Neural Language Models for Code Quality Identification

Srinivasan Sengamedu*
Amazon
USA

Hangqi Zhao†
Amazon
USA

ABSTRACT

Neural Language Models for code have lead to interesting applications such as code completion and bug fix generation. Another type of code related application is the identification of code quality issues such as repetitive code and unnatural code. Neural language models contain implicit knowledge about such aspects. We propose a framework to detect code quality issues using neural language models. To handle repository-specific conventions, we use local or repository-specific models. The models are successful in detecting real-world code quality issues with low false positive rate.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**.

KEYWORDS

Neural Language Model, Code Quality

ACM Reference Format:
Srinivasan Sengamedu and Hangqi Zhao. 2022. Neural Language Models for Code Quality Identification. In *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3549034.3561175

1 INTRODUCTION

It is now well-known that source code exhibits strong statistical regularities known as "naturalness" [10], 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. Language 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 [11], i.e., the cross entropy of buggy code is higher.

*Email: sengamed@amazon.com
†Currently with Twitter. Email: hzhao@twitter.com

MaLTeSQuE '22, November 18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9456-7/22/11…$15.00
https://doi.org/10.1145/3549034.3561175

While these recent progress on language modeling and deep learning have enhanced our understanding of source code, 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. Unlike bugs which are related to *code correctness*, code quality is about *code maintenance* such as code readability, consistent adherence to coding conventions, etc. Low code quality increases tech debt and harms the health of software development cycle in the long term [19]. Traditional approaches to code quality relied on notions such as cyclomatic complexity. If neural language models capture properties of good code, how do we provide actionable recommendations on bad code? For example, developers cannot easily act on recommendations such as "This segment of code has high cross-entropy", because the recommendation 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. *The challenge is to make the implicit knowledge in language models more explicit and actionable.* From a practical perspective, many automated tools such as static analyzers have been 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.

At a high level, we build two types of language models that work on different granularities: global model built at corpus level to capture good coding practices and local model finetuned at repository level to capture project-specific conventions. 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. We define rules on top of these models to provide actionable recommendations. Each type of model plays a role in providing different code quality recommendations. User evaluation shows that this approach has low false positive rate.

2 RELATED WORK

Code quality is often enforced using *rule-based* static analysis tools such as FindBugs, PMD, and CheckStyle.[1] Our focus is data-driven code quality analysis.

There are several ground work on studying source code with language models. Hindle [10] first 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. [3] on GitHub code in a cross-project setting. Such language models have enabled new tools for software engineering tasks. [11] has found buggy code has lower probability than correct code on average,

[1]See https://blog.sonarsource.com/what-makes-checkstyle-pmd-findbugs-and-macker-complementary/ for a comparison between the tools.

suggesting LM can be used for detecting code related issues. N-gram LMs has also been used for syntax error checking [5], code completion [13, 23] and bug detection by identifying low probability sequences [25].

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 [2] for a survey. Advanced machine learning models and architectures have been leveraged in language models for source code, including RNN [26], LSTM [8, 20], Pointer Network [17] and GRU [16]. Transformer-based neural language models have also been developed to learn representations of source code and automate software engineering tasks [4, 7, 15, 18]. 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 [1] and write [14] code that take common and conventional forms.

The application of generative models for code generation and bug fix generation are two recent trends. Codex [6] is a state-of-the art code generation model powering GitHub CoPilot. Language models have also been applied for generating bug fixes. See e.g., [9].

3 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 first 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 [21]. 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.

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 1(a).

The test set consists of 5k randomly sampled GitHub packages satisfying the same license and star rating conditions as the training set and not overlapping with the training set.

Model Training: For preprocessing, we removed all the comments, empty lines and extra leading and trailing spaces in the Java code. 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.

Formally, given a sequence of tokens $t_0, t_1, t_2 ... t_N$, the trained language model learns a probability distribution $P(t_0, t_1, t_2 ... t_N) = \prod_{m=1}^{N} P(t_m | t_{m-1} ... t_0)$. The *cross entropy* of the sequence is the log probability per token, defined as $C(t_0, t_1, t_2 ... t_N) = -\frac{1}{N} \sum_{m=1}^{N} \log_2 P(t_m | t_{m-1} ... t_0)$. 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 [12]. 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.

4 LANGUAGE MODEL VARIATIONS

We use the following variations of the GPT-2 based language model.

Full vs Skeletal Model: 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 skeletal model replaces all the identifiers and literals 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.

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 show the the statistical summary of cross-entropy distribution at method-level for both models in Table 1(b).

The average cross-entropy for the full model is 2.23 (bits). We found several papers on language models of Java code in similar

Table 1: Details of the model. (a) Training Dataset (b) Cross-entropy at method level (c) Performance Comparison

(a)

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

(b)

	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

(c)

Reference	#Projects trained	Model	Cross Entropy
Allamanis & Sutton [3]	10.9k	n-gram	4.86
White et al. [26]	16.2k	LSTM	3.35
Karampatsis et al. [16]	13.4k	GRU	2.40
This work	19.3k	GPT-2	2.23

experimental settings that report cross-entropy, and we compare our performance with them as shown in Table 1(c).

The average cross-entropy of the model is comparable 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.

Project-Specific Models: Previous study [24] 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. Here, to adapt our global model to a new repository, we take the strategy of finetuning the global model on the code from the new repository so the parameters and weights of the model is further optimized.

The local models are expected to have better performance in terms of cross-entropy as they are the fine-tuned versions [16, 24]. To evaluate the performance of our local models, 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 2.

We show, in Section 5.5 that the local model provides 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 code in the following subsections.

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 code from each specific repository but overlooked by the global model.

5 IDENTIFICATION OF CODE QUALITY ISSUES

Our hypothesis is that language models contain rich implicit information about code quality and leveraging implicit metrics (not the usual token or sub-token output) will lead to actionable outputs. More specifically, for an input code to be analyzed, we calculate the *token probability* for each sub-token and *cross entropy* and *log probability* at method level. We then show that we are able to detect four types of code quality issues based on these quantities. Finally, we show that project-specific models can be used to reduce false positives.

In Table 3, 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. In the following sections, we will discuss each type of issue in more details.

5.1 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 and $p(t')$ is higher than a higher-end threshold p_0', 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.

Table 2: 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

Table 3: Identification of code quality issues

Issue	Model	Metric	Logic	Values
Token error ($t \rightarrow t'$)	Full	Token probability	$p(t) < p_0$ and $p(t') > p'_0$	$p_0 = 1e^{-5}, p'_0 = 0.99$
Unnatural code	Skeletal/Full	Method cross-entropy	> threshold	2
Repetitive code	Skeletal	Method cross-entropy	< threshold	0.3
Long and complex code	Skeletal/Full	Method log probability	> threshold	500
False positives of global model	Project-specific	method/token log probability	< threshold	0.99

We use the full model to identify such issues as we need the specific identifier names and literals.

The following is an example we identified from Github repositories. For simplicity we have skipped irrelevant code.

```
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. The recommendation is to use `valignEditor` and `alignEditor` instead of the highlighted variable names. Note that such recommendations cannot be provided by using dictionaries.

5.2 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.

With this approach, we are able to identify diverse types of unnatural code. A few examples are listed below.

Type	Example
Complex Logic	mix of bit and logical operations
Formatting	extra spaces, unnatural line breaks
Unnatural Ordering	`0 <= a`

5.3 Repetitive Code

Both our previous discussion and previous studies have suggested that it is generally desired to have code 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 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.

5.4 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.

5.5 False Positive Suppression

We show how project-specific model understands project-specific conventions and can be used to suppress incorrect recommendations.

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 three 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: Often packages follow local conventions. 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 HUMAN EVALUATION

Threshold Selection: We run our tool on 5k randomly selected GitHub projects that are separate from our training dataset. See Section 3 for details of the training dataset. 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 this experiment, we choose the thresholds to seek for a good balance between the accuracy and coverage of the tool by evaluating 500 recommendations. See Table 3 for threshold settings.

Evaluation: We randomly select a subset of 500 remaining 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 4. 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.

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 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. Due to lack of ground truth, namely, the number of real code quality issues in the dataset, the recall metric is not applicable here.

Table 4: 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%

7 DETECTED CODE QUALITY ISSUES

In this section, we show a few examples of detection in real code, of each type of code quality issue mentioned in Section 5.

7.1 Token-Level Errors

Consider the 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>();
```

7.2 Unnatural Code

7.2.1 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.

7.2.2 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());
}
```

7.2.3 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();
}
```

7.3 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));
...
```

8 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 an unsupervised neural language model for code. 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.

Leveraging unsupervised language models for bug finding is challenging. A recent study [22] investigates the gap between current machine learning models and classic static analysis tools, and has found that many academically interesting research papers have limited practical applicability in the wild. Here, our work takes a step forward to close such gap by showing that making knowledge in neural language model explicit can actually identify code quality issues in real code with low false positives.

REFERENCES

[1] M Allamanis, E T. Barr, C Bird, and C Sutton. 2014. Learning Natural Coding Conventions. In *FSE*.
[2] M Allamanis, E T Barr, P Devanbu, and C Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* (2018).
[3] M. Allamanis and C. Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *MSR*.
[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.
[5] Joshua Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *MSR*.
[6] Mark Chen and et al. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374.
[7] Nadezhda Chirkova and Sergey Troshin. 2020. Empirical Study of Transformers for Source Code. arXiv:2010.07987
[8] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *CoRR* abs/1608.02715 (2016).
[9] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating Bug-Fixes Using Pretrained Transformers. *CoRR* abs/2104.07896 (2021).
[10] Abram Hindle et al. 2012. On the naturalness of software. In *ICSE*.
[11] Baishakhi Ray et al. 2016. On the naturalness of buggy code. In *ICSE*.
[12] Thomas Wolf et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP*.
[13] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. 2015. CACHECA: A Cache Language Model Based Code Suggestion Tool. In *ICSE*.
[14] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. 2015. Will They Like This? Evaluating Code Contributions with Language Models. In *MSR*.
[15] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code.
[16] Karampatsis and et al. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *ICSE*.
[17] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *IJCAI*.
[18] F. Liu, G. Li, Y. Zhao, and Z. Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *ASE*.
[19] Md. Abdullah Al Mamun, A. Martini, Miroslaw Staron, C. Berger, and J. Hansson. 2019. Evolution of Technical Debt: An Exploratory Study. In *IWSM-Mensura*.
[20] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*.
[21] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[22] V. Raychev. 2021. Learning to Find Bugs and Code Quality Problems - What Worked and What not?. In *ICCQ*.
[23] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *PLDI*.
[24] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *ICSE*.
[25] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug Detection with n-Gram Language Models. In *ASE*.
[26] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *MSR*.