Probing Pretrained Models of Source Code

Anonymous ACL submission

Abstract

Deep learning models are widely used for solving challenging code processing tasks, such as code generation or code summarization. Traditionally, a specific model architecture was carefully built to solve a particular code processing task. However, recently general pretrained models such as CodeBERT or CodeT5 have been shown to outperform task-specific models in many applications. While pretrained models are known to learn complex patterns from data, they may fail to understand some properties of source code. To test diverse aspects of code understanding, we introduce a set of diagnosing probing tasks. We show that pretrained models of code indeed contain information about code syntactic structure and correctness, the notion of namespaces, code readability and natural language naming, but lack understanding of code semantics. We also investigate how probing results are affected by using code-specific pretraining objectives, varying the model size, or finetuning.

1 Introduction

Deep learning and especially Natural Language Processing (NLP) methods have been widely and successfully adopted to process source code. Example tasks include code generation (Allamanis et al., 2015; Chen et al., 2021) where the task is usually formulated as to produce a code of a function given the natural description; code translation (Nguyen et al., 2013; Roziere et al., 2020) where the model needs to translate from one programming language to another; and code summarization (Haiduc et al., 2010; Alex et al., 2020) where the task is to produce natural language description (NL) for a given code snippet. Deep learning is also widely used in discriminative tasks, such as automated bug search and repair (Hellendoorn et al., 2020).

In recent years, the focus has shifted from developing task-specific models incorporating prior knowledge about the task, to relying on general pretrained models of code such as CodeBERT (Feng et al., 2020) or CodeT5 (Wang et al., 2021). These models, once pretrained, can be finetuned on the downstream tasks with a little additional cost, surpassing task-specific models. While the performance of the models is high on a wide range of downstream tasks (Lu et al., 2021), the boundary between what the models know and where they fail remains hidden behind the complexity of the downstream tasks. The lack of interpretability of pretrained models limits their practical use. At the same time, a deeper examination of model’s understanding of source code may increase developers’ trust and broaden the applicability of pretrained models.

In NLP, there is an established probing approach for a more fine-grained examination of the knowledge of various aspects of the language, e.g. morphology, syntax, or discourse understanding (Belinkov et al., 2020; Tenney et al., 2019; Koto et al., 2021). Probing usually means training a linear model on top of hidden representations of a model for various simple tasks, e.g. to predict a part-of-speech tag, to detect whether a sentence was corrupted, or to estimate the number of objects in the main clause (Conneau et al., 2018a). Probing experiments may suggest ways to improve the quality of the pretrained model or provide recommendations on how to tune the model better in applied tasks (Belinkov, 2021).

Inspired by the insights probing provided in NLP, we develop probing tasks to understand the extent to which the current state-of-the-art pretrained models capture structural and semantic properties of source code. Our contributions are as follows:

- we introduce a set of syntactic and semantic probing tasks, suitable for testing diverse aspects of code understanding;
- we study an effect of the model choice, pretraining objective choice, and model size on...
probing results;

- we use proibgs to highlight which information about code is preserved by finetuned models in different downstream tasks.

The rest or the work is organized as follows: Section 2 describes our probing tasks; Section 3 describes the pretrained models used for comparison; Section 4 describes the experimental setup of the work; experiments are presented in Section 5; Section 6 is devoted to related work, Section 7 concludes work and discusses limitations, and Section 7 discusses the broader impact.

## 2 Probing tasks

We probe pretrained models of code using linear regression or classification trained on top of code representations extracted from each model layer (layers weights are not finetuned) (Alain and Bengio, 2016). We develop auxiliary tasks (with synthetic data or data borrowed from other works) that test models’ understanding of various properties of source code: strict syntactic structure and correctness, the notions of data flow and namespaces, naming, the underlying algorithm, and readability. We consider both global tasks (predicting a property of the whole code snippet) and local tasks (predicting a property of a particular token or a group of tokens). For each task, we introduce a simple but as strong as possible baseline. Figure 1 illustrates all tasks.

**Notation.** Pretrained models of code usually follow the standard NLP methodology: representing a code snippet as a sequence of subtokens, e.g. byte-pair encoding subtokens, and pretraining the model on a large corpora of source code using masked language modeling. We denote the sequence of subtokens as $s_1, \ldots, s_m$. Let us denote $t(s_i)$ a mapping from a subtoken $s_i$ into a corresponding code token $t(s_i)$, e.g. for a subtoken sequence $[1, 2, 1], \text{let } d, \text{ we extract the model’s embedding } \mathbf{w}_i \in \mathbb{R}^d \text{ for a particular layer } \ell, \text{ where } d \text{ is the size of hidden representations}.

### 2.1 Token Path Type

The first three probing tasks test whether pretrained models contain information about the syntactic structure and correctness of code. The first task consists of predicting the position of a token in the Abstract Syntax Tree (AST). Given a subtoken $s_i$ and the corresponding embedding $\mathbf{w}_i$, the task is to predict the path type from the root to the $t(s_i)$ token, e.g. $[1, 2, 1]$, meaning go to the first child, then to the second one, then to the first one. This task is formulated as a classification problem over 15 most common path types, this is a local task. As a baseline, we consider constant prediction w. r. t. a subtoken, i.e. we select the most frequent class (path type) for each subtoken in the vocabulary.

### 2.2 AST depth

The second syntactic task is defined on a code snippet level (global task) and consists of predicting the depth of the AST built from the snippet (regression problem). The baseline for this task is
defined as a linear regression trained on a single feature – the number of tokens in the code snippet, this baseline outperforms the mean depth baseline computed over the whole dataset.

### 2.3 Is Bracket Misused

The third task evaluates the model’s understanding of syntactic correctness. In this task, we replace 30% of the brackets in code with wrong brackets and ask a probing model to identify if the particular bracket token is misused (binary classification, local task). The baseline is the most frequent label for each bracket type `{ } [ ] <>.

### 2.4 Edge Prediction in Data Flow Graph edge classification

The next task measures to what extent a model encodes the information about the data flow. Given two subtokens $s_i$ and $s_j$, the task is to predict a data flow edge type between $t(s_i)$ and $t(s_j)$. There can be no edge (negative example), a "comesFrom" edge, or a "compFrom" edge. The task is formulated as classification of a pair of subtokens (their embeddings are concatenated), this is the local task. An example in Figure 2 illustrates the task.

To obtain a balanced training set, in addition to existing data flow edges we select a roughly equal number of negative examples by selecting random pairs of nodes from AST with suitable node types (e.g. pairs of identifiers, constants, etc.). The constant baseline in this task predicts (the most frequent) "comesFrom" for a pair of the same tokens and "No Edge" for a pair of different tokens. We found that the performance of this baseline is similar to the performance of a constant baseline which selects the most frequent class for each pair of tokens.

### 2.5 Is Variable Declared

This task tests the model’s understanding of the notion of namespaces. The model is asked, whether there is an ‘undeclared variable name’ error for a particular expression with an identifier. For example, in the first code snippet the identifier $y$ is correctly used after a declaration:

```java
int x = 0;
if (x == 0) {
    int y = x;
    System.out.println(y);
}
```

However, in the second snippet there is an error, since $y$ is outside of the scope of it’s declaration:

```java
int x = 0;
if (x == 0) {
    int y = x;
}
System.out.println(y);
```

We generate positive and negative examples using the following procedure. For a code snippet, we find a variable name declaration, e.g. `float x = 0`. Next, we find a random place in code after the variable declaration where we can insert a printing expression e.g. `System.out.println(x)`, and define a label for binary classification analyzing variable scopes. The task is formulated for the subtokens of the token representing the inserted variable name (local task). The baseline in this task is a constant prediction that the variable is declared.

### 2.6 Variable Name

The next task targets the ability to link code elements and their natural language descriptions. A model should predict a variable name, given a code snippet with all occurrences of the original name replaced with a placeholder `var`. This task requires semantic understanding of the variable’s role in the program (local task).

We formulate this task as classification, targeting only 15 most popular identifier names. The feature vector is a mean hidden representation for all occurrences of the identifier in code. In such way, the model should be able to predict the identifier name based on the context in which the variable was used. The baseline in this task is defined by the bag-of-words model: we count occurrences of all subtokens in the code snippet, convert them to tf-idf values and train a linear classifier on these features. This baseline substantially outperforms...
the constant baseline which always predicts the most frequent variable name.

### 2.7 Sorting Type

The next (global) task also tests the ability of models to understand code semantics, particularly, to distinguish different algorithms. We use the dataset (Nghi D. Q. et al., 2019) consisting of different implementations of 10 sorting algorithms downloaded from Github and manually labelled. The dataset size is 1000 examples. As a baseline, we again use the bag-of-words model described above, which performs much better than the most frequent sorting type baseline.

### 2.8 Readability

Finally, we consider a readability property of code. Readability defines how easy code is for the programmers to understand and maintain. Generally, readability depends on visual appearance of code (spaces, new lines etc), the meaningfulness of variable and function names, the quality of comments, and the particular algorithmic implementation (the same algorithm could be written in different ways, some of them more and some of them less readable). We use the 200 examples dataset provided by Scalabrino et al. (2018) and obtained by collecting a set of functions and asking developers to rate readability on the scale from 1 to 5 (several ratings per example). The task is then converted by the authors to binary classification by treating all snippets with rating \(\leq 3.6\) as not readable and the rest ones as readable. This is a global task and as the baseline we use the bag-of-words model which outperforms the constant most frequent class prediction.

### 3 Models

In this section, we briefly describe the models to be compared. We have selected several widely used pretrained models, which vary in the model architecture, pretraining objective, model size, and training datasets.

#### 3.1 CodeBERT

CodeBERT (Feng et al., 2020) is one of the first attempts to pretrain a Transformer-based encoder model for source code representation learning and comprehension. It is a 12 layer encoder model based on RoBERTa-base (125M) (Liu et al., 2019) and trained with masked language modeling and replaced token detection objectives. The model is trained on 6M CodeSearchNet dataset (Husain et al., 2020), composed of functions from 6 programming languages (Java, Python, JavaScript, PHP, Ruby, Go) and NL comments.

#### 3.2 GraphCodeBERT

GraphCodeBERT (Guo et al., 2021) extends the work of (Feng et al., 2020), by introducing data flow-related objectives.

#### 3.3 PLBART

Ahmad et al. (2021a) introduced a 140M parameter PLBART model with 6 encoder and 6 decoder layers. The model is based on the BART (Lewis et al., 2020) architecture. The authors released a PLBART \(^1\) checkpoint pretrained on the 470M Java, 210M Python functions/methods and on 47M NL descriptions, as well as a PLBART_large checkpoint (400M, 12 layer encoder, 12 layer decoder).

#### 3.4 CodeT5

CodeT5 (Wang et al., 2021) is an encoder-decoder model based on the T5 (Raffel et al., 2020) architecture and pretrained on 8.35M functions in 8 programming languages (Python, Java, JavaScript, PHP, Ruby, Go, C, and C#). The model combines the masked language modeling objective with code-specific objectives, including identifier tagging and predicting variable names. We experiment with two released model checkpoints \(^2\): CodeT5-base (220M) and CodeT5-small (60M).

#### 3.5 CodeGPT2

CodeGPT2 (Lu et al., 2021) is a decoder only model based on the GPT-2 architecture (Radford et al., 2019). The 117M model consists of 12 layers and is pretrained on the CodeSearchNet (Husain et al., 2020) dataset. We used CodeGPT-small-java-adaptedGPT2 checkpoint \(^3\), that is initialized from GPT-2 model and then trained on code corpus.

#### 3.6 BERT

We also consider the text-based model, BERT, to understand the effect of code-specific pretraining. We use a 110M 12-layer BERT model (Devlin et al., 2019), “bert-base-cased” checkpoint from Hugging Face \(^4\), trained on the Book Corpus (Zhu et al., 2015).

---

1. https://github.com/wasiahmad/PLBART
4 Experimental setup

4.1 Data and preprocessing

For our experiments which involve synthetic data (first 6 tasks), we use the test dataset provided by Ahmad et al. (2021a) consisting of 10k examples of Java functions and methods. Preprocessing removes comments and new line symbols, and standardizes code snippets. For each pretrained model, we apply its subtokenization procedure. All models have a limit of 512 input subtokens. We crop subtoken sequences that are longer than 512 subtokens. We use commonly used open access datasets intended for research purposes.

4.2 Probing details

For each probing task, we average results over four runs using 4-fold cross-validation. For each model, we use a single checkpoint as usually only one checkpoint is released.

Each pretrained model returns representations for a sequence of subtokens \(s_1, \ldots, s_m\), e.g. from byte-pair encoding. When the task is formulated on a code snippet level, the layer-wise embeddings of the snippet are obtained by averaging subtoken embeddings, following (Koto et al., 2021).

For the probing models, we use linear models from scikit-learn (0.24.2) (Pedregosa et al., 2011), including \texttt{SGDClassifier} with logistic regression loss for classification tasks (we select optimal alpha parameter via grid search over \([0.0001, 0.001, 0.01, 0.1, 1, 10, 100]\) range, and set tolerance to 0.0001); and \texttt{RidgeCV} for regression tasks (grid search for alpha over \([0.0001, 0.001, 0.01, 0.1, 1, 10, 100]\) range).

5 Experiments

Our research questions are as follows:

- To what extent do the models pretrained on code capture information about source code properties?
- Does multitask pretraining with code-specific objectives provide richer representations?
- How does the model size affect probing results? Which representations are better: provided by the encoder or by the decoder? Which layers provide better representations?
- Does finetuning preserve syntactic and semantic information in different downstream tasks?

We used a single Tesla V100 GPU for the forward pass to collect embeddings, and 4 CPU for each training pipeline of the linear models. Our total computational budget is 864 CPU hours and 20 GPU hours.

5.1 Comparison of different models

In this subsection, we study the performance of different pretrained models in all probing tasks. In this experiment, we report the results for the best performing layer representation for each model: the layer is chosen using the first fold and the results are averaged over three remaining folds. Figure 3 presents the results.

Overall, we observe that probing performance of pretrained models varies among tasks. In syntax-related (three first tasks), namespace-related (”Is Variable Declared”), readability-related and variable naming tasks, models pretrained on code substantially outperform baselines and hence contain knowledge about the mentioned code properties to some extent. However, in ”Edge Prediction” and ”Sorting type” tasks, probed models perform similarly to simple baselines. Both tasks require the deeper semantic understanding of code, in contrast
to most previously listed tasks (except, maybe, variable naming). In sorting type prediction, the model needs to distinguish algorithms, and in edge prediction, the model needs to distinguish contexts in which variables are used. For example, in Figure 2, variable b (number 8) has the edge "ComesFrom" with b (number 2) where it was introduced but does not have the edge with b (number 5) where it was used in condition. Most pretrained models perform similarly to the baseline which predicts the edge based on the coincidence of two identifiers. To sum up, models pretrained on code contain knowledge about source code specific properties but lack understanding of code semantics.

Comparing different models, we find that there is no single model that excels for all the tasks. In "Node path type", "AST depth", "Variable Name", "Sorting type" and "Readability" tasks, all models pretrained on code perform similarly. In "Bracket is misused", CodeBERT unexpectedly outperforms other models because its main pretraining objective is masked language modeling, while other models additionally incorporate other objectives, with CodeT5 having 5 pretraining tasks and performing worse than other models. CodeGPT performs even worse because it only sees the left context which may be not enough to predict the misused bracket. In "Edge Prediction", GraphCodeBERT performs best because it used the edge prediction objective during pretraining. Similarly in "Variable is Undeclared", CodeT5 performs best potentially because it used the variable-related pretraining objective. To sum up, models pretrained with code-specific objectives, CodeT5 and GraphCodeBERT, do not show consistent gain over other models, pretrained with single objectives. However, they do perform well in tasks, related to their pretraining objectives.

The BERT model pretrained on textual data performs worse than models pretrained on code in all tasks except the "Readability" task.

### 5.2 Encoder vs Decoder

This subsection compares the representations of the encoder and the decoder. We consider representations of two encoder-decoder models, PLBART-base and CodeT5-base. Table 1 compares best performing encoder representations and best performing decoder representations for all probing tasks.

We observe that in almost all probing tasks, the decoder representations perform worse or on par with the encoder representations. In some tasks, e.g. "Bracket Is Misused", the decoder shows much worse results than the encoder. A possible explanation is that the aim of the encoder is to provide rich representations for the decoder, hence the encoder is more suitable for information extraction.

### 5.3 The effect of the model size

In this subsection, we are interested whether larger models capture more information about the source code properties than smaller models. Table 1 reports the performance of CodeT5-base and CodeT5-small models, and of PLBART-large and PLBART-base models (other models are not available in variable sizes). We find that in three tasks the larger models expectantly perform better than smaller models but in the majority of the tasks the performance is similar.

### 5.4 Per-layer probing performance

We now analyse probing results for different Transformer layers. Figure 4 shows the per-layer performance of all considered pretrained models.

In syntax-related, namespace-related, data flow-related, and readability-related tasks, middle layers
In this section, we study the effect of finetuning on probing results. Specifically, we are interested in 1) whether finetuned models preserve information contained in pretrained models; 2) does pretraining enrich the representations of finetuned models, compared with the representations of models trained from scratch.

In this section, we focus on the PLBART model and finetune it for 5 downstream tasks: 3 generative tasks (Code Translation from Python to Java, Java Code Generation based on natural language descriptions, Java Code Summarization into textual description) and 2 discriminative tasks (Clone Detection, Defect Prediction). We use the AVATAR dataset (Ahmad et al., 2021b) in the Code Translation task and CodeXGLEU benchmark (Lu et al., 2021) in other tasks (MIT license). We use scripts for PLBART finetuning on these tasks provided in PLBART 5 and AVATAR 6 repositories.

Figure 5 compares 3 scenarios: the PLBART checkpoint after pretraining (leftmost bar), checkpoints after PLBART finetuning on each of 5 downstream tasks (dark bars), and checkpoints after training from scratch on each of 5 downstream tasks (semi-transparent bars). We also include baselines for reference.

Models finetuned for discriminative tasks exhibit a highest information loss between the initial pretrained stage and the finetuned stage, which may indicate that models trained on these tasks rely on some spurious features, rather than on code syntax or semantics.

Among generative tasks, the Code Translation model exhibits almost no gap between pretrained and finetuned stages. This could be attributed to having code as both input and output of the task. Code Generation and Code Summarization models have code only as either the input or the output of the task, and usually exhibit a slightly larger gap.

As for models, trained from scratch for downstream tasks (semi-transparent bars), the overall trend is similar across the downstream tasks, but the absolute results are usually much worse, compared to finetuned models, and sometimes are close to simple baselines. The downstream tasks alone do not provide high-quality code representations.

---

5 https://github.com/wasiahmad/PLBART
6 https://github.com/wasiahmad/AVATAR
6 Related Work

Probing became a universal tool in NLP for testing pretrained models’ understanding or knowledge of various language aspects. Pioneer works of Köhn (2015) and Gupta et al. (2015) used probing to study the morphological, syntactic, and semantic properties of static word embeddings, while the later works of Ettinger et al. (2016) and Conneau et al. (2018b) used probing to study sentence embeddings. Recent works propose more complex probing frameworks, e.g. Hewitt and Manning (2019) utilize metric learning approach to restore parse trees from embeddings, or consider new language aspects, e.g. discourse understanding (Koto et al., 2021). We refer to Belinkov (2021) for a broad review of existing probing works in NLP.

In the context of source code, Karmakar and Robbes (2021) make first steps towards probing pretrained models. However, they only consider four simple tasks and two code models, CodeBERT and GraphCodeBERT. In contrast to their work, we propose a wider set of more complex tasks, including several token-wise tasks, consider a wider range of pretrained models, and investigate various dimensions, including different pretraining objectives, model sizes, and the effect of finetuning.

7 Conclusion and discussion

We presented a diagnosis tool, based on probing tasks, that can be used to estimate to which extent deep learning models capture the information about various properties of source code in their hidden representations. Our results show that pretrained models of code do contain information about code syntactic structure and correctness, the notion of namespaces, code readability and natural language-based naming. However, pretrained models lack understanding of code semantics, which means that their usefulness in applied tasks requiring semantic understanding of code may be limited.

Using code-specific pretraining objectives enriches the understanding of the code aspects addressed in the corresponding objective. However, this multitask pretraining does not provide generally richer representations. This result may suggest practitioners to choose pretrained models which pretraining objectives are better aligned with the considered applied task. Future works may consider developing more diverse code-specific pretraining objectives.

We also found that finetuning may deteriorate the model’s understanding of code properties, especially in classification downstream tasks. This may suggest including code-specific objectives in finetuning, especially if multi-stage finetuning (Pruksachatkun et al., 2020) is used.

The limitations of our probing toolkit is that it does not cover all possible aspects of source code and only considers simple linear probing. However, we argue that our toolkit covers main distinguished properties of code. As for linear probing, it was successfully used in a lot of NLP probing approaches Belinkov (2021) and is well suitable for particular research questions considered in the paper.

Broader Impact

The main goal of this paper is to provide an empirical study of the existent models. Since we do not propose new models, there are no potential social
risks to the best of our knowledge. Our work may benefit the research community providing more introspection to the current state-of-the-art models.

References


