# Probing API Name Knowledge in Pre-trained Code Models

Anonymous ACL submission

## Abstract

Recent advances in pre-trained code models, like CodeBERT and Codex, have demonstrated remarkable performance across diverse tasks. However, the accurate and clear use of APIs is vital for optimal program functionality, necessitating a deep understanding of API fully qualified names both structurally and semantically. Despite their prowess, current models often falter in suggesting appropriate APIs during code generation, with the underlying reasons remaining largely unexplored. To bridge this gap, we leverage the knowledge probing technique and employ cloze-style tests to gauge the knowledge embedded within these models. Our in-depth analysis assesses a model's grasp of API fully qualified names from two angles: API call and API import. The results shed light on the strengths and weaknesses of existing pre-trained code models. We posit that integrating API structure during pre-training can enhance API usage and code representation. This research aims to steer the evolution of code intelligence and set the course for subsequent investigations.

## 1 Introduction

Recent advances in code intelligence have incorporated pre-training techniques, where models are pre-trained on large-scale unlabeled source code corpora to learn the code's representation and semantics. The pre-trained code models, such as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021), can be fine-tuned for various downstream code tasks, such as code completion and translation (Allamanis et al., 2018; Xu et al., 2022; Niu et al.). Despite the improvement, there is still a significant performance gap between their performance and that of human developers when it comes to using APIs correctly. For instance, several studies have demonstrated that even state-of-the-art pre-trained code models, such as Codex (Chen et al., 2021) and GPT-4 (OpenAI, 2023), inevitably hallucinate or misuse the APIs during code generation (Lai et al., 2022; Yu et al., 2023; Zhong and Wang, 2023). However, few studies have investigated the reasons behind these models' poor API usage performance.

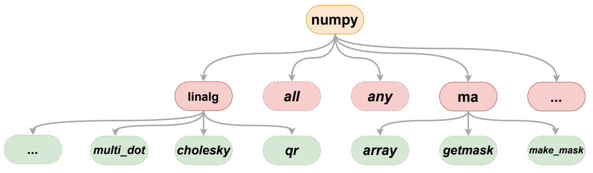

Figure 1: An example of module design of <numpy> library.

Basically, correct API usage depends on two types of knowledge: (1) how to invoke an API, and (2) which API to invoke, with the former being a fundamental step towards the latter. To invoke an API, one must have knowledge of the code grammar of importing libraries and composing the correct API name based on the import and call statements. However, code models are not explicitly instructed by code grammar. Although we often assume that the models can learn to use APIs effectively by observing a large number of examples, this assumption has been poorly validated. Therefore, we pose the following question: *Do Pre-trained Code Models Possess Knowledge of Correct API Names?*

APIs are often identified by their fully qualified names, encompassing the function name and its associated package, module, or class. For simplicity, we will use "module" to denote all these entities. Libraries structure APIs into hierarchical modules to guide developers in comprehending the available features. This hierarchy is such that higher-level modules act as parents to their direct sub-modules, with APIs as the terminal nodes. The full name of an API is derived by traversing this hierarchy from the root to the terminal node, connecting each name with dots. How an API name is invoked depends on the module's level of import and potential aliasing. While intuitive for developers, this convention can be intricate for code models to grasp.

Furthermore, the structure of API names offers insights into code modularization and namespace design. If models can effectively represent these names, they might aid in refining API namespace design by suggesting more accurate and relevant options. Take, for instance, the Python library `numpy` for array and matrix operations. As depicted in Figure 1, `numpy` has modules like `linalg` for linear algebra functions such as `multi_dot`, `cholesky`, and `qr`, and `ma` for masked array operations. This namespace design can inform the creation of related or analogous libraries in other languages.

In our study, we target the interpretability of code models, introducing a probing task for fully qualified API names. We unveil `INK`, an automated framework, to assess code models using cloze-style quizzes derived from commonly used source code corpora. We ensure models are tested on familiar knowledge, reflecting their training constraints. We then craft import and API call statements, masking tokens at module levels. For example, with the `<multi_dot>` module in Figure 1, we generate quizzes like `<numpy.[MASK]alg.multi_dot>` from the API call (`<numpy.linalg.multi_dot>`) and its import (`<from numpy.linalg import multi_dot>`). The goal is for models to predict the masked token. To further understand pre-trained code models' proficiency in API names, we outline research questions (RQs) to guide future advancements:

- **RQ1: How well do code models understand API calls and imports?** The code models are assessed on their prediction of masked import or API call statements. The findings can help identify areas where code models may struggle and guide improvements in their API name learning via the understanding of this question.

- **RQ2: Do code models understand API import aliases?** This RQ further assesses code models' understanding of API import aliases. The quizzes are designed based on an alias-defining import statement followed by an API call based on the imported module. Tokens are selectively masked out for the statements.

- **RQ3: How well can code models memorize and generalize on API names?** We divide APIs into two groups based on whether they are seen during the training phase. The outcomes indicate if the code models possess robust generalization capabilities and appropriate memorization skills. Models' ability to apply learned knowledge to unseen APIs would be helpful for API namespace design.

For the evaluation, we construct the first benchmark on Python API name knowledge, `PyINK`, and analyze 10 pre-trained code models, including the variants of CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), PLBART (Ahmad et al., 2021). Our work is complementary to the development of more advanced techniques for modeling API representations in code, thereby enhancing the efficiency and accuracy of code models. Additionally, the insights gained from this work can pave the way for a deeper understanding of how pre-training impacts the performance of code models, facilitating more informed design decisions in this domain. In summary, our main contributions include[1]:

- A cloze-style evaluation framework `INK`, to probe and benchmark the knowledge of API names in pre-trained code models.

- An implementation of `INK`, which lowers the bar for designing probing techniques on API name knowledge.

- An evaluation dataset based on `INK`, `PyINK`, containing diverse API name cloze-style pop quizzes.

- A comprehensive study on understanding the Python API name knowledge of pre-trained code models via `PyINK`.

## 2 Background and Related Work

This section provides a comprehensive overview of the background and related work that form the foundation of our research.

### 2.1 Pre-trained Code Models

Pre-trained language models like BERT (Devlin et al., 2019) have revolutionized natural language processing tasks by transferring knowledge from vast corpora (Qiu et al., 2020). Similarly, the software domain has benefited from models such as CodeBERT, GraphCodeBERT, and PLBART,

[1]Resources are available at `https://anonymous.4open.science/r/API-Name-Probing`.

which harness software's inherent naturalness (Hindle et al., 2016). These models are adept at tasks ranging from code completion to code summarization.

### 2.2 Knowledge Probing

The evaluation of internal representations and knowledge of language models is a fundamental and critical process, which involves the technique of knowledge probing (Liu et al., 2023). This approach entails presenting a set of questions or statements to the model to assess its understanding of specific concepts or relationships. The inputs are typically presented as cloze sentences with particular concepts or relationships masked as discrete prompts to test the model's performance. We formalize the knowledge probing approach by considering the input cloze sentence, denoted as $S$, where "Alan Turing was born in `[MASK]`" is an example.

Formally, we define the knowledge probing approach as follows,

$$f(S) = \frac{1}{|S|}\sum_{k=1}^{|S|} log(P(t_k)|S;\theta), t_k \in \mathcal{V}(\theta) \quad (1)$$

where $\theta$ represents the model parameters, $\mathcal{V}(\theta)$ denotes the vocabulary learned by the language model, and $t_k$ is a token inside the model's vocabulary $\mathcal{V}(\theta)$. The contextualized likelihood $f(S)$ represents the possibility of replacing `[MASK]` with the token $t_k$ as per the model's prediction. The final prediction corresponds to the token $t_k$ that maximizes $f(S)$.

### 2.3 Generation-based API Recommendation

API recommendation, which suggests specific APIs from natural language (NL) queries, has been approached in two primary ways: (1) Rank-based and (2) Generation-based recommendations. The former leverages knowledge graphs or bases to identify suitable APIs by interpreting the semantic content of the NL query (Huang et al., 2018; Rahman et al., 2016; Thung et al., 2013). In contrast, the latter, more aligned with our work, employs deep learning to generate API sequences from NL queries.

DeepAPI (Gu et al., 2016) pioneered this by treating it as a machine translation challenge, using an end-to-end supervised model. Later studies explored fine-tuning pre-trained code models for this task (Martin and Guo, 2022; Hadi et al., 2022). While these models show improved performance, they have limitations: (1) They are fine-tuned on limited APIs, hindering generalization. (2) The fine-tuning process lacks transparency in capturing API knowledge. (3) The task relies on NL inputs, which only gauge the semantic understanding of API sequence use. (4) Evaluation is predominantly through the BLEU score (Papineni et al., 2002), which assesses similarity but not necessarily synthesis accuracy.

## 3 `INK`: An evaluation framework of API Name Knowledge

### 3.1 Motivation

Previous research in natural language processing has utilized cloze sentences for token prediction as a means of interpreting the knowledge encoded by pre-trained language models. Building upon this work, we examine probing API name knowledge in CodeBERT-MLM - a variant of CodeBERT pre-trained solely on mask language modeling - with cloze-style sentences serving as pop quizzes, as depicted in Figure 2. We use `<tensorflow.compat.v2.boolean _mask>` as an example and transform it into a cloze-style pop quiz, as shown in Figure 2. In this study, we define the API module levels as each hierarchical level within the fully qualified name, separated by a period. There are four module levels in the API call statement: (1) `<tensorflow>` representing the top module level, (2) `<compat>` as the second module level, (3) `<v2>` as the third module level, and (4) `<boolean_mask>` as the last call level. For each level, we request CodeBERT-MLM to fill in the blank via **first** token prediction, as determined by its tokenizer. As our results demonstrate, CodeBERT-MLM correctly predicts the masked token on the first attempt, except for the third level of `<v2>`. We contend that code models, such as CodeBERT-MLM, can learn API names during function pre-training.

Given that `<tensorflow.compat.v2.boolean _mask>` can be reconstituted to the form of API import statement, we then investigate how well CodeBERT-MLM understands the API import statements. As demonstrated in Figure 2, we transform the API import statement into four cloze templates by masking the first token of each module level, similar to the ones of the API call. From the results illustrated in Figure 2, we discover that predicting some API modules at the first shot is challenging for CodeBERT-MLM, unlike the case

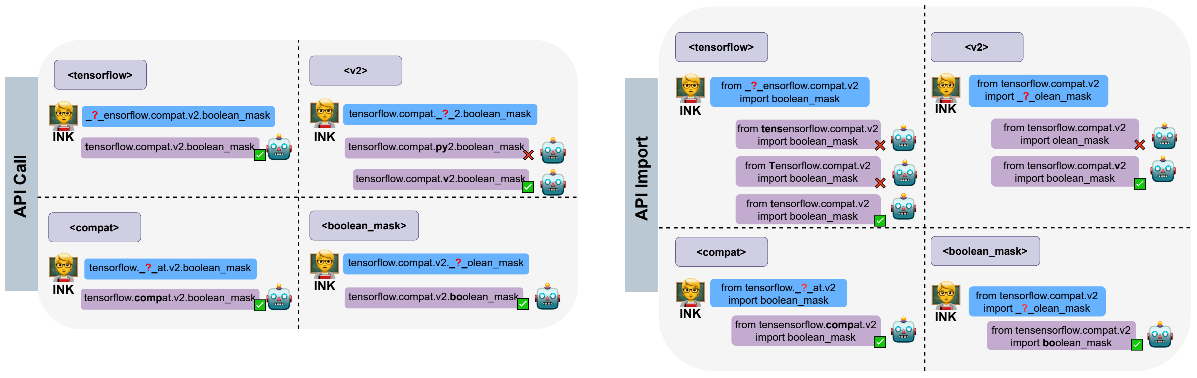
API Call
<tensorflow>
INK
_?_ensorflow.compat.v2.boolean_mask
tensorflow.compat.v2.boolean_mask
<v2>
INK
tensorflow.compat._?_2.boolean_mask
tensorflow.compat.py2.boolean_mask
tensorflow.compat.v2.boolean_mask
<compat>
INK
tensorflow._?_at.v2.boolean_mask
tensorflow.compat.v2.boolean_mask
<boolean_mask>
INK
tensorflow.compat.v2._?_olean_mask
tensorflow.compat.v2.boolean_mask
API Import
<tensorflow>
INK
from _?_ensorflow.compat.v2 import boolean_mask
from tensensorflow.compat.v2 import boolean_mask
from Tensorflow.compat.v2 import boolean_mask
from tensorflow.compat.v2 import boolean_mask
<v2>
INK
from tensorflow.compat.v2 import _?_olean_mask
from tensorflow.compat.v2 import olean_mask
from tensorflow.compat.v2 import boolean_mask
<compat>
INK
from tensorflow._?_at.v2 import boolean_mask
from tensensorflow.compat.v2 import boolean_mask
<boolean_mask>
INK
from tensorflow.compat.v2 import _?_olean_mask
from tensensorflow.compat.v2 import boolean_mask

Figure 2: API fully qualified names such as `<tensorflow.compat.v2.boolean_mask>` can be formalized into the cloze-style tests from two perspectives, API calls and API imports. The example shows the top-$k$ predictions of CodeBERT-MLM model for each test. Note that the dialogues are for illustration only.

of API call probing. This behavior suggests that code models possess varying degrees of knowledge in API import and API call statements.

Having identified several potential patterns from our preliminary knowledge probing study, which provides clues to API name knowledge, we find it necessary to further explore this phenomenon through quantitative analysis and systematic evaluation. Motivated by the aforementioned observations, this paper investigates whether pre-trained code models learn the API names and to what extent they store API name knowledge, by conducting knowledge probing as the pop quiz. Specifically, we analyze two perspectives of API names, i.e., API call and API import, within the purview of code models.

### 3.2 API Name Knowledge Probing Template

We consider three main types of transformation of evaluating API name knowledge: **API calls**, **API imports** and **API import aliases**. As aforementioned in Section 3.1, cloze-style pop quizzes are structured based on each call level split by the delimiters of ".", "from" and "import". To benchmark models fairly, we construct pop quizzes by unifying the entire vocabulary of each model. For all the evaluation, we follow previous work of knowledge probing in language models (Petroni et al., 2019) and choose to evaluate the prediction of a single token masking in the pop quiz. We provide the detailed design of each process as follows.

#### 3.2.1 Evaluation Design on API Call

We treat each API call as a modular pattern on the basis of each module level. To formalize the pop quiz construction on API calls, an example of API call `<A.B.C>` and a code model $\mathcal{M}$ are given to demonstrate the workflow. The model $\mathcal{M}$ firstly tokenizes the API as follows,

$$\mathcal{M}(\texttt{<A.B.C>}) \rightarrow$$

$$\Big\{\{t_1^A, t_2^A, \ldots, t_{N_a}^A\}, t^{dot}, \{t_1^B, t_2^B, \ldots, t_{N_b}^B\}, t^{dot},$$

$$\{t_1^C, t_2^C, \ldots, t_{N_c}^C\}\Big\}$$

where each $t$ represents the token produced by the model $\mathcal{M}$, and $N$ represents the length of tokens in each level. For each level, the tokens are grouped by $\{\ldots\}$. When converting the tokenized API to the pop quiz, we mask a specific token by replacing $t$ with a "`[MASK]`" in each level. To visualize the pop quiz input, we mask the last token in the second module level of `<A.B.C>` as follows:

$$\texttt{<A.B.C>} \rightarrow \texttt{<A.B'[MASK].C>} \rightarrow \texttt{<A.B'\_\_.C>}$$

where $B'$ is the concatenation of $\{t_1^B, \ldots, t_{N_b-1}^B\}$. We prompt the model $\mathcal{M}$ to fill in the blank of `<A.B'__.C>` via mask prediction.

#### 3.2.2 Evaluation Design on API Import

We explore the evaluation design on the API import statement of "`from` ... `import` ...". Similarly, we consider the example of `<from A.B import C>`. Using the model $\mathcal{M}$ to tokenize the API import, we can devise the following tokens:

$$\mathcal{M}(\texttt{<from A.B import C>}) \rightarrow$$

$$\Big\{t^{from}, \{t_1^A, t_2^A, \ldots, t_{N_a}^A\}, t^{dot},$$

$$\{t_1^B, t_2^B, \ldots, t_{N_b}^B\}, t^{import}, \{t_1^C, t_2^C, \ldots, t_{N_c}^C\}\Big\}$$

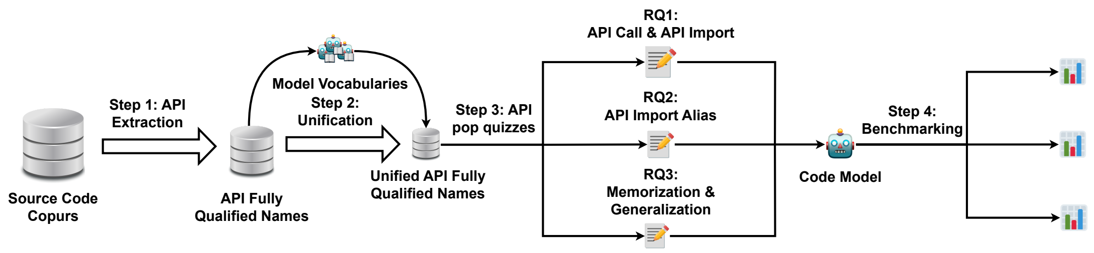

Figure 3: An overview of the `INK` framework.

We visualize an example of API import quiz, where the first token in the bottom level of `<from A.B import C>` is masked:

`<from A.B import C>`

$\rightarrow$ `<from A.B import [MASK]C'>`

$\rightarrow$ `<from A.B import _C'>`

where $C'$ is the concatenation of $\{t_2^C, \ldots, t_{N_c}^C\}$. We probe the model $\mathcal{M}$ to fill in the blank of `<from A.B import _C'>` via mask prediction.

#### 3.2.3 Evaluation Design on API Import Alias

We note that import aliases are supported in some programming languages, such as Python. For example, "`import` ... as ..." and "`from` ... `import` ... as ..." are the typical import alias syntax. Therefore, we further examine pre-trained code model's understanding of the aliases of API calls after packages and libraries imports. We illustrate the design choice via the example of `<import A as K \n K.B.C>`, where `<K>` is the alias and `K.B.C` is the API call statement. After being tokenized by the model $\mathcal{M}$, the example is formalized as follows:

$$\mathcal{M}(\texttt{<import A as K \textbackslash n K.B.C>}) \rightarrow$$

$$\Big\{t^{import}, \{t_1^A, t_2^A, \ldots, t_{N_a}^A\}, t^{as}, \{t_1^K, t_2^K, \ldots, t_{N_a}^K\},$$

$$t^{newline}, \{t_1^K, t_2^K, \ldots, t_{N_a}^K\}, t^{dot},$$

$$\{t_1^B, t_2^B, \ldots, t_{N_b}^B\}, t^{dot}, \{t_1^C, t_2^C, \ldots, t_{N_c}^C\}\Big\}$$

We then convert the example to the following API alias pop quiz with the masked last token of `<C>`:

`<import A as K \n K.B.C>`

$\rightarrow$ `<import A as K \n K.B.C'[MASK]>`

$\rightarrow$ `<import A as K \n K.B.C'_>`

where $C'$ is the concatenation of $\{t_1^C, \ldots, t_{N_c-1}^C\}$. We probe the model $\mathcal{M}$ to fill in the blank of `<import A as K \n K.B.C'_>` via mask prediction. Note that we only mask the tokens in the part of the API call after the alias name.

#### 3.2.4 Evaluation of Code Models: A Unified Vocabulary Approach

Code models may tokenize the same API call or import statement differently due to variations in their respective vocabularies. For example, an API call such as `<A.B.C>` may be tokenized as a single token by one model $\mathcal{M}_A$ with vocabulary $\mathcal{V}_A$, while another model $\mathcal{M}_B$ with vocabulary $\mathcal{V}_B$ may tokenize it into multiple tokens. To ensure a fair comparison, we generate models over a unified vocabulary that is the intersection of the vocabularies of all considered code models. We define the unified vocabulary as the set of tokens that are presented in the vocabularies of all considered code models.

To evaluate the performance of code models on this unified vocabulary, we categorize the evaluation of each name level into two types: (1) *partial module masking* and (2) *full module masking*. For the former, we tokenize the API call and import statements into multiple tokens and mask only one of the tokens. In contrast, when code models segment the entire module level as a single token, we denote it as the full module masking.

#### 3.2.5 Benchmarking Code Models via Knowledge Probing

We utilize the knowledge probing technique to evaluate and assess the performance of code models effectively. The primary objective of this approach is to delve into the code models' comprehension of API names and their proficiency in generating precise and contextually fitting tokens within those names. To achieve this, we present cloze-style quizzes where certain tokens are intentionally masked, prompting the code model to predict the tokens for the masked positions. To gauge the accuracy of these predictions, we compare them against the ground-truth tokens in the API names. This benchmarking process allows us to determine the correctness of predictions.

## 4 Experiment Setup

We introduce the basic experimental setup about the datasets and models, and the evaluation metrics used throughout the evaluation.

### 4.1 `PyINK`: Evaluation on Python API Name Knowledge

| | First Token | Last Token | Full | Total |
|---|---|---|---|---|
| API Call | 108,863 | 111,373 | 69,349 | 289,585 |
| API Import | 108,863 | 111,373 | 69,349 | 289,585 |
| API Import Alias | 7,385 | 7,122 | 3,464 | 17,971 |

Table 1: Overview of the number of pop quizzes in `PyINK`.

We adopt the widely-used pre-training corpus, CSNet (Husain et al., 2019), which is a collection of datasets and benchmarks for semantic code retrieval, containing functions and their corresponding comments of six programming languages extracted from GitHub repositories, to evaluate the API name knowledge of code models. We focus on Python set, which contains 457,461 functions. To extract Python APIs from CSNet, we use our proposed `INK` framework, which leverages static analysis on the entire file to extract API usage, following the method described in (Wang et al., 2021). To achieve this, we clone all repositories corresponding to the functions in CSNet and analyze the API usage in each function. This process results in a new benchmark, denoted as `PyINK`, which is designed to evaluate the name knowledge of Python APIs. `PyINK` contains 597,141 main pop quizzes for evaluation, and we only consider **full** module masking, or partial masking of the **first** and **last** tokens of each module level to maintain consistency.

Our approach successfully extracts 79,754 unique APIs from 8,294 Python libraries in 13,519 repositories, indicating the diverse usage of Python APIs. Moreover, to ensure a fair comparison, we unify the vocabularies in the considered code models, resulting in 289,585, 289,585, and 17,971 samples for API call, import, and import with aliases statements, respectively, in the benchmark.

### 4.2 Code Models

We conduct extensive studies on ten selected models from the variants of CodeBERT, GraphCodeBERT and PLBART. The description of each model is summarized in Table 2.

| Model | | pre-trained Dataset | Objective | #Param | Fine-tuned |
|---|---|---|---|---|---|
| CodeBERT | MLM | CSNet | MLM | 125M | ✗ |
| | MLM-Python | CSNet+CodeParrot* | MLM | 125M | ✗ |
| GraphCodeBERT | MLM | CSNet | MLM* | 125M | ✗ |
| PLBART | Base | N/A | DAE | 140M | ✗ |
| | Large | N/A | DAE | 406M | ✗ |
| | CSNet | CSNet | DAE | 140M | ✗ |
| | Sum | N/A | DAE | 140M | ✓ |
| | Gen | N/A | DAE | 140M | ✓ |
| | MT | N/A | DAE | 140M | ✓ |

Table 2: The overview of selected code models for evaluation. Note that only the train split of CSNet is used during pre-training. MLM: Masked Language Modeling. DAE: Denoising Auto-Encoding. CodeBERT-MLM-Python uses the Python split of CodeParrot (Wolf et al., 2020) for continuous pre-training. PLBART-Sum and PLBART-Gen are the PLBART-Base model fine-tuned on CSNet Python code summarization and code generation tasks, respectively. PLBART-MT adopts multitask learning and is fine-tuned on both tasks that the previous two models use.

### 4.3 Evaluation Metric

We present an evaluation methodology based on rank-based metrics in the context of API name prediction. Our approach involves computing results per test sample and means across pop quizzes, utilizing the mean precision at k ($P@k$) metric. Specifically, $P@k$ is computed as 1 if the target object is ranked among the top k results, and 0 otherwise.

## 5 Results

### 5.1 RQ1: How well do code models understand API calls and imports?

Our evaluation assesses the capability of pre-trained code models to encode knowledge of Python API names for both API calls and imports. We computed $P@k$ scores for each masking strategy and present the results in Table 4. In addition, we provide a few examples in Table 3. Firstly, we have observed that the relative performances of different code models remain consistent when we vary the value of $k$ in the $P@k$ metric. Secondly, as $k$ increases, the improvement in performance for each model becomes less significant. These observations provide strong evidence to support the effectiveness of the `PyINK` benchmark. When comparing among the model variants, our analysis reveals that

| | Test | Answer | CodeBERT-MLM-Python | GraphCodeBERT-MLM |
|---|---|---|---|---|
| API Call | stsci.tools.fileutil.buildFITS___( | Name | file [0.10], File [0.06], Image [0.04], image [0.03], Data [0.02] | )\ [0.42], ): [0.08], () [0.06], ); [0.05], ) [0.04] |
| | win32___.QueryServiceStatusEx( | service | api [0.59], gui [0.25], service [0.04], security [0.02], com [0.01] | service [0.33], Service [0.09], net [0.08], api [0.06], API [0.05] |
| | demand___.bdew.ElecSlp( | lib | 2 [0.12], lib [0.09], 1 [0.07], 3 [0.03], py [0.02] | com [0.47], org [0.17], google [0.16], com [0.03], org [0.02] |
| | cltk.prosody.latin.string____.move_consonant_right( | utils | left [0.02], table [0.02], translation [0.02], char [0.02], right [0.02] | utils [0.28], list [0.16], selection [0.10], table [0.08], util [0.04] |
| | ___fuzz.rand.randint( | gram | \n [0.40], # [0.34], py [0.09], ## [0.02], . [0.01] | # [0.26], ^ [0.05], open [0.05], ! [0.03], test [0.03] |
| API Import | from stsci.tools.fileutil import buildFITS___ | Name | \n [0.94], \n\n [0.03], File [0.00], file [0.00], _ [0.00] | File [0.38], file [0.23], Filename [0.02], Tools [0.02], Files [0.01] |
| | from win32___ import QueryServiceStatusEx | service | api [0.43], service [0.07], query [0.07], security [0.06], db [0.04] | service [0.35], file [0.22], net [0.19], db [0.02], security [0.02] |
| | from demand___.bdew import ElecSlp | lib | er [0.03], 1 [0.03], y [0.02], en [0.02], 2 [0.02] | igo [0.05], y [0.02], com [0.02], en [0.02], ache [0.02] |
| | from cltk.prosody.latin.string____ import move_consonant_right | utils | move [0.14], right [0.08], tools [0.04], translation [0.03], left [0.03] | utils [0.56], selection [0.10], util [0.06], list [0.03], table [0.02] |
| | from ___fuzz.rand import randint | gram | py [0.29], sk [0.06], fuzzy [0.05], core [0.03], ham [0.03] | ham [0.43], exp [0.12], math [0.05], http [0.04], easy [0.03] |

Table 3: Examples of prediction for CodeBERT-MLM-Python and GraphCodeBERT-MLM on API calls and imports. The last two columns reports the top five tokens generated together with the associated probabilities (in square brackets).

| | | | $P@1\uparrow$ | $P@5\uparrow$ | $P@10\uparrow$ | $P@20\uparrow$ | $P@30\uparrow$ | $P@40\uparrow$ | $P@50\uparrow$ |
|---|---|---|---|---|---|---|---|---|---|
| API Call | CodeBERT | MLM | 23.90 | 42.34 | 50.28 | 58.17 | 62.75 | 65.87 | 68.28 |
| | | MLM-Python | **29.35** | **47.51** | **55.67** | **63.61** | **68.06** | **71.06** | **73.33** |
| | GraphCodeBERT | MLM | 25.89 | 43.30 | 50.74 | 58.28 | 62.73 | 65.87 | 68.26 |
| | PLBART | Base | 1.65 | 6.58 | 10.43 | 15.71 | 19.40 | 22.44 | 24.99 |
| | | Large | 2.29 | 8.63 | 13.22 | 18.62 | 22.19 | 25.02 | 27.40 |
| | | CSNet | 2.57 | 10.78 | 15.57 | 21.49 | 25.55 | 28.76 | 31.41 |
| | | Sum | 2.11 | 9.65 | 14.46 | 19.98 | 23.53 | 26.21 | 28.46 |
| | | Gen | 4.31 | 14.51 | 20.69 | 27.21 | 31.27 | 34.23 | 36.59 |
| | | MT | 4.79 | 15.80 | 22.71 | 30.27 | 34.83 | 38.10 | 40.67 |
| API Import | CodeBERT | MLM | 26.33 | 43.87 | 51.19 | 58.63 | 62.89 | 65.85 | 68.16 |
| | | MLM-Python | 29.67 | **49.26** | **56.82** | **64.18** | **68.34** | **71.23** | **73.39** |
| | GraphCodeBERT | MLM | **30.76** | 48.24 | 55.35 | 62.50 | 66.68 | 69.55 | 71.73 |
| | PLBART | Base | 2.64 | 12.96 | 20.12 | 28.21 | 33.22 | 36.90 | 39.83 |
| | | Large | 5.66 | 18.03 | 25.72 | 34.20 | 39.18 | 42.69 | 45.44 |
| | | CSNet | 3.44 | 14.90 | 21.95 | 30.31 | 35.79 | 39.86 | 43.11 |
| | | Sum | 2.88 | 12.26 | 18.90 | 26.09 | 30.44 | 33.61 | 36.18 |
| | | Gen | 3.83 | 15.25 | 23.59 | 32.52 | 37.72 | 41.38 | 44.25 |
| | | MT | 4.92 | 19.16 | 28.02 | 36.65 | 41.65 | 45.33 | 48.16 |

Table 4: $P@k$ scores on selected code models, focusing API calls and API imports.

| | | | | $P@1\uparrow$ | $P@5\uparrow$ | $P@10\uparrow$ | $P@20\uparrow$ | $P@30\uparrow$ | $P@40\uparrow$ | $P@50\uparrow$ |
|---|---|---|---|---|---|---|---|---|---|---|
| API Call With Alias | CodeBERT | MLM | Alias | 15.83 | 32.54 | 40.40 | 48.46 | 53.50 | 56.93 | 59.88 |
| | | | Adv. Alias | 14.10 | 29.84 | 37.66 | 45.89 | 50.86 | 54.37 | 57.16 |
| | | MLM-Python | Alias | **24.17** | **42.30** | **50.19** | **58.02** | **62.68** | **66.05** | **68.37** |
| | | | Adv. Alias | 20.35 | 37.30 | 45.33 | 53.42 | 58.22 | 61.72 | 64.37 |
| | GraphCodeBERT | MLM | Alias | 18.91 | 36.85 | 43.99 | 51.66 | 56.18 | 59.58 | 62.17 |
| | | | Adv. Alias | 16.89 | 33.63 | 40.88 | 48.48 | 53.12 | 56.44 | 59.09 |
| | PLBART | Base | Alias | 0.84 | 4.20 | 8.05 | 14.56 | 19.34 | 23.46 | 26.84 |
| | | | Adv. Alias | 0.38 | 2.62 | 5.60 | 10.81 | 14.99 | 18.45 | 21.47 |
| | | Large | Alias | 3.36 | 11.64 | 18.34 | 26.15 | 30.88 | 34.71 | 37.58 |
| | | | Adv. Alias | 1.49 | 7.19 | 12.70 | 19.63 | 23.92 | 27.08 | 29.69 |
| | | CSNet | Alias | 1.75 | 7.86 | 12.37 | 18.59 | 22.93 | 26.87 | 30.56 |
| | | | Adv. Alias | 1.68 | 6.63 | 10.39 | 15.48 | 19.30 | 22.45 | 25.21 |
| | | Sum | Alias | 1.21 | 4.93 | 8.44 | 14.20 | 18.14 | 21.50 | 24.21 |
| | | | Adv. Alias | 1.02 | 3.75 | 6.36 | 10.43 | 13.74 | 16.55 | 18.95 |
| | | Gen | Alias | 3.43 | 12.87 | 19.29 | 26.85 | 31.66 | 35.05 | 37.61 |
| | | | Adv. Alias | 2.13 | 8.54 | 13.55 | 19.79 | 23.89 | 27.08 | 29.67 |
| | | MT | Alias | 3.56 | 11.68 | 18.61 | 26.42 | 31.35 | 34.80 | 37.82 |
| | | | Adv. Alias | 2.39 | 8.08 | 12.98 | 19.72 | 24.47 | 28.03 | 30.94 |

Table 5: Comparison of $P@k$ scores on API import alias quizzes among selected code models.

CodeBERT-MLM-Python and GraphCodeBERT-MLM demonstrate superior performance on API calls and imports compared to other models. However, their overall precision of 30% measured by $P@1$ falls short of perfection, indicating a lack of knowledge about API names. While we expected GPT-3.5-turbo to have a better understanding of API names, it shows that the model performs slightly worse than CodeBERT-MLM.

Additionally, our comparison shows that PLBART variants perform much worse on understanding Python API name knowledge than BERT-like models, which can be explained by the pre-training objectives of PLBART. PLBART-Large consistently outperforms other variants, indicating that model size may be an important factor in the amount of stored API name knowledge. However, this finding should be interpreted in light of the scaling law of mixed-modal language models (Aghajanyan et al., 2023), which suggests that larger models are likely to achieve better performance on downstream tasks, such as code generation. Finally, we find that pre-trained data can influence the understanding of API names to some extent, as shown by the performance gap between PLBART-Base and PLBART-CSNet. Our results indicate that fine-tuning on code generation tasks can improve the performance of pre-trained models, while text generation tasks may negatively impact them.

**Takeaways:** Although CodeBERT-MLM-Python and GraphCodeBERT-MLM show superior performance on API call and import name knowledge among code models, there is a significant margin for the improvement.

## 5.2 RQ2: Do code models understand API import aliases?

In order to assess the code models on the knowledge of API import aliases, we pair the 17,971 API import alias quizzes with the adversarial examples designed to test the models' robustness. To construct the adversarial set, we randomly selected 10 distinct aliases that are used in other modules and replaced the original aliases in the quizzes with these new aliases. For example, "`import numpy as np \n np.load_(`" will be transformed to "`import numpy as pmd \n pmd.load_(`" via the replacement of "np". To the end, we collect 179,710 adversarial quizzes.

We report $P@K$ results of ten models in Table 5 and illustrate examples in Appendix B. Based on the comparison, CodeBERT-MLM-Python consistently outperforms GraphCodeBERT-MLM, achieving higher $P@1$ scores of up to 24.17% for Alias scenarios compared to 18.91%. CodeBERT variants also show better overall performance with $P@50$ scores ranging from 59.88% to 68.37%, while GraphCodeBERT-MLM ranges from 59.09% to 62.17%. Our initial finding suggests that code models have a weaker understanding of API aliases compared to API calls and imports, as shown in Table 4. This indicates that current code models encode little knowledge of import aliases. Based on the performance of the GPT-3.5-turbo model on 1,000 randomly sampled quizzes, we can infer that it has a greater capability to understand API import aliases. When comparing the results of the original API import alias quizzes with those of the adversarial aliases, we found only minor discrepancies, indicating that these code models have strong robustness in understanding API import aliases. We further analyze the distribution of the API import aliases and find that an API is paired with 1.16 aliases on average, and 8% of APIs have more than 1 alias. We hypothesize that these code models are able to learn the compositional patterns of these APIs via different aliases, and thus manage to generalize to adversarial import aliases.

**Takeaways:** Although code models show robustness in understanding API import aliases, their encoding of API knowledge is limited to partial information.

### 5.3 RQ3: How well can code models memorize and generalize on API names?

| | | | $P@1\uparrow$ | $P@5\uparrow$ | $P@10\uparrow$ | $P@20\uparrow$ | $P@30\uparrow$ | $P@40\uparrow$ | $P@50\uparrow$ |
|---|---|---|---|---|---|---|---|---|---|
| API Call | CodeBERT | Seen | 23.96 | 42.42 | 50.38 | 58.26 | 62.84 | 65.96 | 68.39 |
| | | Unseen | 21.54 | 39.42 | 46.62 | 54.92 | 59.60 | 62.63 | 64.35 |
| | GraphCodeBERT | Seen | **26.00** | **43.45** | **50.91** | **58.45** | **62.89** | **66.02** | **68.42** |
| | | Unseen | 21.86 | 37.42 | 44.44 | 51.89 | 56.97 | 60.26 | 62.36 |
| | PLBART | Seen | 2.83 | 11.24 | 16.09 | 22.08 | 26.20 | 29.45 | 32.10 |
| | | Unseen | 2.00 | 8.31 | 12.80 | 18.38 | 21.94 | 24.52 | 27.32 |
| API Import | CodeBERT | Seen | 26.49 | 44.01 | 51.34 | 58.80 | 63.06 | 66.03 | 68.35 |
| | | Unseen | 20.46 | 38.83 | 45.82 | 52.58 | 56.49 | 58.94 | 61.08 |
| | GraphCodeBERT | Seen | **30.99** | **48.51** | **55.61** | **62.74** | **66.92** | **69.79** | **71.96** |
| | | Unseen | 22.37 | 38.41 | 45.81 | 53.69 | 57.83 | 60.78 | 63.16 |
| | PLBART | Seen | 3.64 | 15.29 | 22.39 | 30.80 | 36.29 | 40.37 | 43.63 |
| | | Unseen | 2.17 | 9.96 | 16.42 | 23.97 | 29.08 | 32.59 | 35.65 |

Table 6: Comparison of $P@k$ scores on **seen** and **unseen** API name quizzes among selected code models.

We evaluate whether code models demonstrate a deeper understanding of the names of **seen** APIs during pre-training than the **unseen** ones by conducting experiments on CodeBERT-MLM, GraphCodeBERT-MLM, and PLBART-CSNet, which were pre-trained on the *train set* of CSNet. To create our `PyINK-Mem` version, we take all APIs appearing during training (*train set*) as the **seen** split and the remaining (*test set*) APIs as the **unseen** split. We filter out all the APIs belonging to the **seen** libraries in the **unseen** split. The `PyINK-Mem` **seen** split contained 281,945 quizzes for API calls and 281,945 quizzes for API imports. The unseen split had 7,640 API call quizzes and 7,640 API import quizzes. We note that the selected models do not memorize any structures of the open-source repositories, due to the function-level pre-training objective.

In Table 6, we measure the model performance via $P@k$ up to $P@50$. Our inspection of the results on API call pop quizzes suggests there are slight differences between **seen** and **unseen** sets, indicating the strong generalization ability of these code models to new APIs. Among the three models we evaluated, CodeBERT-MLM demonstrates the most robust performance, while GraphCodeBERT-MLM demonstrates a greater ability to memorize API names during pre-training. Surprisingly, we found that there were 1,288 and 5,468 distinct ground-truth tokens in the **seen** and **unseen** splits for API calls, respectively, and 1,257 tokens (97.59% of the **unseen** split) were overlapped. This indicates that the API namespace designs share unexpected commonalities.

**Takeaways:** Code models demonstrate impressive generalization abilities in predicting the names of programming functions for new domains and reasonable memorizations of APIs from the training data.

## 6 Conclusion

In this paper, we have explored the interpretability of code models for source code (CodeBERT, GraphCodeBERT and PLBART). We conduct a thorough API name knowledge analysis based on a large-scale benchmark, `PyINK`, from the following four aspects, aiming to give an interpretation of code models. Firstly, we determine the API name knowledge stored by code models from two perspectives, API call and API import. Secondly, we investigate whether code models can robustly understand API import aliases. Thirdly, we revisit the settings in deep API learning and assess if providing additional natural language context can help code model retrieve more precise API name knowledge. Fourthly, we examine the memorization and generalization of code models on API names. The analysis in this paper has revealed several interesting findings that can inspire future studies on code representation learning and interpretation of knowledge encoded by code models.

## Limitations

The possible limitations lie in the choice of evaluation data. While we use a widely-used corpus, CSNet, which covers a substantial number of Python APIs, it is important to acknowledge that there are additional resources, such as Stack Overflow[2], that may contain more Python APIs. Moreover, CSNet was proposed in 2019, and new APIs may have been developed since then. We contend that our evaluation of PyINK using CSNet is statistically significant, but we also acknowledge the limitations of this corpus. Furthermore, code models may exhibit different behaviors when evaluated with APIs in other programming languages, such as Java and C. To address this threat to validity, we can enhance the completeness of our evaluation by incorporating more programming languages on which these code models are trained. By evaluating the code models in a broader range of programming languages, we can better ensure their robustness and generalizability to real-world programming tasks.

## References

Armen Aghajanyan, Lili Yu, Alexis Conneau, Wei-Ning Hsu, Karen Hambardzumyan, Susan Zhang, Stephen Roller, Naman Goyal, Omer Levy, and Luke Zettlemoyer. 2023. Scaling laws for generative mixed-modal language models. *arXiv preprint arXiv:2301.03728*.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 631–642.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Mohammad Abdul Hadi, Imam Nur Bani Yusuf, Ferdian Thung, Kien Gia Luong, Jiang Lingxiao, Fatemeh H Fard, and David Lo. 2022. On the effectiveness of pretrained models for api learning. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 309–320.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 293–304.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *arXiv preprint arXiv:2211.11501*.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

James Martin and Jin LC Guo. 2022. Deep api learning revisited. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 321–330.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. Deep learning meets software engineering: A survey on pre-trained models of source code.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

[2]https://stackoverflow.com/

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language models as knowledge bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2463–2473.

Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897.

Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359. IEEE.

Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic recommendation of api methods from feature requests. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 290–300. IEEE.

Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring execution environments of jupyter notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1622–1633. IEEE.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45.

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Trans. Softw. Eng. Methodol.*, 31(2).

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

Li Zhong and Zilong Wang. 2023. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*.

## A Experiment Details

We conduct extensive studies on ten selected models from the variants of CodeBERT, GraphCodeBERT and PLBART on A100 GPUs. We also include GPT-3.5-turbo for the assessment, though it is not specifically trained with mask prediction. We instruct GPT-3.5-turbo to predict top-20 masked tokens for each test of 1,000 randomly selected samples on API calls and imports, respectively. To guide GPT-3.5-turbo to correctly perform the prediction task, we prompt it with the instruction of "Predict top-20 answers of `<mask>` part of the following Python API fully qualified name, pay attention to the connected characters right after `<mask>`. Note the number of masked characters is at least one. Print each of 20 answers per line with index ".

## B Examples of RQ2

We present examples in Table 7.

## C Furture Study on The Effect of Natural Language Context on Knowledge Probing

We further examine the impact of natural language (NL) context on a code model's ability to comprehend API names. To build a dataset of API-related NL context, we utilize the NL queries designed for Python API sequence usages in the work of (Martin and Guo, 2022). For example, `<os.path.isfile>` is paired with "file directory check". We selected queries containing API sequences containing the `PyINK` API and transformed them into API pop quizzes. As some of the NL queries are long and infeasible for code models to process, we only choose the 10 shortest ones among these satisfied NL queries, and filter out the cases where the length is no more than 512 tokens determined by the CodeBERT tokenizer. At the end, we collected 56,645 pop quizzes for API calls, and 56,644 for API imports. We compute the average $P@k$ on each API name pop quiz with the group of NL queries concatenated ahead. We compare them with the overall results without adding NL context at the beginning in Figure 4.

Our results show that incorporating natural language queries led to a 2% improvement in probing API call name knowledge, although we contend that this may be due to their non-API-focused design, as they were initially created for API sequences rather than individual API calls (Martin and Guo, 2022). In addition, the relative performances among code models do not change before and after adding NL context, suggesting `PyINK` is robust for API name evaluation. We anticipate that incorporating API-focused natural language context will yield more substantial gains.

| | Test | Answer | CodeBERT-MLM-Python | GraphCodeBERT-MLM |
|---|---|---|---|---|
| API Alias | import weka.flow.conversion as conversion conversion.<mask>Through( | Pass | pass [0.50], Pass [0.17], run [0.05], Go [0.02], Run [0.01] | pass [0.66], Pass [0.15], cycle [0.08], write [0.04], walk [0.01] |
| | import memote.support.helpers as helpers helpers.<mask>_converting_reactions( | find | test [0.29], get [0.10], load [0.08], check [0.07], create [0.05] | start [0.27], allow [0.14], stop [0.08], try [0.04], begin [0.04] |
| | import pandas as pd pd.HDF<mask>( | Store | Store [0.35], 5 [0.30], S [0.11], Parser [0.04], 4 [0.04] | 5 [0.34], 4 [0.15], 3 [0.07], 0 [0.06], s [0.05] |
| | import numpy as np np.<mask>ccos( | ar | ar [0.39], g [0.05], n [0.04], cal [0.04], b [0.03] | ar [0.94], eu [0.03], ar [0.02], g [0.00], e [0.00] |
| API Adv. Alias | import weka.flow.conversion as ip ip.<mask>Through( | Pass | pass [0.51], run [0.08], Pass [0.06], connect [0.02], import [0.02] | pass [0.69], Pass [0.20], cycle [0.06], write [0.01], run [0.01] |
| | import memote.support.helpers as hermite hermite.<mask>_converting_reactions( | find | test [0.27], get [0.13], check [0.09], load [0.05], create [0.04] | start [0.41], stop [0.15], try [0.04], begin [0.03], allow [0.03] |
| | import pandas as IT IT.HDF<mask>( | Store | 5 [0.61], S [0.13], Store [0.11], 4 [0.03], Frame [0.01] | 5 [0.27], 3 [0.12], s [0.11], 0 [0.10], 4 [0.07] |
| | import numpy as simplejson simplejson.<mask>ccos( | ar | py [0.09], simple [0.04], g [0.04], cal [0.03], fun [0.03] | ar [0.28], eu [0.21], g [0.09], e [0.08], ce [0.06] |

Table 7: Examples of prediction for CodeBERT-MLM-Python and GraphCodeBERT-MLM on API import aliases and their adversarial samples. The last two columns reports the top five tokens generated together with the associated probabilities (in square brackets).

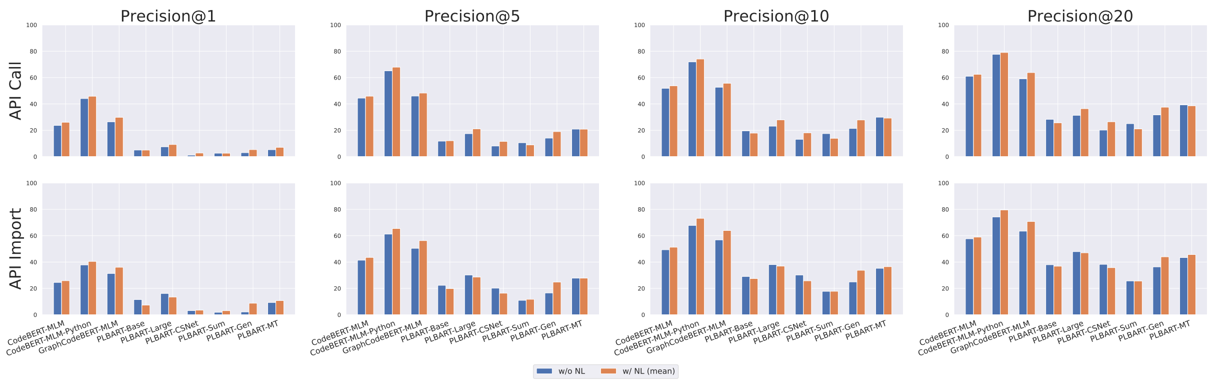

Figure 4: Comparisons of $P$@1, $P$@5, $P$@10 and $P$@20 scores of selected code models. **w/o NL**: No natural language context is provided along with pop quizzes. **w/ NL**: Mean performances when natural language context is provided along with pop quizzes.