Probing Pretrained Models of Source Codes

Sergey Troshin, Nadezhda Chirkova*
HSE University
{stroshin, nchirkova}@hse.ru

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 diagnostic probing tasks. We show that pretrained models of code indeed contain information about code syntactic structure, the notions of identifiers, and namespaces, but they may fail to recognize more complex code properties such as semantic equivalence. 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., 2020a) 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 (NL) description 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, 2022).

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, pre-

*Now at Naver Labs Europe

Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, pages 371 - 383
December 8, 2022 ©2022 Association for Computational Linguistics

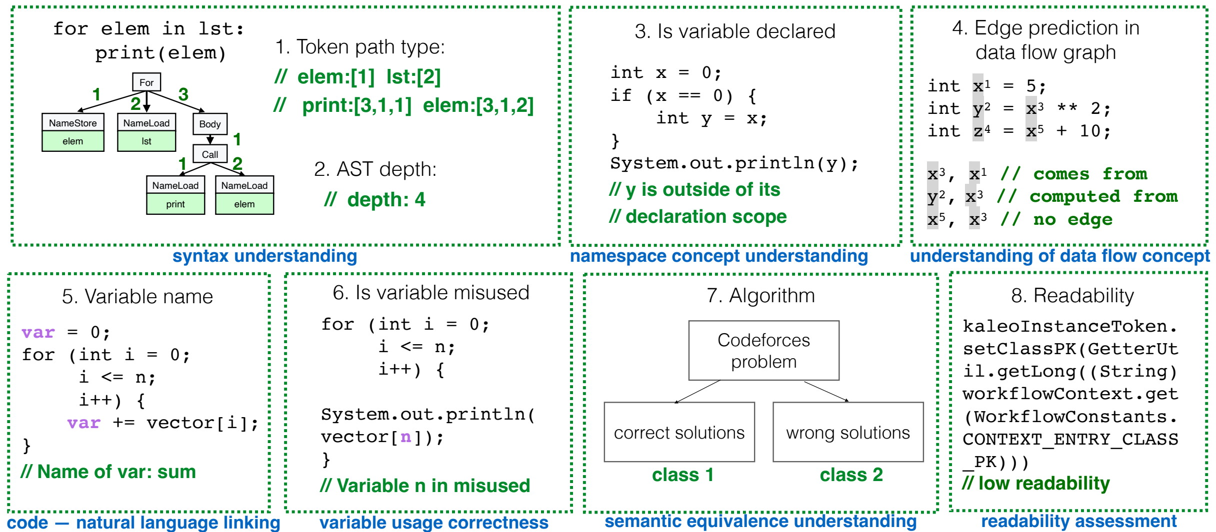

Figure 1: Illustration of the proposed probing tasks testing various aspects of code understanding.

training objective choice, and model size on probing results;

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

We release our code[1].

2 Probing tasks

We probe pretrained models of code using linear regression or classification trained on top of code representations extracted from each layer of each model (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, the notions of data flow and namespaces, naming, semantic equivalence, 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. For all classification tasks, we measure test error $(1 - accuracy)$, and for "AST depth" regression task, we calculate the mean absolute error $MAE(y_{true}, y_{pred}) = \frac{1}{N}\sum_{i=1}^{N}|y^i_{true} - y^i_{pred}|$.

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 [_(, _for, _public, _get, Status], $t(_get) =$ getStatus. For each subtoken s_i, we extract the model's embedding $\mathbf{w}_i^\ell \in R^d$ for a particular layer ℓ, where d is the size of hidden representations.

2.1 Token Path Type

The first two probing tasks test whether pretrained models contain information about the syntactic structure 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 $\mathbf{w}_i^\ell$, 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, which is a local task, is formulated as a classification problem by selecting target subtokens corresponding to 15 most frequent path types. 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

[1] https://github.com/serjtroshin/probings4code

linear regression trained on a single feature – the number of tokens in the code snippet, this baseline outperforms the median depth baseline computed over the whole dataset.

2.3 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:

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

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

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

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: is variable declared before used? The task is formulated for the mean subtoken embedding of the inserted variable name (local task). The baseline in this task is a constant prediction that the variable is declared.

2.4 Edge Prediction in Data Flow Graph

The next task measures to what extent a model encodes the information about the data flow. Given two tokens, the task is to predict a data flow edge between them. There can be no edge (negative example), a "comes from" edge, or a "computed from" edge. The task is formulated as classification of a pair of tokens (their mean embeddings over subtokens are concatenated), this is the local task.

In addition to existent data flow edges we select a roughly equal number of "no edge" examples by selecting random pairs of nodes from AST with suitable node types (e.g. pairs of identifiers, constants, etc.). As a baseline, we predict the most frequent edge type for the corresponding pair of tokens, which outperforms the most frequent class baseline.

2.5 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.

2.6 Is Variable Misused

The next local task tests the ability of the model to detect the variable misuse bug (Hellendoorn et al., 2020). We introduce variable misuse by randomly assigning "wrong" identifier name copying from another identifier from the same code snippet. We add "correct" code snippet for each "wrong" snippet, formulating the task as a binary classification problem, where the input is identifier's subtokens (mean embedding). The baseline is the bag of words predictor, which, for this task, is better than most frequent class predictor.

2.7 Algorithm

The next (global) semantic task also tests the ability of models to distinguish computationally equivalent codes from other codes. To obtain a dataset for this task, we select a simple problem from the CodeForces competition[2], which can be reformulated as to check if each character in a string has a neighbor equal to it. We download "wrong answer" and "accepted" Python submissions from the contest and filter out too long codes (>1000 characters) obtaining 550 "accepted" and 384 "wrong answer" submissions.

[2] https://codeforces.com/problemset/problem/1671/A

The task is to distinguish “correct" code from “wrong" and formulated as a binary classification problem. The task requires deeper understanding of the data and control flow since the “accepted" and “wrong answer" solutions are usually very similar visually. It should be hard for a model to make predictions based only on spurious surface or syntactic features to succeed in this task. As a baseline, we again use the bag-of-words model described above.

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 $\leqslant 3.6$ as not readable and the rest ones as readable, as in Scalabrino et al. (2018). 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 CodeSeachNet 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. They encourage the models to learn structure-aware representations by predicting randomly selected “comes from" data-flow edges.

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[3] checkpoint pretrained on the data collected by Roziere et al. (2020b), which is 470M Java, 210M Python functions/methods, and pretrained the 47M NL descriptions. They release a PLBART_large checkpoint as well (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[4]: CodeT5-base (220M) and CodeT5-small (60M).

3.5 CodeGPT2

CodeGPT2 (Lu et al., 2021) is a decoder only model based on 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[5], 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.,

[3] https://github.com/wasiahmad/PLBART
[4] https://github.com/salesforce/CodeT5
[5] https://huggingface.co/microsoft/ /CodeGPT-small-java-adaptedGPT2

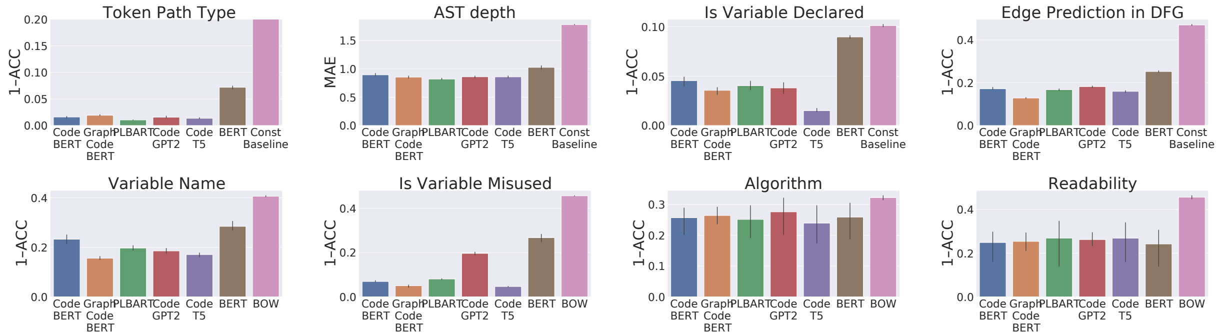

Figure 2: Results for the best performing layer representations, for all probing tasks. Metrics are the lower the better.

2019), *"bert-base-cased"* checkpoint from Hugging Face[6], trained on the Book Corpus (Zhu et al., 2015).

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 with removed comments and new line symbols, and standardized code snippets. For the two remaining tasks the datasets were mentioned in the task descriptions. For each pretrained model, we apply it's 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 *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 *RidgeCV* for regression tasks (grid search for alpha over $[0.0001, 0.001, 0.01, 0.1, 1, 10, 100]$ range). In addition to linear probings, we also probe pretrained models with a 3-layer MLP (see Appendix B), however, the results for MLP are similar.

[6] https://huggingface.co/bert-base-cased

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 training 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 2 presents the results.

Overall, we observe that the probing performance of pretrained models exceeds the performance of the simple baseline in all tasks. However, in the semantic-related “Readability" and “Algorithm" tasks, the pretrained models are very close to the simple baseline and thus do not capture much more information relevant to these tasks. The BERT model pretrained on textual data performs worse than the models pretrained on code in all tasks except the semantic-related “Readability" and “Algorithm" tasks, where all pretrained models perform similarily. We conclude that *models pretrained on code contain knowledge about basic source code properties but lack a deeper semantic understanding of code.*

Comparing different models, we find that the models pretrained with code-specific objectives, GraphCodeBert and CodeT5, are better or on par with other models for all tasks. In “Edge Prediction in DFG", GraphCodeBERT performs best because it uses the edge prediction objective during pretraining. Similarly in “Variable is Undeclared", “Is Variable Misused" CodeT5 and GraphCodeBERT perform best potentially because they use the variable-related pretraining objectives. CodeGPT2 performs worse for “Is Variable Misused" task, because it only sees the left context which may be not enough to predict the misused variable. To sum up, *models pretrained with code-specific objectives, CodeT5 and GraphCodeBERT, show consistent gain for the tasks related to their pretraining objectives, over other models, pretrained with single objectives, or perform on par with them.*

To better understand how pretrained models perform on each task, we perform an ablation study masking different code components: identifiers, keywords, and punctuation. This ablation study is described in Appendix A. The main finding is that masking punctuation hurts the probing performance of the model pretrained on source code in almost all tasks, while masking language keywords and renaming identifiers do not have much effect (except the variable naming task where renaming identifiers has a significant effect).

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. “Is Variable 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 variable related 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 3 shows the per-layer performance of all considered pretrained models. In syntax-related, namespace-related, data flow-related, algorithm-related and readability-related tasks, *middle layers (4–10) usually provide the most informative representations*. In the “Variable Name" (and partly in “Is Variable Misused"), the last layers consistently perform better because the task is closely related to the masked language modeling objective which is usually solved on top of last layers.

5.5 The effect of finetuning

In this section, we study the effect of finetuning on probing results. Specifically, we are interested 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

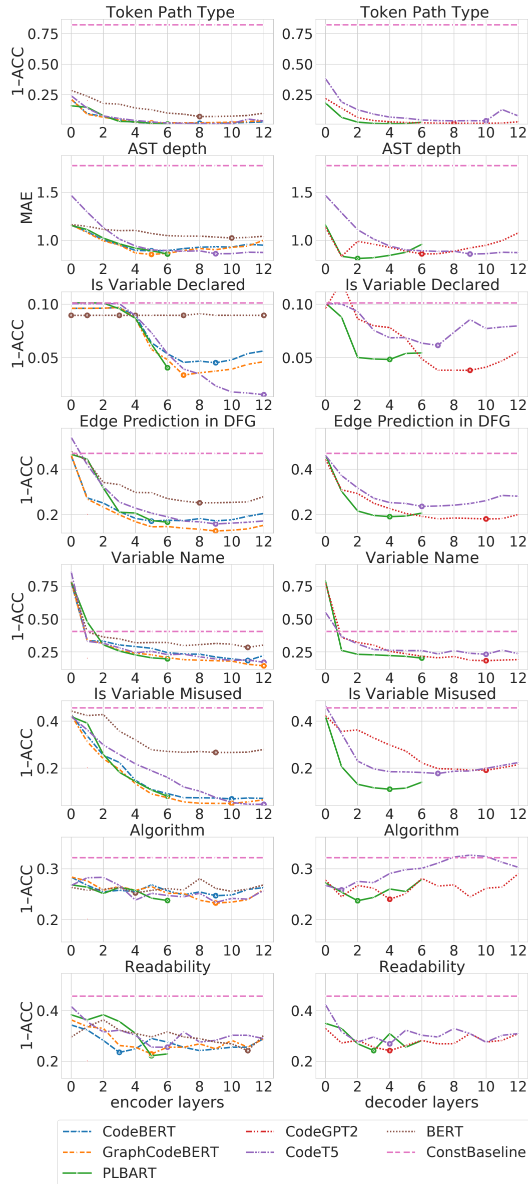

Figure 3: Per-layer probing performance of four pretrained models. Dots highlight the best layers for a particular model.

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[7] and AVATAR[8] repositories.

Figure 4 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.

[7]`https://github.com/wasiahmad/PLBART`

[8]`https://github.com/wasiahmad/AVATAR`

Models finetuned for discriminative tasks exhibit the 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.*

6 Related Work

Probing became a universal tool in NLP for testing pretrained models' understanding or knowledge of various language aspects. A simple linear probing was used in (Gupta et al., 2015) to test whether referential knowledge is already encoded in word embeddings, while Köhn (2015) got insights into the behaviour of word embedding in terms of morphological and syntactical properties. Probing tasks were developed to evaluate sentence embeddings (Ettinger et al., 2016; Conneau et al., 2018b) whether they incorporate compositional, or surface (length of the sentence), syntax (tree depth, top constituent), and text semantics (e.g. tense of a sentence) knowledge. Hewitt and Manning (2019) proposed more complex probing tasks, questioning the possibility to parse the whole dependency trees from the sentence embeddings using a metric learning approach. More recent studies for language models include the study of emerging capabilities of large language models (Wei et al., 2022). We refer to Belinkov (2022) for a broad review of existing probing works in NLP.

In the context of source code, Karmakar and Robbes (2021) made the first steps towards probing pretrained models. However, they only consider four simple tasks and tree code models, CodeBERT, CodeBERTa and GraphCodeBERT. In contrast to their work, we propose a wider set of tasks, including several token-wise tasks, consider a wider

	PLBART		CodeT5		PLBART		CodeT5	
Task	encoder	decoder	encoder	decoder	base	large	small	base
Token Path Type	**0.011**	0.012	**0.013**	0.036	**0.011**	0.014	**0.012**	0.013
AST depth	0.866	**0.820**	**0.867**	**0.864**	**0.820**	**0.850**	**0.863**	**0.864**
Is Variable Declared	**0.042**	**0.049**	**0.014**	0.061	0.042	**0.019**	0.025	**0.014**
Edge Prediction in DFG	**0.167**	0.191	**0.161**	0.230	**0.167**	**0.167**	**0.162**	**0.161**
Variable Name	**0.186**	**0.194**	**0.162**	0.211	**0.186**	**0.165**	0.208	**0.162**
Is Variable Misused	**0.080**	0.112	**0.046**	0.176	0.080	**0.053**	0.064	**0.046**
Algorithm	**0.239**	**0.251**	**0.228**	**0.268**	**0.246**	**0.225**	**0.235**	**0.228**
Readability	**0.226**	**0.237**	**0.247**	**0.246**	**0.216**	**0.238**	**0.242**	**0.221**

Table 1: Encoder vs decoder performance for PLBART-base and CodeT5-base; and comparison of small vs large models: PLBART-base vs PLBART-large, and CodeT5-small vs CodeT5-base. Metrics: MAE for "AST depth", otherwise test error (1-accuracy).

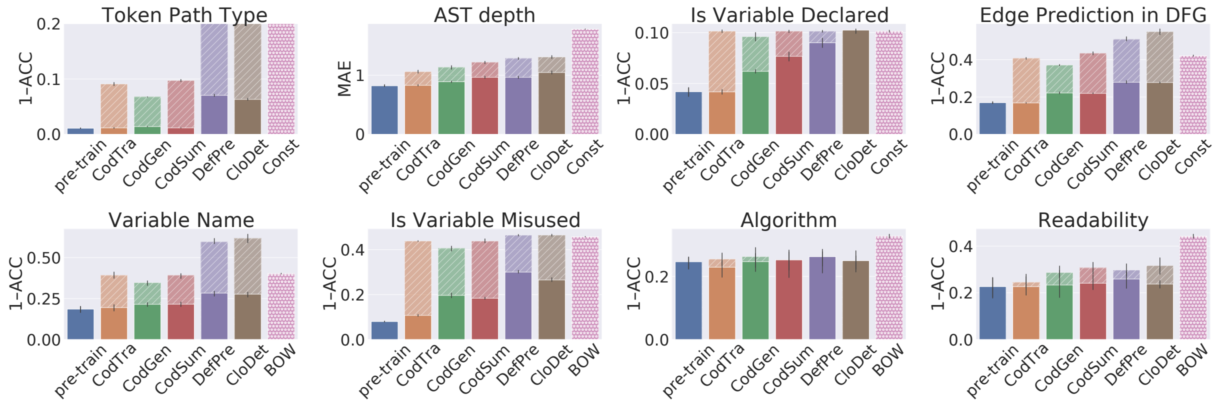

Figure 4: Results on the effect of finetuning. Pre-train (leftmost bar): pretrained-only checkpoint. The following bars: dark – finetuned models, semi-transparent – models trained from scratch. Results for 5 downstream tasks: Code Translation, Code Generation, Code Summarization, Defect Prediction, Clone Detection.

range of pretrained models, and investigate various dimensions, including different pretraining objectives, model sizes, and the effect of finetuning.

A line of work investigate pretrained models for code in different directions. anonymous authors (2022) show that CodeBERT relies on the high token overlap between query and code solving code search task rather than deeper syntax or semantic features, and Sharma et al. (2022) shows that BERT trained on code pays more attention to identifiers and separators. Our work provides another view on the analysis of pretrained models of code, from the probing perspective, and complements these results.

Recently created BIG-bench benchmark (Srivastava et al., 2022) contains a number of challenging code-related probing tasks for testing large language models capabilites, including programming synthesis and code summarization tasks, which are close to complex downstream tasks. In contrast, we aim at developing simple probing tasks targeting specific code understanding aspects.

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, the notion of namespaces, data flow, code readability and natural language-based naming. However, pretrained models show limited understanding of code semantics, which means that their usefullness in applied tasks requiring semantic understanding of code may be limited.

Using code-specific pretraining objectives (CodeT5, GraphCodeBert) enriches the understanding of the code aspects addressed in the corresponding objective. This result may suggest practitioners to choose pretrained models which pretraining objectives are better aligned with the considered applied task.

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.

Limitations

In this section, we discuss the limitations of our probing toolkit.

Our probing setup does not cover all possible aspects of source code. However, we were aiming at covering diverse properties of code.

Our experiments are limited to the two most popular high-level languages, which are usually used to evaluate pretrained model for code: Java (7 tasks) and Python (1 task). It would be interesting to compare the models on low-level languages like C/C++.

The linear models used for probings may appear limited in their capacity, however, they were successfully used in a lot of NLP probing approaches (Belinkov, 2022) and are well suitable for particular research questions considered in the paper. Moreover, we also experiment with a 3-layer MLP and find that our main results hold for MLP.

Finally, in this work, we only considered open-sourced pretrained checkpoints. It would be interesting to compare the performance of pretrained models across a wide range of model sizes.

Ethics Statement

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.

Acknowledgments

The results were supported by the Russian Science Foundation grant №19-71-30020. The research was supported in part through the computational resources of HPC facilities at NRU HSE.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. 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, Online. Association for Computational Linguistics.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021b. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Guillaume Alain and Yoshua Bengio. 2016. Understanding intermediate layers using linear classifier probes. *CoRR*, abs/1610.01644.

LeClair Alex, Haque Sakib, Wu Lingfei, and McMillan Collin. 2020. Improved code summarization via a graph neural network. In *2020 IEEE/ACM International Conference on Program Comprehension*.

Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France. PMLR.

anonymous authors. 2022. Analyzing codebert's performance on natural language code search.

Yonatan Belinkov. 2022. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48(1):207–219.

Yonatan Belinkov, Nadir Durrani, Fahim Dalvi, Hassan Sajjad, and James Glass. 2020. On the linguistic representational power of neural machine translation models.

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

Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 703–715, New York, NY, USA. Association for Computing Machinery.

Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018a. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia. Association for Computational Linguistics.

Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018b. What you can cram into a single $&!#* vector: Probing

sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia. Association for Computational Linguistics.

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, Minneapolis, Minnesota. Association for Computational Linguistics.

Allyson Ettinger, Ahmed Elgohary, and Philip Resnik. 2016. Probing for semantic evidence of composition by means of simple classification tasks. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 134–139, Berlin, Germany. Association for Computational Linguistics.

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

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*.

Abhijeet Gupta, Gemma Boleda, Marco Baroni, and Sebastian Padó. 2015. Distributional vectors encode referential attributes. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 12–21, Lisbon, Portugal. Association for Computational Linguistics.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations*.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. 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 4129–4138, Minneapolis, Minnesota. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336.

Arne Köhn. 2015. What's in an embedding? analyzing word embeddings through multilingual evaluation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2067–2073, Lisbon, Portugal. Association for Computational Linguistics.

Fajri Koto, Jey Han Lau, and Timothy Baldwin. 2021. Discourse probing of pretrained language models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3849–3864, Online. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA. Association for Computing Machinery.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and Desmaison et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Yada Pruksachatkun, Jason Phang, Haokun Liu, Phu Mon Htut, Xiaoyi Zhang, Richard Yuanzhe Pang, Clara Vania, Katharina Kann, and Samuel R. Bowman. 2020. Intermediate-task transfer learning with pretrained language models: When and why does it work? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5231–5247, Online. Association for Computational Linguistics.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020a. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020b. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30.

Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. 2022. An exploratory study on code attention in bert.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, and Safaya et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models.

Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R. Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R. Bowman, Dipanjan Das, and Ellie Pavlick. 2019. What do you learn from context? probing for sentence structure in contextualized word representations. In *International Conference on Learning Representations*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models.

Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *arXiv preprint arXiv:1506.06724*.

A Ablation study

In this section we perform an ablation study of different code components to understand which component of code is important for each of the probing tasks, i. e. identifier names, language specific keywords (e.g. “for", “if"), or punctuation (e.g. “(", “:", “<") . We experiments with two pretrained models: pretrained on code (PLBART) and text model (BERT). For ablation of identifiers, we rely on the methodology of (Chirkova and Troshin, 2021) and apply syntax-preserving anonymization, replacing identifiers inside a code snippet with placeholders (“var1", “var2", “var3"). To ablate language-specific keywords or punctuation, we simply replace them with “MASK".

The ablation results are presented in Figure 5. Overall, for all tasks the most influential component is punctuation: masking it hurts the quality the most, except for the “Variable Name" task, where anonymizing identifier leads to the worst quality. With punctuation being masked, PLBART model is close to the quality of the text BERT model or performs even worse.

In contrast, masking language-specific keywords does not hurt the performance significantly.

To conclude, the models pretrained on code rely heavily on punctuation, for almost all tasks, and also rely on identifier names for variable related tasks.

B MLP

Linear probings may appear limited, thus we also include the results for 3-layer MLP model for comparison. We implement an MLP in PyTorch (Paszke et al., 2019) with ReLU nonlinearity and hidden size 128. We train it with AdamW (Loshchilov and Hutter, 2019) optimizer with batch size 512 and we use Optuna (Akiba et al., 2019) with 10 trials to search over learning rate (high=0.1, low=0.0001, log domain) and weight decay (high=0.1, low=0.00001, log domain) minimizing error on validation set (0.1% of train set) for regression/classification tasks. We reduce learning rate by factor 0.1 with patience 5, use early stopping with patience 10, and maximum number of update 5000.

The results for MLP (Figures 6,7) are very similar to the results with linear models, both quantitatively and qualitatively. For edge prediction in data flow graph, MLP outperformes linear model significantly, but for other tasks the results are roughly the same.

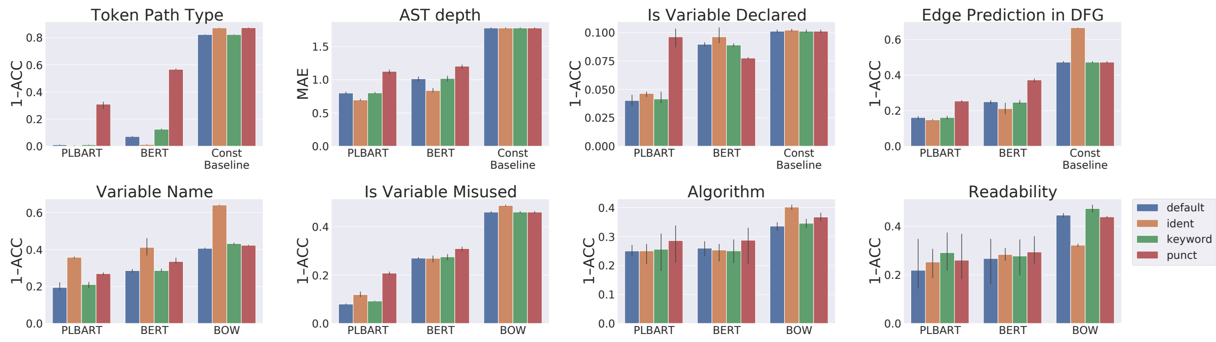

Figure 5: The results for ablation study: no ablation (default), anonymization of identifiers (ident), masking keywords (keyword), and masking punctuation (punct). Metrics are the lower the better.

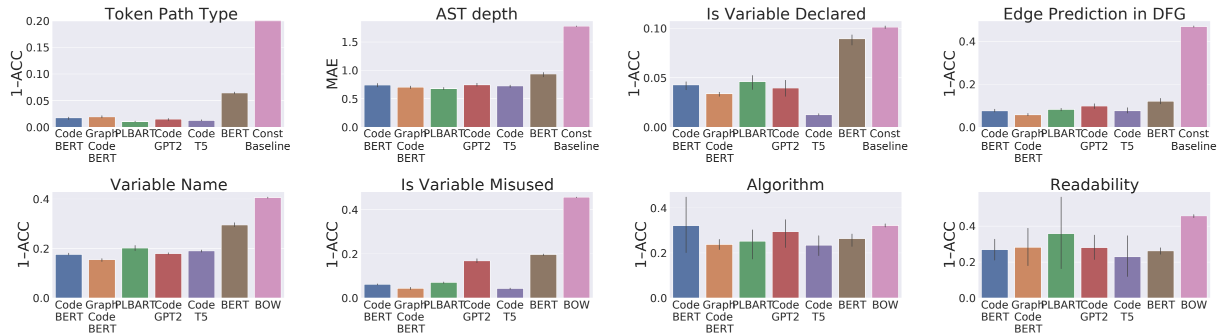

Figure 6: Results for the best performing layer representations, for all probing tasks, *3-layer MLP*. Metrics are the lower the better.

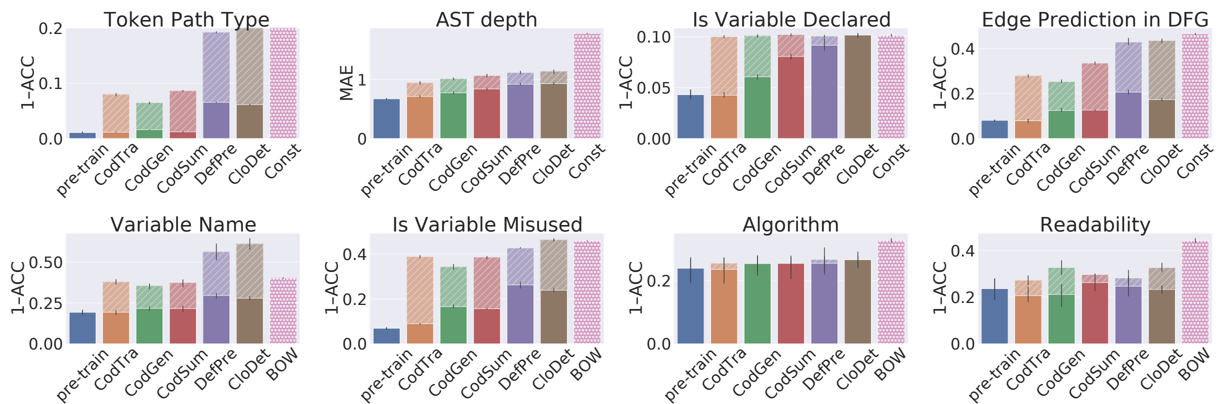

Figure 7: Results on the effect of finetuning, *3-layer MLP*. Pre-train (leftmost bar): pretrained-only checkpoint. The following bars: dark – finetuned models, semi-transparent – models trained from scratch. Results for 5 downstream tasks: Code Translation, Code Generation, Code Summarization, Defect Prediction, Clone Detection.