How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines?

Johannes Villmow*+, Viola Campos*+, Jean Petry*, Amine Abbad-Andaloussi†, Adrian Ulges* and Barbara Weber†
* RheinMain University of Applied Sciences
Wiesbaden, Germany
Emails: johannes.villmow@hs-rm.de, viola.campos@hs-rm.de, jean.marcel.petry@gmail.com, adrian.ulges@hs-rm.de
†University of St. Gallen
St. Gallen, Switzerland
Emails: amine.abbad-andaloussi@unisg.ch, barbara.weber@unisg.ch

***Abstract*—Using meaningful identifiers in source code reduces the risk of errors, the cognitive load of developers, and speeds up the development process. Therefore, recent research has looked into an AI-based analysis of identifiers, for which large-scale language models appear to offer great potential. Based on tokens' probabilities, such models can suggest identifiers that are likely to appear in a given context. While current research has used language models to predict the most likely identifier names, studies on assessing the quality of given identifiers are scarce. To this end, we explore adherence to identifier naming guidelines as a proxy for identifier quality and propose and evaluate two unsupervised approaches for spotting violations: First, a generative approach, which uses the probability distribution of the language model directly without fine-tuning. Second, a discriminative method, which fine-tunes the model's encoder to discriminate between original identifiers and similar drop-in replacements suggested by a weak AI.**

We demonstrate that the proposed approaches can successfully detect violations of common guidelines for identifier naming. To do so, we have developed a dataset built on widely accepted identifier naming guidelines. The manually annotated dataset contains more than 6000 dense annotations of identifiers for 28 common guidelines. Using the data, we show that the generative approach achieves the best results, but that the particular masking strategy and scoring method matter substantially. Also, we demonstrate our approach to outperform other recent code transformers. In a per-guideline analysis, we highlight the potential and limitations of language models, and provide a blue-print for training and evaluating their ability to identify bad identifier names in source code. We make our dataset and models' implementation publicly available to encourage future research on AI-based identifier quality assessment.

***Index Terms*—Masked Language Models, Source Code Identifiers, Naming Guidelines, Code Quality, Identifier Quality Assessment, Generative Approaches, Language Model Fine-tuning, Code Transformers, AI-based Code Analysis.**

I. Introduction

The comprehension of source code is essential for conducting software development activities where developers are required to understand, change, or maintain existing code. This fundamental activity can be compromised when working with source code that has low readability [1]. In the literature, several factors have been shown to affect this quality attribute, notably those associated with the lexicon used to name identifiers [1]–[6]. In this vein, studies have demonstrated that good (i.e., clear, meaningful and concise) identifiers provide semantic cues that foster the comprehension of source code [6], while bad (i.e., non-intention reveling, ambiguous, vague) identifiers affect developers' cognitive load (i.e., mental effort) negatively and impede the understanding of source code [1]. Accordingly, a number of approaches and tools have been proposed to assess and improve the quality of identifiers [7], [8], most recently using transformer-based language models (LMs) [9]–[11]. These AI approaches are trained on large datasets of open source projects, have been successfully applied for smart autocompletion and have recently matured into widely used products [12], [13]. Given a piece of source code, an LM yields a probability distribution that expresses the likelihood of words (or *tokens*) in the code. Recent work in the area has focused on *identifier renaming*, i.e., suggesting new, better versions for identifiers [14]. This is conceptually rather straightforward: One samples identifiers from the language model's token distribution and picks the most likely one, which supposedly fits the given context best.

In practice, however, developers will unlikely use LM-based tools for automatic blanket renaming of *all* identifiers in their software projects. Rather, one could envision interactive tools to support software reviews, where the AI points developers to code passages with suboptimal identifiers that require refactoring. These can then be either auto-corrected or manually corrected, leading to interactive improvements of source code. To really focus on refactoring hotspots and to limit developers' distractions, the more interesting question is not to rename but to reliably *assess* identifiers, i.e., to decide if a given identifier should be refactored. Therefore, the scope of this paper is not on renaming but on estimating given identifiers' quality: We explore different strategies of using language models to predict if an identifier is "good", given its context.

This work has been supported by the Federal State of Hesse / Germany, Program LOEWE 5 (Project "Code Buddy"), The author Amine Abbad-Andaloussi is supported by the International Postdoctoral Fellowship (IPF) Grant (Number: 1031574) from the University of St. Gallen, Switzerland.

+Both authors contributed equally to this research.

The evaluation of identifier quality in source code presents a significant challenge due to its subjective and difficult-to-quantify nature. To address this issue we rely on *coding guidelines* as well-defined indicators of identifier quality. These naming guidelines encompass widely accepted criteria for well-chosen identifiers, such that deviations from these guidelines can serve as a clear indication of poor identifier quality [5], [15], [16]. Adhering to these guidelines is considered a basic prerequisite for producing high-quality code. Therefore, the question of whether language models can identify "bad" identifiers can be reframed as how effectively they can detect violations of established, empirically validated coding guidelines. Accordingly, we define identifier quality in our context based on compliance with these guidelines: An identifier is considered "good" if it follows established naming guidelines, and "bad" if it violates them. It is important to note that the difficulty of verifying guideline violations may vary. While a language model holds promise for guidelines covering semantics, other violations (e.g. addressing identifier syntax) can be verified using simple rule-based methods. Nevertheless, it remains intriguing to explore the extent to which language models for evaluating identifier quality align with guidelines from either of these categories. From the practical software engineer's perspective, a good LM for quality assessment should be in line with any recognized best practices in software development.

To this end, our first contribution is a novel approach towards identifier assessment, which we call CODEDOCTOR. Since we are not aware of any annotated large-scale datasets that would facilitate a supervised training, CODEDOCTOR's training is self-supervised in a sense that no manual annotations of identifier quality are required. Specifically, we investigate two different variations of the CODEDOCTOR model:

1) **Generative rating**: We first train a transformer-based language model on a large dataset of open-source code. The model learns to predict held-out identifier names (e.g. variable and method names) and code passages. To assess an identifier's quality in a new piece of code, we let the LM estimate the identifier's probability and then analyze either its likelihood directly, or its likelihood ratio compared with the model's most favored identifier (see Section III-B).
2) **Discriminative rating**: Our second approach fine-tunes the above language model with a discriminative training, in which we randomly replace identifiers with alternative versions suggested by a weak AI. This way, the model learns to discriminate real identifiers from fake ones. (see Section III-D).

Finally, to evaluate identifier quality, we address the fact that identifier quality is subjective. We turn towards *coding guidelines*, and violations thereof, as a well-defined criterion for spotting "bad" identifiers.

To the best of our knowledge, we present the first analysis focusing on the accuracy of language models in detecting guideline violations.We present a dataset that contains 6203 dense annotations of identifiers for 28 common guidelines, and evaluate our models on this newly developed dataset. Thereby, we also take a detailed look at our LMs' performance across different guidelines (e.g., guidelines addressing identifier syntax vs. guidelines covering semantics) and discuss what makes a guideline challenging to predict. We make both the dataset and our models' implementation public [17] to invite future research on AI-based identifier quality assessment.

Our work has implications on current research, practice, and education. From a research perspective, we demonstrate the potential and limitations of language models in identifying bad identifiers in source code. Additionally, we investigate different masking and scoring variations. Since these are shown to have a significant impact on scoring quality, our training and evaluation methods can serve as a blue-print for developing such models for similar use cases. In practice, our approach can be integrated into existing IDEs as a plugin, assisting developers in detecting potential areas of poor identifier quality during code review and thus improving the quality of source code. Finally, our approach can be deployed in educational settings to help students write clean and readable code during practical exercise sessions.

II. RELATED WORK

First, we discuss the quality of code identifiers and their impact on code comprehension. We then provide a brief overview of work aimed at assessing and improving the quality of identifiers, with a focus on current data-driven approaches. For a comprehensive survey, please refer to [10].

A. Identifier Naming and Source Code Comprehension

The quality of source code identifiers has been investigated in particular with respect to their relationship to code comprehension [1]–[7], [18]. In [19], the authors suggested that the meaning conveyed by words used in identifiers significantly influences the comprehension of source code [19]. Hence, effective identifiers naming can help finding key points of interest (i.e., semantic cues), and infer their role within the entire code [7]. This proposition is further supported in [6], demonstrating that the presence of semantic cues enhances code comprehension [6].

To explain the relationship between identifiers' names and code comprehension from a cognitive perspective, it is necessary to differentiate the essential and accidental complexity layers of source code [20], [21]. While the former originates from the complexity of the implemented software functionalities, the latter emerges from developers' poor implementation and bad writing style. Accidental complexity, which includes identifier quality, can impose additional cognitive load on developers [22] when bad identifiers are used, creating more strain on human memory during mentally demanding tasks and increasing the likelihood for committing errors [1], [23]–[26].

In the literature, several empirical studies have investigated the impact of identifier quality on code comprehension [1]–[6]. These studies have indicated that good naming of methods and variables aids comprehension [2], [3], [6], and that full identifier names are more comprehensible than abbreviated ones [3]. Additionally, poor identifier names that violate existing naming guidelines have been associated with certain defect types in source code [5], and high cognitive load [1]. These studies provide ample evidence for the importance of good identifiers' naming, which raises the need for adhering to the existing guidelines on that matter.

The literature comprises several collections of identifier naming guidelines (e.g., [5], [15], [16]) targeting the code syntax and vocabulary as well as the alignment of identifiers with the data types and the methods which they refer to [5], [15], [16]. Syntax guidelines focus on the construction and formulation of identifiers from words [15]. Vocabulary guidelines emphasize the importance of choosing appropriate words, assuming that the choice of words has a direct impact on the understandability of the code [15]. Data types guidelines, in turn, emphasize the importance of choosing identifiers revealing the type of the data they refer to [15]. Lastly, method name guidelines address the relevance of aligning the name of methods with the underlying implementation [16]. Many of these guidelines were shown to impact developers' comprehension of the source code when tested separately (e.g. in [3], [4]) or in a combined manner (e.g., in [1], [5]).

Given the significance of the existing guidelines [5], [15], [16] and their empirically verified effect on developers' comprehension of the source code, they can serve as a benchmark for evaluating the quality of existing code and identifying suboptimal identifiers.

B. Automatic Evaluation and Improvement of Code Identifiers

The insight that meaningful variable names are crucial to code understanding has led to the development of various rule-based techniques to suggest meaningful and consistent identifier names. These approaches include learning coding style rules from existing source code [27], [28] or examining relations between variables to infer whether a variable name should be changed along with others [29], [30]. In [7] and [31] poorly named variables are identified using a dictionary-based approach or static code analysis.

a) Learning-based approaches: In recent years, numerous learning-based works have emerged, taking advantage of the fact that software exhibits characteristics similar to natural language, such as being highly repetitive and predictable [32], and as such can be effectively represented by statistical language models. This hypothesis of the *naturalness of software* motivated the development of large language models pre-trained on source code [12], [33]–[36] that have demonstrated remarkable performance on program synthesis and other code related tasks [13]. Furthermore, Ray et al. [37] show for a corpus of bug-fix commits that buggy code has, on average, a lower probability than correct code, suggesting that language modeling can be leveraged to detect code quality related issues.

With NATURALIZE, a framework for refactoring tasks, Allamanis et al. [38] presented the first work using statistical natural language modeling to learn identifier naming along with other coding conventions within a code-base. The authors further develop the approach by training the model only on tokens containing lexical information and by additionally considering the type information of a variable [8]. Chen et al. analyze and evaluate variable names by learning semantic similarity [9]. The study is the first to investigate a transformer-based language model in its model comparisons. None of the above work on variable naming, however, considers our task of estimating identifier quality with transformers.

A similar line of work [39]–[43] studies the evaluation and suggestion of method names using probabilistic language modeling [39], convolutional neural networks [40], abstract summarization [40] or contextualized representations [42], [43]. In contrast to the aforementioned work, we investigate identifiers in general, considering both variable and method naming.

Most similar to our approach are studies investigating the use of transformer-based LMs to assess or improve the quality of code. Sengamedu et al. [11] utilize transformer models to identify code quality related issues and provide recommendations for refactorings. However, they evaluated code quality either at sub-word or function level, whereas our focus is specifically on the quality of identifiers in code. Also, like in other studies on using language models for code-related activities (e.g. [44], [45]), benchmarking happens against existing code, which is implicitly assumed to be good. We challenge this assumption and present results on a dataset with manually annotated identifiers. Finally, a recent work [14] studied the potential of three data-driven approaches for automatic variable renaming, two of which are based on transformer models. The authors evaluate the approaches by mining refactored identifiers in projects and assume the refactored version to be good. While this approach is very likely to yield solid ground truth, it offers little insight into *why* identifiers were refactored and which properties make bad identifiers easy or difficult to spot. Therefore, we specifically address LM-based identifier assessment in combination with coding guidelines, including a detailed analysis of which guidelines are challenging to assess for language models.

III. APPROACH

Following common practice in neural text processing, we start with tokenizing input code snippets into a sequence of input tokens $x_1, ..., x_n$. Our goal is to analyze *identifiers* in this token sequence. These include names for variables, methods, modules and types. Note that tokenization may segment long identifiers into multiple – so-called "subword" – tokens (e.g., `data-Base-Connection`) [46]. Therefore, in general an identifier is not a single token but a subsequence of tokens. Note also that an identifier may occur multiple times in the token sequence, and that we keep track of these occurrences utilizing the code's syntax tree. We use the `tree-sitter` library[1] to build the trees and track its identifiers.

[1]https://tree-sitter.github.io/tree-sitter/

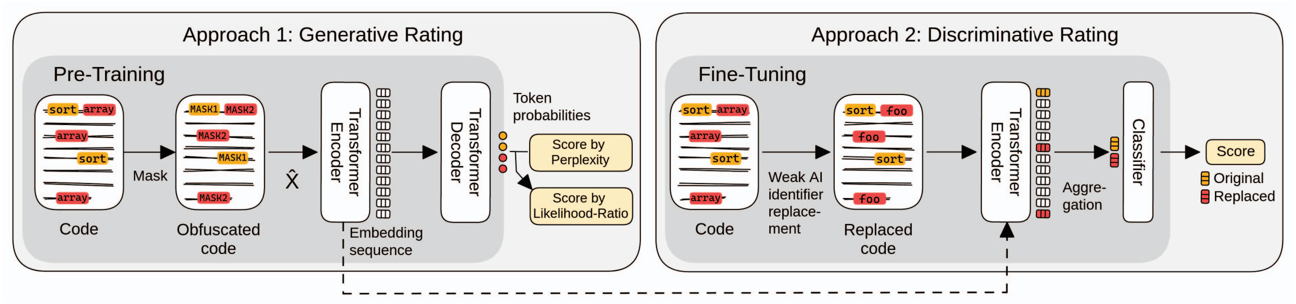


Fig. 1. Our approach uses an encoder-decoder transformer, which we pre-train using span prediction and identifier deobfuscation (not illustrated) . Given the pre-trained model, we investigate two ranking strategies: A generative rating (left), and a discriminative rating which fine-tunes the encoder using identifiers replaced by a weak AI (right).

The token sequence is then fed to a transformer model [47] with two components[2]: First, an encoder, which processes the input sequence $x_1, .., x_n$ and produces a sequence of token *embeddings*. Second, a decoder, which generates an output token sequence conditioned on the input tokens' embeddings.

We pre-train our transformer with a common *masked language modeling* task: The model is fed randomly selected code passages in 16 popular programming languages from a large-scale set of open-source projects[3]. In each input passage, we mask out random subsequences (referred to as *spans*), and train the model to estimate these subsequences. By solving this task, the model learns to estimate the likelihood of tokens to appear in a given context of code. Particularly, this includes the ability to estimate the likelihood of identifiers. Given the pre-trained model, we investigate two strategies for identifier quality assessment (both are sketched in Figure 1):

- Generative rating, which directly uses the likelihood produced by the pre-trained model.
- Discriminative rating, which uses only the model's encoder and fine-tunes it to discriminate real identifiers from fake ones inserted using a FASTTEXT [49] model.

Both strategies are self-supervised in a sense that no labeled training data is required. We will discuss the details behind each approach in Sections III-B and III-D, and discuss their implications from a probabilistic perspective in Section III-C.

A. Pre-Training

Pre-training operates on unlabeled code passages, each a series of tokens $X = x_1, \dots, x_n$. In each training step, we randomly replace multiple subsequences of X, $Y_1, \dots, Y_m$, with mask tokens $x_{MASK_1}, \dots, x_{MASK_m}$, obtaining an obfuscated version $\hat{X}$ of the code passage. We create a target sequence by concatenating (denoted by $\oplus$) the held-out subsequences, preceding each one with its mask symbol:

$$Y = [x_{MASK_1}] \oplus Y_1 \oplus \dots \oplus [x_{MASK_m}] \oplus Y_m \oplus [x_{EOS}]$$

The goal of pre-training is to reconstruct the subsequence Y, given $\hat{X}$. To do so, our model's decoder predicts the probabilities of the target sequence Y's tokens $y_1 \dots y_o$ given $\hat{X}$ in an autoregressive fashion, i.e., for each y_i we obtain $p(y_i|y_{<i}, \hat{X})$. We learn the model's parameters Θ by maximizing the (averaged) log likelihood

$$logL_\Theta(Y|\hat{X}) = \frac{1}{o}\sum_{i=1}^{o} log\ p(y_i|y_{<i}, \hat{X}) \tag{1}$$

We train the model on two tasks simultaneously by randomly sampling two different kinds of subsequences $Y_1, \dots, Y_m$,:

a) 1. Identifier deobfuscation: If $Y_1, \dots, Y_m$ refer to identifiers, the model is asked to recover hidden identifiers. This task is known as *identifier deobfuscation* [50], and is obviously closely related to assessing identifier quality. We sample identifiers to obfuscate using the following procedure: Let $\mathcal{I}$ be the set of identifiers that occur in X. We draw a percentage $p \sim \mathcal{N}(60, 25^2)$ and select a random subset $\mathcal{I}'$ of p percent of identifiers from $\mathcal{I}$. For the ith identifier in $\mathcal{I}'$, we replace *each* of its occurrences with 95% chance with an identifier-specific mask token x_{MASK_i}. Note that x_{MASK_i} can occur multiple times in $\hat{X}$ and each identifier can consist of multiple tokens.

b) 2. Span Prediction: $Y_1, \dots, Y_m$ can also refer to longer subspans of code, in which case our task is known as span prediction [34]. We add this training task to force our model to gain a broader understanding of the structure of code. For span prediction we either (1) mask a single large span with $|Y_1| \sim \mathcal{N}(150, 90^2)$ or (2) mask multiple short spans $|Y_i| \sim Poisson(8)$ covering around 15% of the original code. When masking short spans, in 90% of the samples we select syntactically complete spans aligned with the code's syntax tree [51] (e.g. the parameters of a function call). This hardens the task, since syntax alignment produces challenging spans more often than random span masking.

B. Generative Rating

During pre-training our model has already learned to deobfuscate identifier names. We denote with $logL(Y^i, \hat{X})$ the log-likelihood from Equation (1), averaged only over those tokens belonging to the subsequence Y_i representing identifier no. i.

[2]Specifically, we opt for the T5-base architecture with 12 layers in encoder/decoder and a total of 245M parameters [48].

[3]The complete list of programming languages can be found in [17].

We will refer to this quantity as an identifier's log-likelihood in the following, and suggest two ways of using it to estimate the identifier's quality:

1. Perplexity: Perplexity measures how well our model predicts the identifier and is commonly used to evaluate language models. Essentially, it reflects the degree of uncertainty the model has in its predictions, with higher perplexity indicating lower probabilities for the identifier's tokens. We use perplexity directly to assess an identifier I^i's quality:

$$Score(I^i) = e^{-logL(Y^i, \hat{X})} \tag{2}$$

The higher this score, the more "suspicious" the identifier.

2. Log-likelihood Ratio: A high perplexity, as introduced in the last section, may indicate a poorly chosen identifier, but could also occur due to the inherent complexity of the context. To distinguish both situations, we study a second scoring method that estimates whether the language model finds another identifier much more plausible: Let Y_i be an identifier we assess, and let $\hat{Y}_i$ be the supposedly best identifier that our language model predicts in place of Y_i (using beam search of width 5). Then we define our second score as the ratio of both identifiers' log-likelihoods:

$$Score(I^i) = \frac{logL(\hat{Y}^i, \hat{X})}{logL(Y^i, \hat{X})} \tag{3}$$

We refer to this score as the likelihood ratio in the following. The higher it is, the more "suspicious" the identifier.

Note that the log-likelihood – and with it both our scoring methods – is strongly influenced by two factors: First, the *overall amount of masking*: The more identifiers are obfuscated, the less information the model can draw from identifiers and the less confident its predictions. Second, the *order* in which identifiers occur in the target sequence: Since during generation the model's decoder attends to token to the left of the current token, the model can exploit identifiers $1, \ldots, i$ as context when predicting identifier $i{+}1$, which impacts its log-likelihood. To address these problems, we explore two options.

a) Mask-Single: The first option is to mask only a single identifier (i.e., $|\mathcal{I}'| = 1$). Other identifiers present in the context can now be used to draw conclusions about the masked identifier. It is noteworthy that this approach requires multiple forward passes - one for each identifier - when scoring all identifiers in X.

b) Mask-All: The other option we explore is to mask all identifiers simultaneously (i.e., $\mathcal{I}' = \mathcal{I}$). When masking all identifiers at once the model has limited information about other identifiers in the context. However, this option requires only a single forward pass through the model, thus is $|\mathcal{I}|$ times faster than *Mask-Single* when rating all identifiers.

C. Probabilistic Interpretation

From a probabilistic perspective, identifier assessment comes as a binary classification problem. Given an Identifier I in a code context X, we should estimate

$$P(C{=}1|I,X) = \frac{P(I,C{=}1,X)}{P(I,C{=}1,X) + P(I,C{=}0,X)}$$
$$= \frac{P(C{=}1,X)\cdot P(I|C{=}1,X)}{P(C{=}1,X)\cdot P(I|C{=}1,X) + P(C{=}0,X)\cdot P(I|C{=}0,X)}$$

where class $C{=}1$ denotes the class of "bad" identifiers and $C{=}0$ "good" ones. Last section's generative rating approach, however, does not rank by (descending) $P(C{=}1|I,X)$ but only by the (ascending) likelihood $P(I|X)$. From a probabilistic perspective, this corresponds to approximating the above score $P(C{=}1|I,X)$ with

$$P(C{=}1|I,X) \approx \frac{const}{const + P(C{=}0,X)\cdot P(I|C{=}0,X)}$$
$$\approx \frac{const}{const + P(X)\cdot P(I|X)}$$

This comes with three potentially problematic assumptions:

First, the language model implicitly assumes all code in pre-training to be "good", i.e., it approximates the class-specific model for Class 0 with a class-independent one: $P(C{=}0,X) \approx P(X)$ and $P(I|C{=}0,X) \approx P(I|X)$. To address this problem to some extent, we apply quality filters on the training data by focusing on high-quality open-source projects, hoping that these will come with mostly good identifiers. Also, we assess the impact of this risk by evaluating on a manually annotated dataset.

Second, since we approximate $P(C{=}0,X) \approx P(X)$, we have no notion of *context quality*: Unfortunately, the likelihood of a bad identifier may be much higher when all other identifiers in the context are bad in a similar fashion. Consider a piece of source code in which all identifiers are single-letter abbreviations (bad quality). The likelihood that a masked identifier in this snippet is also a single letter will be high: After all, the language model predicts the most likely identifier *given this context*. To address this problem, we study two prediction "modes" *Mask-Single* and *Mask-All* (see Section III-B). In the latter case, when masking *all* identifiers at once, we decouple the likelihood estimation from the context's identifiers' quality (at the cost of not being able to exploit other "good" identifiers in the context).

Third, the language model is trained to reproduce any code it is given, but not to discriminate "bad" samples. From a probabilistic perspective, it lacks a proper model for Class 1. In these cases, it is common to assume a uniform distribution for the unknown class, i.e., all identifiers are equally likely to appear in bad code ($P(C{=}1,I,X) = const$). In practice, however, bad identifiers can be expected to follow certain anti-patterns (e.g., developers using specific vague terms).

We address this problem by studying a discriminative approach in the next section, which explicitly draws positive training samples from a weak AI.

D. Discriminative Rating

As outlined in the last section, a potential challenge with the above generative rating is the lack of a counterclass model.

TABLE I
GUIDELINES USED TO ASSESS THE QUALITY OF IDENTIFIERS.

ID	Type	Description	Confirming Example	Violating Example
1	Syntax	Apply standard case with rigorous consistency [5], [15]	`databaseName`	`databaseNAME`
2	Syntax	Use dictionary words and no (uncommon) abbreviations [5], [15]	`databaseName`	`dbNm`
3	Syntax	Expand single-letter names (except for control variables) [5], [15]	`databaseName`	`d`
4	Syntax	Only use one underscore at a time [15]	`database_name`	`database__name`
5	Syntax	Only use underscores between words [5], [15]	`database_name`	`_databaseName`
6	Syntax	Name constant values [15]	`BOILINGPOINT`	`ONEHUNDRED`
7	Syntax	Limit name character length to 20 [15]	`databaseName`	`databaseIdentificationName`
8	Syntax	Limit name word count to four [5], [15]	`databaseName`	`newDatabaseNameOfStudent`
9	Syntax	Qualify values with suffixes [15]	`studentCount`	`countStudent`
10	Vocabulary	Describe meaning [15]	`databaseName`	`foo`
11	Vocabulary	Be precise [15]	`databaseName`	`name`
12	Vocabulary	Use standard language [5], [15]	`terminate`	`whack`
13	Vocabulary	Use a large vocabulary [15]	`learningPerson`	`student`
14	Vocabulary	Omit type information [15]	`databaseNameString`	`databaseName`
15	Data type	Use singular names for values [15]	`User user;`	`User users;`
16	Data type	Use plural names for collections [15]	`User[] users;`	`User[] user;`
17	Data type	Use Boolean variable names that imply true or false [15]	`isFinished`	`finish`
18	Data type	Use positive Boolean names [15]	`isFinished`	`notRunningAnymore`
19	Data type	Attribute name and type should be consistent [15], [16]	`int studentCount;`	`String studentCount;`
20	Method name	Use a verb-phrase name [15]	`createUser(): User`	`userCreation():User`
21	Method name	Don't use `get`, `is` or `has` prefixes for methods with side-effects [15], [16]	`getUser(): User`	`getUser(): User` (creates User)
22	Method name	Only use `get` prefix for field accessors that return a value [15], [16]	`getUser(): User`	`getUser(): void` (fetches API)
23	Method name	Only use `is` and `has` prefixes for Boolean field accessors [15], [16]	`isActive(): Boolean`	`isActive(): void`
24	Method name	Only use `set` prefix for field accessors that don't return a value [15], [16]	`setName(name): void`	`setName(name):String`
25	Method name	Only use transformation verbs for methods that return a transformed value [15], [16]	`toString(): String`	`toString(): void`
26	Method name	Expecting and getting single instance [15], [16]	`getUser(): User`	`getUser(): User[]`
27	Method name	Expecting and getting a collection [15], [16]	`getUsers(): User[]`	`getUsers(): User`
28	Method name	Method name and return type should not contradict [15], [16]	`getUser(): User`	`getUser(): Database`

LMs simply consider all written code presented during training to be good and have never been tasked with bad samples.

To address this problem, we investigate a discriminative rating. We use a weaker language model to sample worse but plausible drop-in replacements for true identifiers. We then replace the true identifiers (instead of masking them as in pre-training), and fine-tune the encoder of the pre-trained encoder-decoder model to tell the true identifiers from the fake ones (note that this approach has successfully been explored with transformers in NLP before [52]). For further details on the generation of the training data please refer to Section IV-B.

Model-wise, we aggregate the encoded representations of all sub-tokens and of all occurrences of an identifier into a single embedding. Formally, let $\mathbf{h} \in \mathbb{R}^{n \times d}$ denote the token embeddings produced by the transformer encoder, I the identifier for which we want to obtain an embedding $\mathbf{s} \in \mathbb{R}^d$, and $g(k, I) \mapsto \{0, 1\}$ a function that returns 1 iff. the token at index k belongs to identifier I and 0 otherwise. We then perform a max-pooling operation:

$$s_i(I) \;=\; max_{\{k|g(k,I)=1\}} \;\; h_{ki} \quad \text{for } i{=}1, \ldots, d. \tag{4}$$

This identifier embedding is fed into a classification head. We project the embedding using a linear layer $\mathbf{W_1} \in \mathbb{R}^{d \times d}$ with ReLU activation, followed by a linear output layer $\mathbf{W_2} \in \mathbb{R}^{d \times 1}$ with a sigmoid layer σ.

$$\mathbf{t} = \text{ReLU}(\mathbf{s} \cdot \mathbf{W_1}) \tag{5}$$

$$P(C{=}1|I, \hat{X}) = \sigma(\mathbf{t} \cdot \mathbf{W_2}) \tag{6}$$

We train by optimizing the binary cross entropy loss averaged over all identifiers in batch.

TABLE II
NUMBER OF CODE SAMPLES IN OUR DATASETS.

Dataset	Train	Validation	Test
pre-training dataset	$32M$	50.000	-
fine-tuning dataset	64.474	13.911	13.854
manually annotated dataset	-	-	1.770

Key to the discriminative approach is the quality of the weak AI's identifier replacements. We would like them to be worse than the real identifiers, but at the same time plausible examples of real-world "bad" identifiers. To assess this, we conducted a blind annotation of 100 real/fake identifier pairs, in which a human annotator was asked to compare the fake and real identifier by quality. In 86% of cases, the annotator found the FASTTEXT suggestions to be worse than the original. In 12% of cases, quality was found comparable, and in 2% of cases even better.

IV. DATASETS

To train and evaluate the models under investigation, we built an evaluation and two training datasets: the *manually annotated dataset*, which we use to evaluate our models' performance against coding guidelines; the large-scale *pre-training dataset*; and the *fine-tuning dataset*, which is used to fine-tune the discriminative model. Table II provides an overview of the size of the datasets.

A. Manually Annotated Dataset

We constructed our evaluation data by initially selecting 20 top active GitHub projects spanning various domains, measured by the number of watchers, forks and collaborators. From these repositories, we randomly selected 59 Java files for annotation, each containing at least one function consisting of a minimum of 10 lines to exclude trivial data classes. Additionally, each file had no more than 5 imports from the Java Class Library and 5 imports from external sources. These constraints were applied to guarantee that the files could be comprehended independently of their parent projects.

To assess the quality of the identifiers, we selected a set of 28 accepted coding guidelines from [15], [5] and [16] by first collecting all proposed guidelines, removing duplicates and near-duplicates, and discarding highly specific guidelines, for which our code samples did not contain enough suitable identifiers (e.g. for class names). As shown in Table I, eight of these guidelines are aimed at syntax, four at vocabulary, six targeting data type, and nine guidelines targeting method naming. We then annotated all original identifiers over all files for their conformance with respect to all guidelines. We found 677 of 2143 identifiers (32%) to violate any of the guidelines. In a second annotation step, for 865 randomly selected identifiers, either a conforming variant was proposed if the identifier already violated a guideline or otherwise, violating substitutions were proposed for specific guidelines. In total, we annotated 4503 guideline violations and 1700 confirming identifiers in our code samples. The annotation process was conducted by six developers. Each single annotation in Phase 1 was made by one person, whereas annotations in Phase 2 were cross-checked by a second person.

Given these annotations, we generated our evaluation dataset. For each of our 59 files, we first replaced all occurrences of "bad" identifiers with "good" ones (if available). For each code file, we generated 30 random variations of the file, each one by sampling up to 5 identifiers for which "bad" versions existed and replacing them. This resulted in 1770 densely labeled code passages.

B. Training Datasets

Our training data is unlabeled, and is based on the assumption that identifiers used in open-source code are generally well chosen by the original developers. We pre-trained our model on a dataset of $33M$ code files scraped from GitHub in 16 different programming languages. The code files were sourced from repositories that had more than 10 stars. The dataset was preprocessed by performing per file deduplication and 570 repositories were kept for validation. On this *pre-training dataset* we generated training samples (Section III-A).

For the discriminative training, we replaced identifiers with semantically similar but less meaningful substitutes, obtaining a large-scale training set where the model learns to distinguish between original and substituted identifier names. We generated this *fine-tuning dataset*, as outlined in Section III-D, by replacing up to 20% of the identifiers in a dataset of $100k$ java files with similar, less meaningful identifiers generated by FASTTEXT. To do so, we trained an unsupervised FASTTEXT word embedding on $1M$ identifiers in our pre-training data. For each real identifier I, we sampled from the top 5 most similar identifiers to I, where the probability of drawing an identifier is proportional to the cosine similarity between the FASTTEXT embeddings. For example, we could replace `test` with `test2`, `ALLOWABLE_ERROR` with `ERROR_CODE`, or `setModel` with `setRepeatMode`.

TABLE III
COMPARISON OF GENERATIVE SCORING METHODS.

	Perplexity		Likelihood-Ratio	
Model	Mask-Single	Mask-All	Mask-Single	Mask-All
INCODER	22.69	25.27	23.43	23.78
CODEDOCTOR	55.14	52.79	**62.00**	28.15

V. EXPERIMENTAL SETUP

The aim of our study is to evaluate transformer-based techniques for assessing the quality of code identifiers. To this end, we evaluated the two approaches described in Section III using our manually annotated evaluation data. We describe evaluation metrics and training protocol in the following. Furthermore, since we compare the results of our models against those of two current language models pre-trained on code, we describe our usage of these reference models in Section V-B.

A. Evaluation Procedure

To evaluate how well our models detect identifiers violating the guidelines, we set up a retrieval task, much like pinpointing and ranking potential hotspots for code review: We rank all identifiers in a file by their likelihood to violate any of the coding guidelines, as indicated by our model's score. For each guideline G and code file d, we measure how well our ranking reflects violations of guideline G. To do so, we use the average precision $AP(G, d)$, which is a common metric for retrieval tasks[4]. Thereby, we take into account that

1) code files may contain identifiers that violate *other* guidelines than G. We filter these from the result list when evaluating Guideline G.
2) of each source code file $d \in D$, multiple random variations have been sampled (see Section IV-A), in which the same identifier may occur. We average each identifier's precision over these occurrences.

Finally, we average precision over all input documents, obtaining the so-called mean average precision (MAP):

$$MAP(G, D) = \frac{\sum_{d \in D} AP(G, d)}{|D|} \tag{7}$$

where D denotes the collection of code files in which violations of Guideline G occur. For further details, please refer to the evaluation script in our replication package [17].

[4]Measuring AP fits practical settings in which the software revisor identifies ranked hotspots and stops reviewing once resources are depleted or the potential identifiers seem less promising. Unlike precision and recall, we evaluate the order of results and no threshold on the scores is required.

B. Implementation of Other Language Models

To benchmark the performance of our models against current language models pre-trained on source code, we compare our approaches with two other transformer models from current research, namely INCODER [35] as a pre-trained decoder model for the generative rating, and GRAPHCODEBERT [53] as a pre-trained encoder for the discriminative rating.

a) Discriminative rating using GRAPHCODEBERT*:* The GRAPHCODEBERT model proposed in [53] is a transformer encoder language model trained on code. GRAPHCODEBERT was pre-trained on masked language modeling, but without a special focus on identifier deobfuscation. Additionally, two structure-aware pre-training tasks were introduced, where the model learned to predict data flow between variables and to align representations between source code and code structure. We replace our encoder with the pre-trained GRAPHCODEBERT , but otherwise follow the same approach as described in Section III-D, in which we pool the identifiers' embeddings and use a classification layer for prediction, and conduct a discriminative learning during which we fine-tune into the encoder. Since GRAPHCODEBERT only supports input lengths of up to 512 tokens, we split longer code files into multiple chunks. Thereby, we use overlapping chunks to provide sufficient context for the model whenever possible, and average scores from identifiers appearing in multiple chunks before evaluation. For fine-tuning, we use an effective batch size of 84 and sweep over the learning rate.

b) Generative rating using INCODER*:* [35] introduced INCODER as a unified generative model that is trained to generate code via left-to-right generation as well as via infilling. The authors evaluate on variable renaming in a zero-shot setting, rendering the model a suitable candidate for our comparison. To produce input sequences, we use the released code in the infilling setting. Contrary to our approach in Section III-B, the INCODER model uses mask tokens only once (e.g. `MASK1`, `MASK2`, ...), one for each occurrence of an identifier. Similarly, we experimented with aggregating the individual scores of each identifier occurrence using the minimal, maximal and average score. We found mean pooling to give the best result.

C. Hardware and Training

For pre-training, we used 8 A100 GPUs and trained for a total of 1M stochastic gradient descent steps. We optimize the log-likelihood from Equation 1, averaged over all tokens in the batch, using the AdamW [54] optimizer with a linear warmup for 30k steps, followed by polynomial decay a learning rate peak of 0.0002. The batch size was dynamically set to 6000 tokens per batch. This model was used directly in the generative approach without further modification.

We fine-tuned the discriminative approaches on a single A6000 GPU. We swept over the optimal learning rate and used early stopping on validation F1-score of our fine-tuning dataset. Finally, we ran the tests on guideline violation only once at the very end.

TABLE IV
COMPARISON BETWEEN APPROACHES AND STATE-OF-THE-ART.

Model	Method	#Parameters	MAP
Baseline	Random	-	18.18
INCODER	Generative	1.3B	25.27
GRAPHCODEBERT	Discriminative	125M	46.32
CODEDOCTOR	Discriminative	110M	52.57
CODEDOCTOR	Generative	247M	**62.00**

VI. RESULTS

We compare the different versions and scoring techniques of our CODEDOCTOR model in Section VI-A, compare them with other recent code transformers in VI-B, and conduct a detailed guideline-specific analysis of our results in Section VI-C.

A. Scoring Methods for Generative Rating

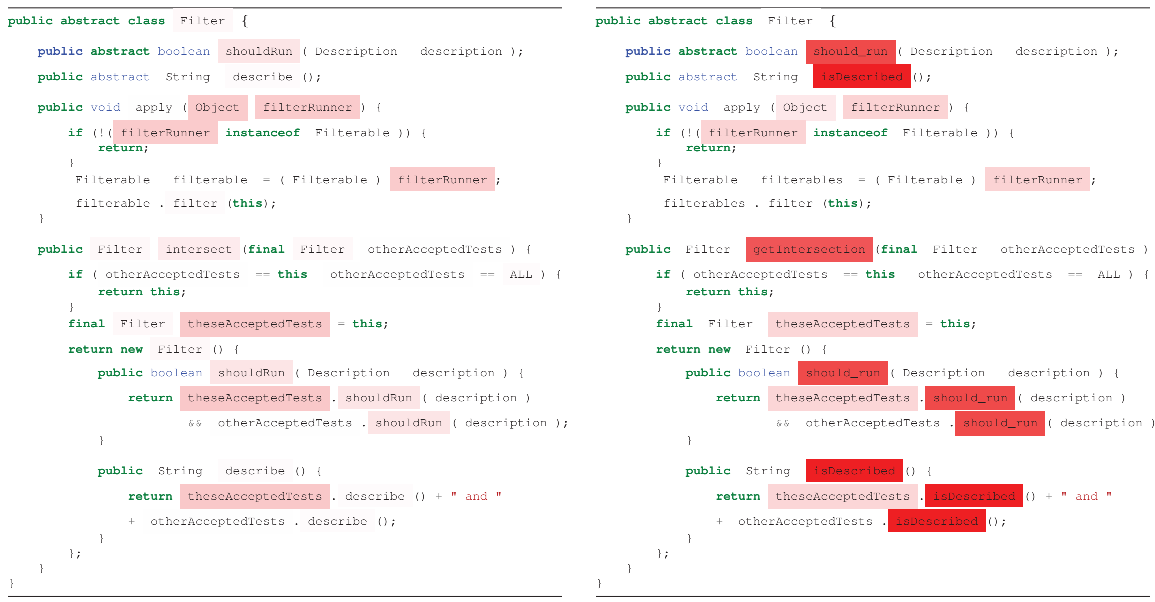
We first focus on the generative rating and compare its different scoring methods and the impact of the amount of masking in Table III. As outlined in Section III-B, we score the given identifier either by perplexity or by likelihood ratio (compare Equations (2) and (3)) and mask in the input either only the identifier or all identifiers together.

As Table III shows, we find our model to outperform INCODER significantly (these findings will be discussed in-depth in the next section). Also, we find that the performance is most often higher when only a single identifier is masked in context. Especially the Likelihood-ratio benefits from mask-single and outperforms perplexity by around 6 percentage points.Instead, when using mask-all, the performance of Likelihood-ratio degenerates by more than 30%. We found that generation simply becomes too unreliable when masking all identifiers at once, such that likelihood scores – and with them their ratio – become unstable.

From an efficiency perspective, it should be noted that – to compute the likelihood ratio – we need to explicitly *generate* the best version for each identifier (multiple forward passes) before scoring the target identifier (one forward pass). Also, note that using mask-single instead of mask-all requires multiple forward passes (one per identifier to assess), while for mask-all a single forward pass suffices. Therefore, for practical settings in which efficiency matters, perplexity + mask-all offers an alternative at $\approx$10 percentage points less MAP. Throughout the next experiments, we will report results for the best-performing approach, likelihood ratio + mask-single. Figure 2 shows an example code snippet from our dataset, where the identifiers are scored with this approach and color-coded accordingly.

B. Comparison with Other Language Models

Our results in Table IV show that our CODEDOCTOR models outperform GRAPHCODEBERT and INCODER both in the discriminative and in the generative approach. Our best approach yields 62% MAP on our evaluation data compared to 25.3% for INCODER and 46.3% for GRAPHCODEBERT.



```
public abstract class Filter {

    public abstract boolean shouldRun ( Description description );

    public abstract String describe ();

    public void apply ( Object filterRunner ) {

        if (!( filterRunner instanceof Filterable )) {
            return;
        }
        Filterable filterable = ( Filterable ) filterRunner ;

        filterable . filter (this);
    }

    public Filter intersect (final Filter otherAcceptedTests ) {

        if ( otherAcceptedTests == this otherAcceptedTests == ALL ) {
            return this;
        }
        final Filter theseAcceptedTests = this;

        return new Filter () {

            public boolean shouldRun ( Description description ) {

                return theseAcceptedTests . shouldRun ( description )

                        && otherAcceptedTests . shouldRun ( description );
            }

            public String describe () {

                return theseAcceptedTests . describe () + " and "

                    + otherAcceptedTests . describe ();
            }
        };
    }
}
```

```
public abstract class Filter {

    public abstract boolean should_run ( Description description );

    public abstract String isDescribed ();

    public void apply ( Object filterRunner ) {

        if (!( filterRunner instanceof Filterable )) {
            return;
        }
        Filterable filterables = ( Filterable ) filterRunner ;

        filterables . filter (this);
    }

    public Filter getIntersection (final Filter otherAcceptedTests ) {

        if ( otherAcceptedTests == this otherAcceptedTests == ALL ) {
            return this;
        }
        final Filter theseAcceptedTests = this;

        return new Filter () {

            public boolean should_run ( Description description ) {

                return theseAcceptedTests . should_run ( description )

                        && otherAcceptedTests . should_run ( description );
            }

            public String isDescribed () {

                return theseAcceptedTests . isDescribed () + " and "

                    + otherAcceptedTests . isDescribed ();
            }
        };
    }
}
```

Fig. 2. Example from our dataset. On the left the (shortened) file without guideline violations, on the right the violated version. We highlighted identifiers by scaling the likelihood-ratio score of the CODEDOCTOR model linearly between 1-50. Note that even in the original the model finds some identifiers suspicious (left). However it spots most guideline violations (right) but fails to detect the identifier `filterables` (which violates Guideline 13).

For the generative approach, we find our model to outperform INCODER significantly, regardless of the scoring or masking method used. We assume that this is due to several reasons: First, while INCODER needs to introduce a new mask token for each occurrence of an identifier, our model considers all instances of a masked identifier for a prediction. This enables CODEDOCTOR to exploit information shared across multiple occurrences of the same identifier to its advantage. Additionally, INCODER's tokenization spans tokens across whitespace and punctuation symbols to represent common code idioms as single tokens. The authors report a slight decrease in performance when breaking tokenization at word boundaries [35], which is needed for predicting single identifier names. We hypothesize, however, that our model's key advantage is its pre-training objective, which explicitly includes identifier deobfuscation as one of the tasks, enabling the model to generate reasonable identifier names without further fine-tuning. In comparison, we have observed that INCODER tends to generate lengthy passages of source code rather than single identifiers.

In the discriminative approach, our CODEDOCTOR encoder outperforms GRAPHCODEBERT with a mean average precision of 52.6% compared to 46.3%, though the difference in performance is not as pronounced as for the generative approach. Again, we attribute the advantage to the fact that our pre-training is more closely aligned with the target task.

C. Guideline-Specific Analysis

Figure 3 illustrates the MAP scores by guideline, including a random baseline, the generative approach (by perplexity and by likelihood-ratio, both using the mask-single setting), and the discriminative approach. Overall, the MAP scores of our models vary strongly (from 29% to 91%), indicating that the language models cope differently well with different guidelines. Our models perform about 3 times better than the random baseline on average.

We investigate the performance across the four different types of guidelines (syntax, vocabulary, data type, method name), and find the models to perform similarly well over the four categories, with a slight advantage for method naming guidelines (mean MAP is 0.66 using the likelihood ratio) and a slight disadvantage for data type guidelines (mean MAP is 0.56 using the likelihood ratio). Particularly, note that supposedly "simple" syntax-based guidelines – e.g. rules regarding abbreviations (Guidelines 2,3) or overly long identifiers (Guidelines 7,8) – are not covered better than the others. In contrast, some guidelines that require rather complex semantic reasoning across the code are amongst the highest-scored guidelines, e.g. Guideline 13 (use a large vocabulary, MAP 0.75) and Guideline 28 (method name vs. return type, MAP 0.72). This indicates a potential for combining LM-based scoring systems with rule-based ones. Our evaluation also reveals that rather than the guideline *type*, another factor appears to have more impact, namely how often we expect the guideline to be violated in practical open-source development. For example, Guideline 3 (single-letter abbreviation) could be expected to be violated often in practice. Other guidelines with a tendency to be violated include 14 (omit type information), or 21 (don't use get/is/has for methods with side effects). Therefore, we inspected for each guideline how well we expect open-source code (and hence the language model) to adhere to it. Our expectations match well with our results, e.g. Guidelines 3, 14, 21 are found amongst the more challenging

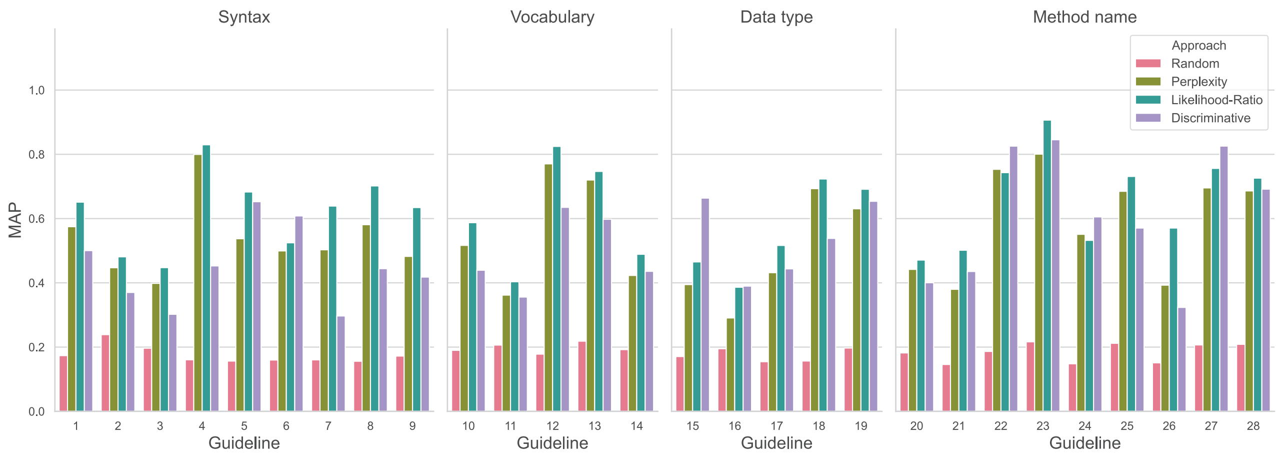


Fig. 3. Results per guideline. The guideline IDs correspond to those in Table I.

ones for our models.

When comparing the different scoring methods across the guidelines, we see that the likelihood ratio not only yields the best performance overall (mean MAP 0.62), but outperforms the other methods quite consistently. The FASTTEXT sampling process seems to produce negative samples which work well on certain guidelines. We find guidelines (15, 16, 27) notably, which deal with single values vs collections, and where the discriminative outperforms the generative approach. This is in line with an in-depth inspection of the FASTTEXT training data: Often, given an identifier such as `customer`, FASTTEXT-based drop-in replacements include the corresponding singular/plural (here: `customers`). A simple regexp-based check revealed that amongst 306.000 replacements in the training set, about 5.400 corresponded to such singular/plural replacements (≈ 1.8 percent), which allows the model to pick up this guideline well. This also indicates the potential for exploring other methods for sampling fake identifiers, which are more closely aligned to dedicated guidelines.

D. Threats to Validity

We utilized a limited set of guidelines for annotation, meaning our results may not extend to all naming guidelines. However, our guidelines do address syntax, semantics, vocabulary, and method naming, all key factors impacting code comprehension as evidenced in previous studies [1], [3]–[5]. The annotation process may have limitations due to individual perceptions of quality. We countered this by providing clear instructions to annotators and cross-checking their work.

Our evaluation data was sourced from public data on GitHub. The model's pre-training was also conducted using data mined from GitHub, which is known to contain a high percentage of forks and (near-)duplicates. Therefore, we cannot rule out that parts of the evaluation data were already encountered by the model during training, which is a common challenge with large language models pre-trained on code. Note, however, that our evaluation is entirely based on guidelines and does not aim at reproducing original identifiers.

VII. CONCLUSION

In conclusion, we show that language models on code can be used for identifier quality estimation. Our self-supervised methods, a generative one and a discriminative one, can successfully spot violations of common identifier naming guidelines. Our evaluation on a manually annotated dataset demonstrates the potential and limitations of language models in this task, and our approach outperforms other recently published code transformers.

In future work, we aim to further improve the performance of CODEDOCTOR. Firstly, we will investigate the extent to which CODEDOCTOR aligns with developers' cognitive load and perception of identifier quality. To improve our model, we will consider fine-tuning with silver standard training data as recently bootstrapped from code reviews in [14]. Additionally, one could try ensemble methods including a rule based detector, as well as the discriminative approach (which performed better on specific guidelines). Furthermore, to address project-specific coding conventions, one could consider training a local model to reduce false positives. Even though our current evaluation focuses on Java, our pre-training features 16 different programming languages from a large-scale set of open-source projects. Our generative model can be directly applied to these other programming languages without further fine-tuning. Conducting this evaluation – which would involve more manual annotation – is another interesting direction for future work.

DATA AVAILABILITY

The replication package [17] for our study includes all necessary code and data for reproducing our results. In particular, we provide the code for training and evaluating the models, the trained models, the dataset used for evaluation, and the predicted scores generated by our different approaches.

REFERENCES

[1] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2140–2178, 2020.

[2] E. M. Gellenbeck and C. R. Cook, "An investigation of procedure and variable names as beacons during program comprehension," in *Empirical studies of programmers: Fourth workshop*. Ablex Publishing, Norwood, NJ, 1991, pp. 65–81.

[3] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE international conference on program comprehension (ICPC'06)*. IEEE, 2006, pp. 3–12.

[4] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.

[5] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, R. Capilla, R. Ferenc, and J. C. Dueñas, Eds. IEEE Computer Society, 2010, pp. 156–165. [Online]. Available: https://doi.org/10.1109/CSMR.2010.27

[6] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. C. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 140–150. [Online]. Available: https://doi.org/10.1145/3106237.3106268

[7] B. Caprile and P. Tonella, "Restructuring program identifier names," in *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*. IEEE Computer Society, 2000, pp. 97–107. [Online]. Available: https://doi.org/10.1109/ICSM.2000.883022

[8] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and NLP to promote a consistent usage of identifiers," in *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*. IEEE Computer Society, 2017, pp. 81–90. [Online]. Available: https://doi.org/10.1109/SCAM.2017.17

[9] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2327–2339. [Online]. Available: https://doi.org/10.1145/3510003.3510162

[10] G. Li, H. Liu, and A. S. Nyamawe, "A survey on renamings of software entities," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 41:1–41:38, 2021. [Online]. Available: https://doi.org/10.1145/3379443

[11] S. Sengamedu and H. Zhao, "Neural language models for code quality identification," in *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2022, Singapore, Singapore, 18 November 2022*, M. Cordy, X. Xie, B. Xu, and B. Stamatia, Eds. ACM, 2022, pp. 5–10. [Online]. Available: https://doi.org/10.1145/3549034.3561175

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021.

[13] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, 2018. [Online]. Available: https://doi.org/10.1145/3212695

[14] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "Automated variable renaming: Are we there yet?" *CoRR*, vol. abs/2212.05738, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2212.05738

[15] P. Hilton and F. Hermans, "Naming guidelines for professional programmers." in *PPIG*, 2017, p. 19.

[16] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[17] "Replication package," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7612762

[18] R. P. L. Buse and W. Weimer, "A metric for software readability," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 121–130. [Online]. Available: https://doi.org/10.1145/1390630.1390647

[19] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[20] F. P. B. Jr., "No silver bullet - essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987. [Online]. Available: https://doi.org/10.1109/MC.1987.1663532

[21] V. Antinyan, "Evaluating essential and accidental code complexity triggers by practitioners' perception," *IEEE Software*, vol. 37, no. 6, pp. 86–93, 2020.

[22] A. Abbad-Andaloussi, "On the relationship between source-code metrics and cognitive load: A systematic tertiary review," *Journal of Systems and Software*, p. 111619, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223000146

[23] J. Sweller, "Cognitive load theory," in *Psychology of learning and motivation*. Elsevier, 2011, vol. 55, pp. 37–76.

[24] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 402–413.

[25] A. Abbad-Andaloussi, T. Sorg, and B. Weber, "Estimating developers' cognitive load at a fine-grained level using eye-tracking measures," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 111–121.

[26] T. Sorg, A. A. Andaloussi, and B. Weber, "Towards a fine-grained analysis of cognitive load during program comprehension," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 748–752. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00092

[27] S. P. Reiss, "Automatic code stylizing," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 74–83. [Online]. Available: https://doi.org/10.1145/1321631.1321645

[28] F. Corbo, C. D. Grosso, and M. D. Penta, "Smart formatter: Learning coding style from existing source code," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE Computer Society, 2007, pp. 525–526. [Online]. Available: https://doi.org/10.1109/ICSM.2007.4362682

[29] P. Jablonski and D. Hou, "Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange, ETX 2007, Montreal, Quebec, Canada, October 21, 2007*, L. Cheng, A. Orso, and M. P. Robillard, Eds. ACM, 2007, pp. 16–20. [Online]. Available: https://doi.org/10.1145/1328279.1328283

[30] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for javascript," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds. ACM, 2013, pp. 323–338. [Online]. Available: https://doi.org/10.1145/2509136.2509520

[31] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE 2010, Cape Town, South Africa, May 4, 2010*, R. Holmes, M. P. Robillard, R. J. Walker, and T. Zimmermann, Eds. ACM, 2010, pp. 1–5. [Online]. Available: https://doi.org/10.1145/1808920.1808921

[32] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds.

IEEE Computer Society, 2012, pp. 837–847. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227135

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[34] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," Sep. 2021. [Online]. Available: http://arxiv.org/abs/2109.00859

[35] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," Apr. 2022.

[36] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, S. Chaudhuri and C. Sutton, Eds. ACM, 2022, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3520312.3534862

[37] B. Ray, V. J. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. T. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 428–439. [Online]. Available: https://doi.org/10.1145/2884781.2884848

[38] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 281–293. [Online]. Available: https://doi.org/10.1145/2635868.2635883

[39] ——, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[40] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00019

[41] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1372–1384. [Online]. Available: https://doi.org/10.1145/3377811.3380926

[42] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 574–586. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00060

[43] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin, "Learning to recommend method names with global context," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1294–1306. [Online]. Available: https://doi.org/10.1145/3510003.3510154

[44] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. D. Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4818–4837, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3128234

[45] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 163–174. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00027

[46] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[48] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[49] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://doi.org/10.1162/tacl_a_00051

[50] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., Feb. 2021, pp. 14 967–14 979. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/7d6548bdc0082aacc950ed35e91fcccb-Paper.pdf

[51] J. Villmow, V. Campos, A. Ulges, and U. Schwanecke, "Addressing leakage in self-supervised contextualized code retrieval," in *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, N. Calzolari, C. Huang, H. Kim, J. Pustejovsky, L. Wanner, K. Choi, P. Ryu, H. Chen, L. Donatelli, H. Ji, S. Kurohashi, P. Paggio, N. Xue, S. Kim, Y. Hahm, Z. He, T. K. Lee, E. Santus, F. Bond, and S. Na, Eds. International Committee on Computational Linguistics, 2022, pp. 1006–1013. [Online]. Available: https://aclanthology.org/2022.coling-1.84

[52] K. Clark, M. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: pre-training text encoders as discriminators rather than generators," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=r1xMH1BtvB

[53] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[54] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg6RiCqY7