDS

A recent study Casalnuovo et al. (2020) has found that even simple code transformations such as adding/removing parentheses, shuffling variable names and swapping operands could make the code more "surprising" and harder for human to comprehend. Interestingly, our model is able to capture such cases. For instance, most developers would write relational operations such as `a > 0` in code, while a mathematically equal expression `0 <= a` is much less common. Another example is nullness checking. In most cases developers use `variable != null` while using `null != variable` is considered surprising by the model.

### 3.3 REPETITIVE CODE

Both our previous discussion and previous studies have suggested that it is generally desired to have codes with low perplexity, so the code is written in a natural and common way. However, we discovered that it is not always optimal to have code with extremely low perplexity. In fact, we firstly found that code with very low perplexity (i.e. too predictable) usually contain repetitions of similar or same structures within the code. This is because once the model has learned the code structure that initially appears in the code, if the same structure appears in subsequent parts of the code, the model would recognize it and be very confident in predicting it with high accuracy. The more times the same structure repeats in code, the more confident the model would be in predicting it, hence lower perplexity of the code. Based on this finding, we determine Java methods with perplexity lower than a certain threshold, which is usually a very low number, as outliers and flag such methods. In order to reduce noises brought by identifier and literals and only focus on repetitions of code structures, we have only used the skeletal model for identifying this type of code issue. Large number of repetitions

in code, especially within a method, could make the code difficult to maintain and prone to errors, including copy-paste errors, so we would recommend developers refactor the code for conciseness and better maintainability.

In particular, our model is capable of identifying long code structures (i.e. multiple lines) with high repetitiveness within a code snippet, which provides the functionality of detecting code clones using our tool. This indicates that although not specifically designed for detecting code clones, our tool is capable of identifying such issues in a way that does not require sophisticated program analysis.

### 3.4 LONG AND COMPLEX CODE

A common type of code quality issue is the code being too long or too complex. Traditionally, there are 2 commonly used measures of code complexity: Line of Code (LOC) and Cyclomatic Complexity (CC). While both are useful metrics, they have limitations as they both focus on a single aspect of code complexity: LOC only captures the size of code and CC captures the number of decisions. For instance, a long code snippet may be structurally and logically simple and straightforward to understand. In our tool, we use the Log Probability (LP) of a code sequence to measure its complexity. The LP is defined as $-log(P(t_1, t_2...t_N))$, where $P(t_1, t_2...t_N)$ is obtained from our neural language model. LP captures both the size (determined by number of tokens) and the unpredictability (determined by token probability) of a code snippet, thus can be interpreted as accumulated unpredictability. A high LP indicates the code snippet is overall long and unpredictable. Based on this intuition, we determine a particular method as an outlier if its LP is higher than a preset threshold based on the overall distribution of this metric. As expected, we found the outliers are mostly both long and structurally complex methods that are preferred to be refactored for better readability and understandability.

### 3.5 SUMMARY

As shown in the sections above, we are able to identify four types of actionable code quality issues by leveraging our neural language model. It is important to note that we have used *a single framework based on the unifying concept of language modeling to detect diverse types of code related issues*. As comparisons, in existing works each type of issue often requires a specialized detector to identify.

## 4 HUMAN EVALUATION

A key question to answer is the usefulness of our language model based approach described in the previous section on real codebases. Unlike other machine learning models in Computer Vision and Natural Language Processing, evaluating the performance of a machine learning model on identifying code related issues is rather challenging due to lack of labeled benchmark datasets. Many publicly available datasets on code related issues are synthesized code examples with deliberately created code issues, while the performance of automated static analysis tools could damp significantly when dealing with real-world codes Raychev (2021); Pradel & Sen (2018). Moreover, since our tool is the first-of-its-kind in automatically identifying such code quality issues, there are no available datasets we could directly utilize to compare with previous studies. Thus, to evaluate the effectiveness of our approach, we run our tool on open-source GitHub codes and have the recommendations provided by our tool manually reviewed by independent (non-author) software developers.

To conduct this experiment, we run our tool on 5k randomly selected GitHub projects that are separate from our training dataset[1]. This generates more than 1k alarms of code quality issues that our tool recommends improvements. The thresholds used in this experiment are empirical values from our study based on the distributions learned from the training packages. For instance, we can use the 95% percentile of the method cross-entropy distribution as a threshold for determining unnatural code. Specifically, for this experiment, we choose the thresholds to seek for a good balance between the accuracy and coverage of the tool. Specifically, we use $p_0 = 1e^{-5}$ and $p_0' = 0.99$ for token-level error detection. We use the cross-entropy threshold of 2 from the skeletal model for

[1]The training dataset statistics are provided in Appendix A.2

determining unnatural code, and 0.3 for determining repetitive code. The log probability threshold for detecting long and complex code is 500. We randomly select a list of recommendations given by our tool that cover various code quality issues we have discussed, and have a group of 11 independent and experienced software developers review and determine if the recommendations are useful. We end up with 177 samples labeled by the reviewers, and the results are shown in Table 2. Note that although it is theoretically possible that certain code example may be flagged as multiple types of issues, we do not observe such cases in the 177 labeled samples.

Table 2: Recommendation Evaluation

| Category | Useful | Not useful | Not sure | Total | Accuracy |
|---|---|---|---|---|---|
| Token error | 9 | 0 | 2 | 11 | 100.0% |
| Unnatural code | 34 | 22 | 20 | 76 | 60.7% |
| Repetitive code | 34 | 23 | 6 | 63 | 59.6% |
| Long and complex code | 18 | 3 | 6 | 27 | 85.7% |
| Overall | 95 | 48 | 34 | 177 | 66.4% |

Out of 177 samples, 95 are rated as useful findings and the reviewer agrees a code change or refactoring could be helpful for better code quality. 48 findings are rated as "Not useful", and 34 are rated as "Not sure", meaning that the findings are potentially useful but the reviewer is not very sure. Excluding the 34 ambiguous "Not sure" ratings, this gives an accuracy (i.e. precision) of 95/143 = 66.4%, suggesting that about 2 out of each 3 findings from our tool is helpful. We also calculated accuracy for each category in the same way. All categories of recommendations have accuracy higher than 59.6%, and token level errors that are more objective has the highest accuracy of 100.0%. Note that due to lack of ground truths, namely, the number of real code quality issues in the dataset is unknown, the recall metric is not applicable here. We observed clear subjectiveness from the reviewers in the judgment of unnatural code, which is also reflected by as many as 20 'Not sure' rating. In addition, the 48 'Not useful' ratings from all categories include cases that reviewers acknowledge the code quality issues identified by the tool but do not think there are good ways of refactoring the code. These preliminary results demonstrate the effectiveness of our language model and our approaches when detecting code quality issues in the wild.

Real code examples of detections can be found in Appendix C.

## 5 Local Model

The language models we have discussed in above sections (both the full and skeletal version) are trained on 19k GitHub repositories, which we call the *global models*. Previous study Tu et al. (2014) has shown that source code exhibits *localness*, i.e. local regularities specific to each programming unit (project, file, module etc.). For instance, each project may have its own preferred use of identifier names which may differ from preferences learned from the whole training corpus. Such local regularities are often overlooked by the global model as it is trained in a cross-project setting. To address this limitation, traditional n-gram language models have used caches to memorize n-grams found in locality. Here, to adapt our global model to a new repository, we take the strategy of continuously training the global model on the codes from the new repository for more steps so the parameters and weights of the model is further optimized. A particular use case of local model is identifying issues during pull requests. For instance, once the global model has been fine-tuned on a snapshot of a code repository (i.e. an early revision), the resulting local model can be used to identify issues as they arise with subsequent revisions.

The local models are expected to have better performance in terms of cross-entropy as they are the fine-tuned versions Tu et al. (2014); Karampatsis et al. (2020), which is also demonstrated in our experiments shown in Appendix D. Here, we show that the local model could also provide extra benefits on the identification of code quality issues. For instance, we observed that certain false positives found by the global model are due to local regularities that the model is unable to capture from other corpus. Such false positives could be effectively suppressed by the local models. We discuss such cases for both token-level errors and unnatural codes in the following subsections.

### TOKEN-LEVEL RECOMMENDATIONS

Identifier names are a large part of local regularities that are often specific to repositories. As they are the main focus of the token-level recommendations, the local model is particularly more effective than the global model for this category. Consider this real example from the `netbeans` repository below.

```
public HgTag (String name, HgLogMessage revisionInfo, boolean local, boolean removable) {
  this.revisionInfo = revisionInfo;
  this.name = name;
  this.local = local;
  this.canRemove = removable;
}
```

Based on the method inputs, the first 3 assignment statements and what it has learned from large corpus, the global model (full version) reasonably infers that in the highlighted statement, `this.` should be followed by the boolean input `removable` (probability = 0.99 for the sub-token `rem`) instead of `canRemove` (probability = $5.4e^{-6}$ for the sub-token `can`). The extremely low probability of the latter is because the global model has never seen the variable `canRemove` in its training. Instead, the local model trained on this package has seen the variable `canRemove` in other methods, so is less confident that `this.` must be followed by `removable` (probability = 0.79). This recommendation is thus suppressed by the local model by the threshold setting of our tool, which avoids flagging a false positive.

### UNNATURAL CODE

Code snippets that are inferred as unnatural by the global model are generally due to they are different from common practices in the large training corpus. However, they may be "natural" within a specific repository. Consider the example shown below.

```
JButton getInfoButton() {
  assert withButtons;
  return infoButton;
}
```

The method is identified by the global skeletal model as unnatural with a high cross-entropy of 2.25. Apart from the assert statement, the model also finds it surprising to start the method with an identifier name `Jbutton` without any other modifiers and visibility setting (although it is generally not always required). We dive deep into the repository and find that the method signature is consistent with other methods in the same file, indicating that this is by design of the developer and not a real issue. Instead, the local model has seen other methods similar to this one in the same repository so finds it less surprising, assigning a cross-entropy of 1.61, which is no longer regarded as an outlier in our tool.

## 6 RELATED WORK

There are several ground work on studying source code with language models. Hindle Hindle et al. (2012) firstly modeled source-code with n-gram of tokens and discovered the repetitive nature of code. A larger n-gram language model is trained in Allamanis et al. Allamanis & Sutton (2013) on GitHub code in a cross-project setting. Such language models have enabled new tools for software engineering tasks. Ray et al. (2016) has found buggy code has lower probability than correct code on average, suggesting LM can be used for detecting code related issues. N-gram LMs has also been used for syntax error checking Campbell et al. (2014), code completion Raychev et al. (2014); Franks et al. (2015) and bug detection by identifying low probability sequences Wang et al. (2016).

Recent progress in Natural Language Processing (NLP) has brought new power for modeling source code with deep neural networks on large scale code corpus, see Allamanis et al. (2018) for a survey. Advanced machine learning models and architectures have been leveraged in language models for source code, including RNN White et al. (2015), LSTM Dam et al. (2016); Rabinovich et al. (2017), Pointer Network Li et al. (2018) and GRU Karampatsis et al. (2020). Transformer-based neural language models have also been developed to learn representations of source code and automate software engineering tasks Alon et al. (2019); Hellendoorn et al. (2020); Chirkova & Troshin (2020); Liu et al. (2020). However, there are limited work on leveraging language models to improve the quality of code for better readability, understandability and maintainability. Studies have

also demonstrated that developers prefer to read Allamanis et al. (2014) and write Hellendoorn et al. (2015) codes that take common and conventional forms.

A related topic with our work is detecting code smells. While the definition of code smell is very broad, some of the issues that we detect such as long and complex method are often regarded as code smells in the software engineering community. Machine learning techniques have also been utilized to detect code smells. For instance, reference Kreimer (2005) uses decision trees to detect large class and long method. Reference Khomh et al. (2009) uses a Bayesian approach to identify blob, functional decomposition and spaghetti code. Similarly, reference Maiga et al. (2012) proposes support vector machines for detecting anti-patterns related to these issues. Reference Yang et al. (2012) leverages machine learning analysis for detecting code clones. Reference Fontana et al. (2013) uses various machine learning algorithms such as SVM and Random Forest to identify issues such as large class, data class, feature envy and long method. Our work differs from these previous studies in several aspects: 1) Previous works generally require labeled datasets that contain positive and negative code samples for training the machine learning model, and in many cases the labeling has to be done manually. Instead, our work is an un-supervised approach. We only need the source code corpus to train the language model and do inference. 2) Previous works to detect code smells use traditional machine learning algorithms such as decision trees and SVMs, while here we use the latest deep learning architecture for code quality issue detection. 3) Some of the issues we identified such as token-level errors, unnatural code and repetitive code are different from what have been addressed before. 4) We have used a single framework to detect diverse types of code quality issues, which are typically addressed using specifically designed datasets and algorithms in previous studies.

Apart from detecting code-related issues, previous study Tu et al. (2014) has also exploited localness of software by building cache language models. The cache model is a separate component which captures locally occurring n-grams. The combined models is a linear interpolation of global and cache models. The cross-entropy values of combined model is shown to be smaller than that of global model. In case of neural language models, we perform fine-tuning instead of separate model and interpolation. While previous works have only focused on the intrinsic performance of local models, here we investigate how specifically the local model can help detect code quality issues with better accuracy.

In addition, despite the tremendous progress in machine learning and deep learning, leveraging these advanced techniques in real code related applications is still challenging. A recent study (Raychev (2021)) investigates the gap between current machine learning models and classic static analysis tools, and has found that when applying machine learning approaches for code related tasks such as bug findings and code quality issue identifications, many academically interesting works have limited practical applicability in the wild. Here, our works takes a step forward to close such gap by showing that leveraging neural language model based approach can actually identify code quality issues in real code with decent accuracy.

## 7 Conclusions

We have proposed a single framework to identify several categories of code quality issues that can cause low code readability and maintainability based on a neural language model. We have developed an automated tool that spots large numbers of such issues in open-source repositories with low false positive rate. We have also presented the way to adapting the global models on specific repositories and studied the resulting local models. Our work paves the way for leveraging larger and more complex deep learning models for source code understanding, as well as more powerful tools for identifying code quality issues in software systems.

## References

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

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations*

*of Software Engineering*, FSE 2014, pp. 281–293, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635883.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018. ISSN 0360-0300. doi: 10.1145/3212695.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Pap

Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 252–261, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597102.

Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do programmers prefer predictable expressions in code? *Cognitive Science*, 44(12):e12921, 2020. doi: https://doi.org/10.1111/cogs.12921.

Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code, 2020.

Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *CoRR*, abs/1608.02715, 2016.

F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pp. 396–399, 2013. doi: 10.1109/ICSM.2013.56.

C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. Cacheca: A cache language model based code suggestion tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 705–708, 2015. doi: 10.1109/ICSE.2015.228.

V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 157–167, 2015. doi: 10.1109/MSR.2015.22.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, 2012.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pp. 1073–1085, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3377811.3380342.

F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pp. 305–314, 2009. doi: 10.1109/QSIC.2009.47.

Jochen Kreimer. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136, 2005. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2005.02.059. URL https://www.sciencedirect.com/science/article/pii/S1571066105051844. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 4159–4165. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/578.

F. Liu, G. Li, Y. Zhao, and Z. Jin. Multi-task learning based pre-trained language model for code completion. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 473–485, 2020.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur. Support vector machines for anti-pattern detection. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 278–281, 2012. doi: 10.1145/2351676.2351723.

Md. Abdullah Al Mamun, A. Martini, Miroslaw Staron, C. Berger, and J. Hansson. Evolution of technical debt: An exploratory study. In *IWSM-Mensura*, 2019.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276517. URL https://doi.org/10.1145/3276517.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *ICSE*, 2016.

V. Raychev. Learning to find bugs and code quality problems - what worked and what not? In *2021 International Conference on Code Quality (ICCQ)*, pp. 1–5, 2021. doi: 10.1109/ICCQ51190.2021.9392977.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 419–428, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594321.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *ICSE*, 2014.

Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pp. 708–719, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970341. URL https://doi.org/10.1145/2970276.2970341.

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 334–345. IEEE Press, 2015. ISBN 9780769555942.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics.

J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Filtering clones for individual user based on machine learning analysis. In *2012 6th International Workshop on Software Clones (IWSC)*, pp. 76–77, 2012. doi: 10.1109/IWSC.2012.6227872.

# Appendix

## A Language Model Structure

Thanks to the rapid progress in deep learning and natural language processing, there have been a variety of Transformer based language models. In this case, we adopted the concept of transfer learning and re-used a powerful language model pre-trained on large scale text data: the GPT-2 model firstly released by OpenAI. We re-used the structure and weights from the pre-trained GPT-2 model as initialization and continuously trained the model on massive Java source code. The primary reason that we chose to use a GPT-2 model is its autoregression nature. On the other hand, instead of training from scratch, we believe the pre-training on text data is a necessary step not only for better computational efficiency, but also for better model performance on source code, considering code is somewhat similar to natural language in nature as we discussed earlier. For example, many messages and logs in source code are actually texts, and other elements in code such as variable and method names may also contain semantics similar to natural language. For more details of the GPT-2 model we refer to the paper from OpenAI Radford et al. (2019). There are multiple versions of GPT-2 model depending on the token embedding size and we have used the model with the embedding size of 768 (i.e. small version). The total number of parameters of the model is 117M. A language model head is added on top to obtain token probabilities in the vocabulary.

Note that OpenAI has released a successor of GPT-2 model, namely, the GPT-3 model Brown et al. (2020). While the GPT-3 model is an enhanced model with more robust performance, it has clear limitation for practical use due to its enormous model size. It has 175B parameters ($\approx$1500 times of the GPT-2 model we use) and requires significant compute resources. Here, we want to experiment with smaller models and we would like to perform a comparative study as future work.

### A.1 Sub-word tokenization

As we discussed earlier, source code does not have an explicit vocabulary. To address the Out-Of-Vocabulary (OOV) issue, we adopted a widely used sub-word tokenization technique in NLP: the Byte-Pair Encoding (BPE) Sennrich et al. (2016). It has been demonstrated Karampatsis et al. (2020) that BPE is very effective in language modeling of source code. No word is OOV in BPE, and it also helps shrink the vocabulary and improves embedding. In this case, we use the BPE with the same vocabulary of the GPT-2 model with 50257 sub-words.

### A.2 Dataset

We train the GPT-2 model on open-source GitHub repositories (Java) that we collected. They are public GitHub repositories which are not forked, have at least 10 stars and have licenses of Apache or MIT. The statistics of our training dataset is shown in Table 3.

The test set consists of 5k randomly sampled GitHub packages satisfying the same license and star rating conditions as the training set.

Table 3: Statistics of Training Dataset

| | |
|---|---|
| #Java Projects | 19.3K |
| #Java Files | 2.7M |
| #Lines of Code (LOC) | 388M |
| #Sub-tokens | 2.5B |

### A.3 MODEL TRAINING

For preprocessing, we removed all the comments, empty lines and extra leading and trailing spaces in the Java codes. We treated each Java method as a training sample. For each Java method, the model iterates through each token, and for each token the model takes all previous tokens in the method and aims to predict the current token correctly. The training loss of this sample (i.e. this method) is the accumulated loss from all of its tokens. We used cross-entropy loss here as a standard approach.

Formerly, given a sequence of tokens $t_1, t_2...t_N$, the trained LM learns a probability distribution $P(t_1, t_2...t_N) = \prod_{m=0}^{N} P(t_m|t_{m-1}...t_0)$. The *cross entropy* of the sequence is the log probability per token, defined as $C(t_1, t_2...t_N) = -\frac{1}{N}\sum_{m=0}^{N} log_2 P(t_m|t_{m-1}...t_0)$. Note that cross entropy is independent of sequence length, which makes it a common metric to compute and compare for the performances of language models. The *perplexity* is the exponential of cross entropy $Perplexity = 2^C$. Lower cross entropy or perplexity indicates the model is more confident in prediction, hence better model performance.

The maximum input sequence length of the model is 1024 by default. We performed padding and truncation to the input data for efficient batch training as a common practice. The model is implemented in PyTorch under the Huggingface framework Wolf et al. (2020). We perform multi-GPU training on 4 V100 GPUs. It is trained for 3 epoch with 167k steps for each epoch, each step containing a batch of 64 training samples. We did not observe significant model quality improvements for training for more steps.

## B QUALITY OF THE LANGUAGE MODELS

To evaluate the intrinsic quality of the language models we build, we randomly collect test packages that are separate from the training set and evaluate our full and skeletal model by calculating the cross-entropy (in bits) for each Java method. We choose to evaluate the cross-entropy on method-level because methods are the most basic individual units of Java projects. We select 150k Java methods from the test packages randomly and evaluate their cross entropy by our models. We plot the distributions for both models in Figure 1. We also show the the statistical summary for both models in Table 4.

It is clear that the cross-entropy of the skeletal model is significantly lower than the full model, demonstrating that identifiers and literals contribute to most of uncertainty and unpredictability of code. The 1st and 3rd quantile values for the full model are 1.64 and 2.74, and the corresponding numbers for the skeletal models are 0.54 and 0.81, respectively. In addition, 95% of all methods have cross-entropy lower than 3.75 for the full model and 1.51 for the skeletal model. Having an understanding of the cross-entropy distribution helps us determine the parameters for code quality issue identification, which we will introduce in later sections.

The average cross-entropy for the full model is 2.23 (bits). We found several papers on language models of Java code in similar experimental settings that report cross-entropy, and we compare our performance with them as shown in Table 5. This work reports the lowest average cross-entropy we have seen compared to previous studies. However, although we believe the average cross-entropy is a relatively robust metric when evaluated on a random, separate and large test dataset, as the test datasets in previous studies are not the same as our experiments and are not publicly available, this may not be a direct apple-to-apple comparison. Despite this, the low average cross-entropy demonstrates decent intrinsic performance of our model.

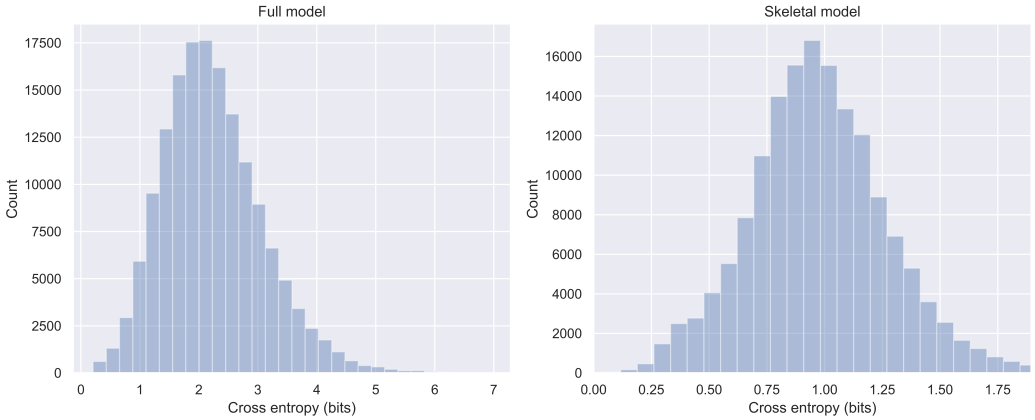


Figure 1: Cross-entropy Distribution of 150k Java methods randomly selected from test dataset for both the full and skeletal models.

Table 4: Cross-entropy Distribution at Method-Level

| | FULL | SKELETAL |
|---|---|---|
| MIN | 0.21 | 0.03 |
| 25% | 1.64 | 0.54 |
| MEDIAN | 2.15 | 0.67 |
| MEAN | 2.23 | 0.68 |
| 75% | 2.74 | 0.81 |
| 95% | 3.75 | 1.51 |
| MAX | 6.95 | 5.02 |

Table 5: Model Performance Comparison

| Reference | #Projects trained | Model | Cross-entropy |
|---|---|---|---|
| Allamanis & Sutton (2013) | 10.9k | n-gram | 4.86 |
| White et al. (2015) | 16.2k | LSTM | 3.35 |
| Karampatsis et al. (2020) | 13.4k | GRU | 2.40 |
| This work | 19.3k | GPT-2 | 2.23 |

## C EXAMPLES OF IDENTIFIED CODE QUALITY ISSUES

In this section, we show 1-2 detection examples of each type of code quality issue in real codes.

### C.1 TOKEN-LEVEL ERRORS

Consider another example shown below.

```
public void save() {
  ...
  preferences.setTasks(getPanel().getTaks());
}
```

In this example, the probability of sub-token `Taks` after `get` in the method name `getTaks` from our model is only $3.72e^{-6}$, while the model predicts an alternative sub-token of `Tasks` with a probability of 0.998, because it has seen `setTasks` before it. The model thus suggests a fix of `getTasks`.

In cases the surprising sub-token identified by the model is not an error, it may also suggest better identifier names. In the code snippet below, the model finds it surprising to have the sub-token `ll` after `ur` (probability = $6.9e^{-5}$), instead it suggests the sub-token `ls` with high probability (0.998). Clearly, `urls` is a better and more common name in this case than the original name `urll`, which is much less meaningful.

```
List<URL> urll = new ArrayList<URL>();
```

### C.2 UNNATURAL CODE

#### COMPLEX LOGIC

A real example identified by the model is shown below.

```
public static boolean bitOp(StateOp stateOp, long[] state, int pos, boolean val) {
   int lpos = pos >> 6;
   int bitoff = (pos & 63);
   return ((longOp(stateOp, state, lpos, (val ? 1l : 0l)<<bitoff, bitoff, 1) >>> bitoff) & 1l)==1l;
}
```

The return statements in this method contains a mix of multiple bit operations, comparison, ternary operator and method calls, which makes the code logic difficult to read and understand, causing a high cross-entropy of 2 for the whole method using the skeletal model.

#### FORMATTING ISSUES

Consider a simple example shown below. The model finds it surprising to have the opening bracket '{' in a new line, as well as the extra space after the '-' sign. Although such formatting issues are minor code smells and may depend on developer preference, we believe it is helpful to flag such issues and suggest developers follow comment coding conventions and practices for better readability.

```
public void moveBack()
{
   move(- getSpeed());
}
```

#### SWAPPED OPERANDS

A common example of checking for nullness is shown below, in which case the developer uses `null != connections` while most developers would use `connections != null` in this situation. The model gives this method a high cross-entropy of 1.89.

```
public void run() {
   if(null != connections)
     close();
}
```

### C.3 REPETITIVE CODE

In the following code example from Github, the developer populates the `bytes` list by repeatedly calling the `getHexValue` method one by one with hard-coded list indices. The model learns such highly repetitive pattern in code and assigns a low cross-entropy of 0.22, suggesting developer refactor the code. In this case, wrapping the code into a for-loop could reduce such repetitiveness for better readability and maintainability and make the code less prone to errors.

```
public static byte[] uuidToBytes( String string )
{
    char[] chars = string.toCharArray();
    byte[] bytes = new byte[16];
    bytes[0] = getHexValue( chars[0], chars[1] );
    bytes[1] = getHexValue( chars[2], chars[3] );
    bytes[2] = getHexValue( chars[4], chars[5] );
    bytes[3] = getHexValue( chars[6], chars[7] );
    bytes[4] = getHexValue( chars[9], chars[10] );
    bytes[5] = getHexValue( chars[11], chars[12] );
    bytes[6] = getHexValue( chars[14], chars[15] );
    bytes[7] = getHexValue( chars[16], chars[17] );
    bytes[8] = getHexValue( chars[19], chars[20] );
```

```
    bytes[9] = getHexValue( chars[21], chars[22] );
    bytes[10] = getHexValue( chars[24], chars[25] );
    bytes[11] = getHexValue( chars[26], chars[27] );
    bytes[12] = getHexValue( chars[28], chars[29] );
    bytes[13] = getHexValue( chars[30], chars[31] );
    bytes[14] = getHexValue( chars[32], chars[33] );
    bytes[15] = getHexValue( chars[34], chars[35] );

    return bytes;
}
```

Consider the following code snippet from a method identified by our tool. The repeated code structure contains 3 statements and it has repeated for more than 10 times in the method (only shown 3 here). Our model assigns a low cross-entropy of 0.2 for this method and suggests refactoring. In this case extracting the repeated structure into a separate method would resolve this code quality issue. This demonstrates that although not specifically designed for detecting code clones, our tool is capable of identifying such issues in a way that does not require sophisticated program analysis.

```
...
newAttribElement = Document.createElement("x1");
nodeElement.appendChild(newAttribElement);
newAttribElement.appendChild
(resultDocument.createTextNode(segX1));

newAttribElement = Document.createElement("x2");
nodeElement.appendChild(newAttribElement);
newAttribElement.appendChild
(resultDocument.createTextNode(segX2));

newAttribElement = Document.createElement("y1");
nodeElement.appendChild(newAttribElement);
newAttribElement.appendChild
(resultDocument.createTextNode(segY1));
...
```

## D Intrinsic Evaluation of the Local Model

To evaluate the performance of local models measured by cross-entropy, we randomly select 5 large Java repositories from our test corpus that are untouched by the global model training and trigger detections by both the global and local models. For each repository, we split all the methods into the training set and testing set with a 7:3 ratio. We continuously train the global model on the training set for 5 epochs and evaluate the model performance measured by average cross-entropy on the testing set. We have done this experiment for both the full and skeletal models, and the comparison of the model performance is shown in Table 6.

Table 6: Global Model vs Local Model

| Repository | Full | | Skeletal | |
|---|---|---|---|---|
| | Global | Local | Global | Local |
| ucarGroup | 2.39 | 1.08 | 1.02 | 0.56 |
| webmetrics | 2.42 | 1.51 | 1.24 | 0.85 |
| netbeans | 2.35 | 1.53 | 1.03 | 0.89 |
| NawaMan | 2.94 | 0.82 | 1.25 | 0.47 |
| cping | 2.43 | 1.31 | 1.07 | 0.77 |

As results show, for both full and skeletal the average cross-entropy has been reduced remarkably for the local model, indicating much better model performance than the global model. We also notice that the full model records more significant decrease of cross-entropy than the skeletal model, primarily due to the fact that each repository has its own conventions of identifier naming, which are captured by the local model after fine-tuning on part of the codes from each specific repository but overlooked by the global model.