



HAL
open science

INSPECT: Intrinsic and Systematic Probing Evaluation for Code Transformers

Anjan Karmakar, Romain Robbes

► **To cite this version:**

Anjan Karmakar, Romain Robbes. INSPECT: Intrinsic and Systematic Probing Evaluation for Code Transformers. *IEEE Transactions on Software Engineering*, 2024, 50 (2), 10.48550/arXiv.2312.05092 . hal-04797531

HAL Id: hal-04797531

<https://hal.science/hal-04797531v1>

Submitted on 22 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSPECT: Intrinsic and Systematic Probing Evaluation for Code Transformers

Anjan Karmakar and Romain Robbes

Abstract—Pre-trained models of source code have recently been successfully applied to a wide variety of Software Engineering tasks; they have also seen some practical adoption in practice, e.g. for code completion. Yet, we still know very little about *what* these pre-trained models learn about source code. In this article, we use *probing*—simple diagnostic tasks that do not further train the models—to discover to what extent pre-trained models learn about specific aspects of source code. We use an extensible framework to define 15 probing tasks that exercise surface, syntactic, structural and semantic characteristics of source code. We probe 8 pre-trained source code models, as well as a natural language model (BERT) as our baseline. We find that models that incorporate some structural information (such as GraphCodeBERT) have a better representation of source code characteristics. Surprisingly, we find that for some probing tasks, BERT is competitive with the source code models, indicating that there are ample opportunities to improve source-code specific pre-training on the respective code characteristics. We encourage other researchers to evaluate their models with our probing task suite, so that they may peer into the hidden layers of the models and identify what intrinsic code characteristics are encoded.

Index Terms—Machine Learning for Source Code, Probing, Benchmarking, Transformers, Pre-trained models

1 INTRODUCTION

THE outstanding success of transformer-based [2] pre-trained models in NLP such as BERT [3], has inspired the creation of a number of similar pre-trained models for source code. These pre-trained models are first trained on a large corpus of code in a self-supervised manner and then fine-tuned on downstream tasks. These models include sequence-based models such as CodeBERT [4], CuBERT [5], or PLBART [6], to name a few. Other models include more structural information on source code, such as GraphCodeBERT [7], or UniXCoder [8]. Finally, several recent models leverage scaling laws [9] to improve performance just by virtue of their size, such as Codex [10], CodeGen [11], InCoder [12] and AlphaCode [13]. This effort is ongoing, with novel pre-trained models of source code being announced regularly.

These large-scale pre-trained transformer models for code have been shown to perform spectacularly well on a wide range of software engineering tasks, which led to the release of a number of new tools. These include source code completion tools, such as Tabnine, IntelliCode [14], or GitHub CoPilot, that leverages OpenAI’s Codex [10] to propose multi-line code completion from natural language prompts. More generally, the increasing availability of these AI-based development tools has triggered discussions of their promises and concerns [15].

The progress made with pre-trained source code models is genuinely encouraging. However, it remains unclear *what exactly do these models learn about source code*. Do they learn the tasks at hand, or, like the mathematic prodigy Clever Hans (a horse¹) [16], do they exploit superficial cues

[17] in their input? Are source code models “stochastic parrots” [18], that overly rely on memorization [19]? There are many such questions about source code models that can be asked, all of which impact our understanding of source code models and their capabilities.

In this paper, we focus on a more specific question, namely: to what extent do pre-trained source code models learn about the specific characteristics of source code? This question is important since pre-trained models come from the field of NLP. However, source code and natural language are very different. For instance, source code is highly structured, and can be unambiguously parsed. Functions or methods form complex structures with an intricate control flow over many statements. Source code identifiers can be formed of multiple natural language words (e.g., `ArrayIndexOutOfBoundsException`). Source code utilizes punctuation in a way that is vastly different than natural language. Small changes to source code can lead to widely different behaviour, crashes, or compilation errors.

In spite of this, several architectures have been applied as-is to source code. For instance, Codex is a variant of the GPT-3 natural language model, that is further fine-tuned on source code; it inherits its tokenization, only adding a few additional tokens to better handle source code indentation. Whether the default NLP training objectives are as effective for structured source code as it is for less structured NLP applications is an open question. On the other hand, some models, such as GraphCodeBERT [7], do adopt additional pre-training objectives to better account for source code’s characteristics. Whether the additional training objectives are effective in learning more source code characteristics is also an open question.

Before addressing these questions however, there is first a methodological issue: *how* can we evaluate specific characteristics of source code, of which there can be many? An emerging field of research addresses these with *probes*.

- Anjan Karmakar is with the Free University of Bozen-Bolzano, Italy.
- Romain Robbes is with the University of Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR5800, Talence, France.

Manuscript received —/revised —

1. https://en.wikipedia.org/wiki/Clever_Hans

Probes are diagnostic tasks, which are designed to evaluate a specific characteristic of the input data. To probe a pre-trained model of interest, one trains a simple classifier to predict specific properties of its input (as specified by the diagnostic task), based on the *frozen* vector embeddings of the pre-trained model. The degree of success in the probing tasks indicates whether the information probed for is present in the pre-trained embeddings, and to which extent. Probing has been extensively used for natural language models, and has already begun to pick up steam with numerous probing tasks [20], [21], [22], [23], [24], [25], [26], [27] investigating a diverse array of natural language properties. We provide more background about probing in the next section.

In this work, we adapt the probing paradigm to evaluate the source code knowledge within pre-trained source code models. In Section 3 we define a set of 15 probing tasks, that cover a variety of source code characteristics. The tasks cover multiple characteristics of the *Java* programming language, such as reasoning on identifiers, source code structure, and code correctness. We also present `INSPECT`, the probing framework that we developed for this study, in Section 4; `INSPECT` allows us to easily extend our probing test suite by defining new tasks, as well as to very easily probe models that are made available on the huggingface model hub. The framework is freely accessible on github at <https://github.com/giganticode/inspect>. We encourage other researchers to use `INSPECT` as an **intrinsic benchmark** of their source code models.

We use our tasks to probe 8 pre-trained code models, and also a natural-language model which serves as our baseline. We provide a detailed methodology of our study, and describe the models and the datasets that we use in section 5. The models range from ones that *should have no knowledge* of source code (the BERT model, trained on a natural language corpus), to models that should have moderate knowledge of source code (e.g., CodeBERT which follows BERT’s training procedure, but on a source code corpus), up to models that should have a more advanced knowledge of source code (e.g., GraphCodeBERT, which has a code-specific pre-training objective, along with data-flow information).

We present the results of our study in section 6. We answer research questions related to the general performance of the models on the probing tasks, as well as comparing the performance of different models, the performance on different categories of tasks, as well as the performance across layers. Overall, we find while the pre-trained models encode some source code characteristics well, there is room for progress: for some tasks, even the most advanced model fail to significantly outperform our baselineBERT, that should have no explicit knowledge of code. We also observe that models that introduce more structural source code information in their training tend to perform better, which provides a way forward.

Finally, we discuss the results 7 and the limitations 8 of this work in section, and conclude in section 9. In particular, we conclude that defining new pre-training procedure that better emphasize source code characteristics should be developed and more systematically investigated, such as via our probing framework, `INSPECT`.

This work is an extension of a short paper previously published in ASE 2021 [1]. Compared to our previous work, we perform a much more extensive study. During the course of this work, we designed 19 new tasks, eventually keeping 13 in the final version, and replacing 2 of the original 4 tasks. Overall, 85% of the tasks are new to this paper. The tasks are divided in 5 different task categories. Having multiple tasks for each category (rather than a single one) allows us to find trends about which source code characteristics tend to be better encoded in the models (e.g., semantic characteristics tend to be better encoded than structural characteristics). The original work evaluated 3 models and a baseline, while this version evaluates 8 models (and the baseline). Having a wider selection of models allow us to find or dismiss certain factors that might impact the quality of the encoding of the characteristics in the models (e.g., models with code-specific training objectives appear to better encode source code characteristics, whereas models that use more generic training objectives are less successful). To account for the growth in the amount of data to present (close to 95% of the data is new to this paper), we conducted an entirely new and much more extensive analysis of the results. This analysis looks at additional research questions and sub-questions in ways that are very different from the original paper. Following the results, we added an extensive discussion of the results, their implications, and a discussion of their relationship to the emerging literature on probing and analyzing source code models. We also expanded the background and limitations section to make the paper much more self-contained.

2 PROBING: BACKGROUND AND RELATED WORK

We provide an introduction to the notion of probing as performed in this study. We invite readers wishing to further explore the topic, to consult the recent survey on probing [28], which also discusses some advanced approaches.

2.1 Probing Neural Networks

The goal of probing is to assess to what extent a Deep Neural Network learns an implicit representation of specific characteristics of its training data during training. Such characteristics are varied and very domain-specific, such as:

- In Neural Machine Translation (NMT), whether a model has some representation of the active/passive voice or tense of the original input [21];
- In Natural Language Processing (NLP), whether a natural language model encodes some information about the word order of a sentence [22];
- Whether a neural chess engine encodes representations of chess concepts, such as the playing side being in check, or the strength of chess pieces [29];
- Whether a source code model encodes representations of a code snippet’s inherent complexity [1]

Note that these characteristics are usually *implicitly* present in the training data, rather than being explicitly fed to the models while training. E.g., the training data for an NMT model is simply the text to translate, and does not explicitly indicates the tense or the active/passive voice of a sentence.

Representation learning. Deep Learning models learn vector representations of their input during training. The learned representations are based on multi-dimensional internal weights, which are optimized during the learning process. Once a model is trained, the learned vector representations can be obtained by processing any input through the model and collecting the weights from the hidden state.

Probing. We use *probes* to ascertain if, and to what extent, learned vector representations of pre-trained models encode specific characteristics of interest; and whether they have been implicitly learned by the models. A *probe* consists of two components: 1) a *probing task* and 2) a *probing classifier*.

Probing Task. A probing task is an auxiliary diagnostic task, generally designed to analyze implicit learning in models. A probing task may require models to predict certain characteristics from the input features given that the model is not directly trained to predict them in the pre-training stage. Each probing task should be a convincing proxy for a given characteristic of interest. For instance, to verify whether a pre-trained code model has some implicit knowledge of code size, a diagnostic task could be constructed to probe on the model’s (frozen) pre-trained weights whether it can classify code snippets by size. If the pre-trained vectors of the model encode such information, the probe results will reflect the same. Similarly, a diagnostic probing task could be used to probe for the concept of word order, to randomly alter half of a set of sentences by swapping two words, and classify inputs as original or altered [26].

Probing Classifiers. One critical aspect regarding probes is that *the pre-trained models are not further trained on the probing tasks*. Rather, the model’s learned pre-trained weights are first extracted and used to compute the vector representations of each input sample (i.e. the pre-trained model is *frozen*), which are then passed on as input features to a *simple* probing classifier, typically a simple linear classifier. Note that the probe does not have access to the original input, just the resulting vector representations. This makes probing for even very simple concepts still informative, as they are not directly accessible to the model.

Only the linear classifier is optimized when training on the diagnostic task. During training, the probe should learn which of its input features—if any—are relevant for the task at hand. Thus, if one or more features encode the concept of interest, the probe should be able to perform the task successfully. On the other hand, if the original model has not learnt the concept of interest during its training, the probe should fail at the task, with a performance on a test set that differs little from random chance accuracy.

Probe Complexity. One of the assumptions behind probing is that a representation of a source code characteristic should be *simple*. Complex representations are more likely to contain spurious relationships (e.g., a model might notice a relationship between the length of a piece of code and its complexity, rather than modelling complexity directly). One way to enforce this is by using simple classifiers: if a simple classifier can successfully solve the task, it means that the representation of the concept is accessible and easy to extract from the model. We use a linear layer as the probing classifier. These are similar in complexity to

a standard classification head that might be attached to a pre-trained model. Note that a linear layer can only learn linear combinations of its input: this limitation imposes a constraint on the simplicity (and thus, in terms of probing, its quality) of the representation of the existing concept in the pre-trained model. Other works use a Multi-Layer Perceptron (MLP) with one hidden layer as a probe. These can learn more complex non-linear combinations of features. If only a complex probe can solve the diagnostic task, there is a risk that the probe itself learns to solve the task as the training progresses, rather than relying on the actual pre-trained vectors of the pre-trained models [30].

Summary and Source Code Example. The goal of a probe is to test whether, and to what extent, a specific characteristic of the training data is encoded in a model’s internal representation. For instance, we might want to test whether a pre-trained source code model such as CodeBERT encodes any notion of the complexity of source code. To do so:

- 1) We first collect a set of training/test inputs (e.g. *Java* code snippets at the method-level) and labels (e.g. each method’s Cyclomatic Complexity).
- 2) We then proceed to extract learned vector representations for each input sample in the task dataset by processing them through a pre-trained model.
- 3) Finally, we train a linear classifier on the extracted vector representations, and evaluate the probe on the test set. This gives us the performance metrics for each layer of the model—with predictions based principally on the learned vector representations.

Fig. 1 illustrates the above steps where pre-trained vector representations and corresponding property labels are used to probe for certain intrinsic information which are expected to be encoded in the model layers.

2.2 Related Work

Probing in NLP. An early study by Shi *et al.* found that LSTM machine translation models capture several concepts of syntax, such as voice, tense, or part of speech [21]. Belinkov *et al.* focused on finer-scale concepts, such as word morphology, finding that character-based machine translation models better capture morphology [23]. Conneau *et al.* defined 10 probing tasks, including sentence length, tense, AST tree depth, and others, finding that the performance on probing tasks outperform several simple baselines [26]. The study by Jawahar *et al.* [31] show that BERT encodes phrase-level information in the lower layers, and a hierarchy of linguistic information in the intermediate layers, with surface features at the bottom, syntactic features in the middle and semantic features at the top of a vector space. Hewitt and Manning designed a *structural probe* [32] that evaluates whether pre-trained models such as BERT [3] or ELMo [33] have representations of a sentence’s *entire* syntax tree embedded as implicit node distances in the vector representations: they find that this is the case for medium to late layers. Clark *et al.* probed BERT’s attention heads, finding that some individual heads attended to specific linguistic concepts [34]. The previous studies are but few examples: studies of the BERT models alone have spawned a subfield known as *BERT-ology* with over 150 studies surveyed [35].

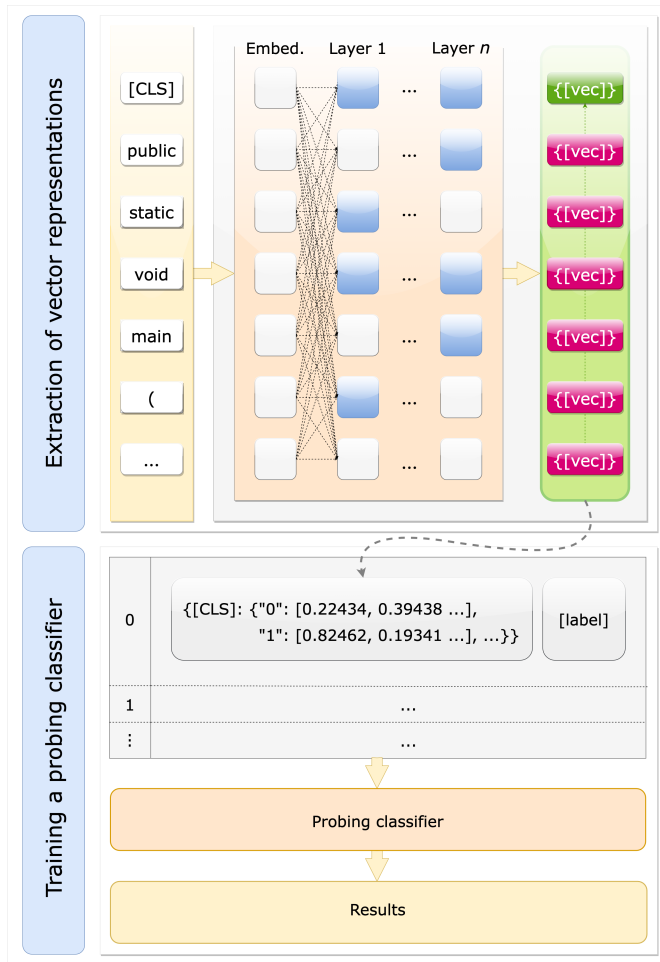


Fig. 1. Overview of the probing procedure

Probing in other fields. Probing has been applied beyond textual models only. DeepMind’s AlphaZero game playing models have been probed for the games of chess [29] and Hex [36]. For chess, the model uses a Convolutional Neural Network to encode the board; the CNN is probed for close to 150 individual chess concepts (e.g., king safety, threats to pieces). Cao *et al.* applied probing to Vision and Language models, that reason on images and text at the same time [37]. Recently, studies relate the hidden state of language models with fMRI data obtained from humans reading the same text the language models are processing [38], [39], [40], finding correlations between some of the language model’s activation weights, and brain activity.

Probing in Software Engineering. Our initial work [1] was the first dedicated study of probing for source code models. The only earlier example we are aware of uses a single coarse task (programming language identification)—and is not the focus of the paper [4]. Since our initial work, several additional studies have emerged.

Wan *et al.* analyzed two pre-trained code transformers, CodeBERT and GraphCodeBERT [41]. They first analyzed the model’s attention heads, and whether the attention aligns with the syntax structure of code, finding that it does. They then probed the word embeddings to find out if they embed values similar to inter-token AST distances, finding

that it does, but that it varies among layers. Finally they checked whether they could reconstruct the AST from the learned representations, finding that they can to an extent.

Other works focused on attention more specifically. Chen *et al.* introduced CAT-probing [42]. They define a metric, the CAT-Score, to relate the attention matrix (how the Transformer’s attention for a given token attends to other tokens) for a given layer, with the distance matrix of the AST nodes. They find that CAT-scores for source code models are correlated with their performance on code summarization, and that the CAT-scores vary per layer and per language, with a tendency for higher scores in the earlier layers. Zhang *et al.* [43] define a similar metric over the attention matrix, an aggregated attention score, with which they can derive a graph of relationships between tokens.

Hernández Lopez *et al.* use the structural probe of Hewitt [32] to study various pre-trained models of source code [44]. They find that CodeBERT and GraphCodeBERT better capture the AST than other code models such as CodeT5 and CodeBERTa, and that the AST representation is more encoded in the middle layers.

These works employ for the most part a holistic approach to probing source code models, primarily focusing on the entire AST. In contrast, we define multiple probing tasks to probe for the presence of more fine-grained concepts than the entire AST. The closest work to ours is the work by Troshin and Chirkova [45] which builds up on our initial work by defining new probing tasks. They define 8 tasks which do not overlap with our 15, as we have an increased focus on metrics and incorrect code detection. In addition, space allows us to conduct a significantly more in-depth analysis of our results (which also span a broader selection of models). Another important difference is the original data used. For 6 tasks, Troshin and Chirkova use an existing dataset of 10,000 java methods with which they derive individual task datasets. The relatively small size of the data prevents the selection of subsets of the data to calibrate the difficulty of the task. For instance, the baseline for their “is variable declared” task (a constant prediction that the variable is declared) has a 90% accuracy. The initial datasets for their two other tasks have 934 and 200 datapoints. In contrast, we start with a much larger initial dataset (with 8 million methods), allowing us to select specific subsets of the data (e.g., all our datasets are balanced). As a consequence of our different task selection and dataset construction, we tend to find lower performance for the source code models. In section 7.6, we compare our results with this study.

Finally, Paltenghi and Pradel [46], and Paltenghi *et al.* relate neural attention weights with human eye-tracking data [47]. They find some correlation between the neural attention weights of language models and eye-tracking data.

3 THE INSPECT TASK SUITE

In this section we introduce our suite of probing tasks. Our probing tasks consider a variety of source code characteristics, grouped in several categories. Selecting the tasks for the probing suite among a pool of candidate tasks proved to be challenging: there are many characteristics of source code that we could select, ranging from the most basic to the most intricate. We thus selected tasks to cover a broad set

of source code characteristics, while minimizing (as much as possible) the number of tasks in our suite. At the highest level, we divide the tasks into 3 broad categories:

- 1) **Token-based tasks**, which include code token tagging tasks and total token count task (Section 3.1).
- 2) **Metrics-based tasks**, which include tasks related to identifying code structures, and unique occurrences of operators and identifiers; as well as implicit understanding of code complexity (Section 3.2).
- 3) **Incorrect-code tasks**, which include switched code, misspelled, and jumbled code tasks (Section 3.3).

The above tasks probe for one or more of the code characteristics with regard to syntactic, semantic, structural, or surface information. In addition to the above tasks, we discuss some candidate tasks that were considered but have not been selected for this study (Appendix B). Should the need arise, our suite of probing tasks is extensible: we have developed a framework (Section 4) to this effect, to help users conduct extended probing studies on further models.

3.1 Token-based tasks

Tokens are the most basic code elements a pre-trained model can reason on. For instance, in order for a pre-trained model to be good at code completion, it must necessarily learn and interpret the syntactic formation of a sequence of code tokens, and predict a syntactically valid next token. Thus, the capacity of source code models to infer the type of individual tokens is an elementary source code characteristic that we want to specifically evaluate. We first define two tasks that specifically target individual tokens: one for the specific case of identifiers (IDN), and another for the other types of tokens (KTX). Subsequently, we define a task to probe for the size of the input, e.g. number of tokens (LEN).

Identifier Tagging (IDN) Source code identifiers make up most of source code, up to 70% according to some estimates [48]. Further, there are coding conventions that influence the format of the identifiers according to its role (e.g. a class name is different from a variable or method name). IDN determines whether pre-trained models have sufficient syntactic and semantic knowledge to distinguish among the different types of user-defined identifiers. Pre-trained models must classify input source code tokens as one of four types of identifiers (package, class, method, and variable names). Unlike most other tasks, we feed a single identifier to the model, rather than a method; the probing classifier uses the [CLS] token as usual. We do this because it is uncommon to find package names in method source code; instead, we collect package names from import declarations, which are much more common.

Keyword Tagging (KTX) This task evaluates the capacity of pre-trained models to discriminate between various types of keywords, operators, and symbols. The goal of the task is to learn the differences between types of keywords (e.g., *float* is in a different category than *return*). We define several classes for keywords, operators, and symbols. We divide keywords into 4 keyword classes, e.g. modifiers (*private*, *protected*, *public*), flow control (e.g., *for*, *if*, *return*),

etc. We divide operators in 5 classes e.g. arithmetic, assignment, relational, etc. We also add *Java* symbols, in a single class (e.g., "{", ";", "}").

We divide each of the classes in training, validation, and test sets to specifically probe for the ability of the models to generalize. For instance, for primitive types, we might have: *int*, *double*, *char*, *boolean* in the training set; *string*, *long* in the validation set, and *float*, *short* in the test set. Thus the probe should look for features that describe the *categories* of tokens in the learned representation, rather than specific tokens. For this task we extract the representation of the specific keyword, symbol, or operator token in question, corresponding to the label, rather than the aggregated input representation, i.e. the [CLS] token.

Code Length (LEN) Length is the most basic attribute we can probe for which may be implicitly learned by the models given any input. It is also the least dependent on source code knowledge, relying on the number of tokens, and as thus is useful to frame the performance of the other tasks. We measure code length in tokens. Since method sizes vary widely, we define 5 classes by binning the values into 5 different labels. Note that predicting the length of a code snippet may appear trivial, but this is not the case: since we probe the vector representation of a single summary token from the models (see section 5), the amount of information the models can use may be limited.

3.2 Metrics-based tasks

Since many models are trained and evaluated on tasks that concern methods (e.g., method naming, method summarization), the method is the next most natural unit to examine. This category of tasks is based on the ability of models to detect quantifiable characteristics of source code.

Since metrics are numeric values, we could have formulated these tasks as regression tasks. We ultimately chose to formulate them as classification tasks for two reasons: in order to be more consistent with the other tasks of the suite—comparing models across all tasks in terms of accuracy, and, to reduce the impact of confounding factors. For example, since some metrics can be correlated to code size, a model that predicts, e.g. a high complexity for all larger methods, could score deceptively well in a regression scenario. We observed this during the design of this study when comparing regression and classification variants of the same task. By defining classes and requiring an exact prediction on those classes, we reduce the incentive of models to rely on factors that might be indirectly correlated.

We subdivide the metrics-based task into two sub-categories: a) count-based metrics, which probe for surface-level metrics of the method, that provide general indicators on the code understanding of the model (OCU, VCU, CSC), and b) complexity-based metrics, which measure various aspects of a method’s complexity (MXN, CPX, NPT).

The count-based metric tasks describe basic characteristics of source code. There could be several alternative metrics that we could use in this category. We focused on metrics describing the most general concepts, and ultimately settled on operators (+, *, etc.) (OCU), and variables (VCU), and code structures (*if*, *for*, *switch* etc.) (CSC).

With the complexity-based metric tasks, we analyze more advanced structure-based and control-flow based reasoning in pre-trained models. Source code complexity involves having a detailed view of the entire method. Since there are various measures of code complexity, we probe for three different aspects of varying difficulty. From easiest to hardest, our tasks are: *MXN*, *CPX*, and *NPT*.

Operator Count Unique (OCU) Operators are one of the most elemental way to manipulate data in source code. This task probes the models for their representations of the concept of source code operators. We count the number of *unique* operators present in the method (e.g., multiple assignment operators will be counted only once). Thus, the task emphasizes more operator diversity than simply counting operators, and is less affected by code size. We define 10 classes representing the exact count of unique operators, without binning.

Variable Count Unique (VCU) Identifiers are omnipresent in source code. *VCU* probes the models for their representations of identifiers at the level of a method. Similarly to *OCU*, we count the number of *unique* identifiers in the method, rather than the total count, to emphasize diversity rather than counting. Likewise, we define 10 classes that represent the exact count of unique variables, without binning.

Code Structure Count (CSC) This task probes for a more comprehensive view of control flow in the method. Here, we count all the code structures present in the method (e.g., *if*, *for*, *do-while*, *while*, *switch* or *try* blocks). Thus this task goes beyond *NMS* and *NML* as it probes for additional structures. For *CSC* we define 10 classes, representing the total count of code structures in the method, without binning.

Maximum Indentation (MXN) We first test for a *proxy* of source complexity, which is the depth of indentation. Indentation often reflects complexity as code blocks inside other code blocks are, by convention, further indented to the right. Indeed, Hindle *et al.* found correlations between indentation level and various complexity metrics [49]. We define 5 classes, as deeper indentation levels are uncommon.

Cyclomatic Complexity (CPX) Cyclomatic complexity is an established measure of complexity: it counts the number of linearly independent paths in the control flow graph [50]. This requires a more advanced level of reasoning as it requires not only knowing about the existence of code structures, but also how they relate to one another and how they might affect control flow. We measure Cyclomatic Complexity to define 10 classes corresponding to exact complexity values ranging from 0 to 9, without binning .

Npath Complexity (NPT) Npath complexity measures the number of possible execution paths in a method, excluding cycles [51]. Compared to Cyclomatic complexity, this metric has a much higher spread of values as the number of possible paths grows faster than the number of linearly independent paths. For a pre-trained model, we expect that reasoning about Npath complexity is thus more complex than Cyclomatic complexity. To accommodate for the larger range of values and *NPT*'s exponential behavior, we use binning to manually define 10 classes. The bins progressively widen: 1, 2, 3, 4–6, 7–8, 9–10, 11–15, 16–20, 21–30, 31–100.

3.3 Incorrect-code tasks

Models trained on existing source code should have some notion of what correct code looks like. Indeed there is evidence that buggy source code is less *natural* than fixed source code for n-gram models [52] and for *LSTMs* [53]. For the final category of tasks, we probe the ability of model to differentiate between code that is correct and incorrect in various ways. Most of the incorrect code samples in these tasks would be easily caught at compile time. Thus, we use these probes to determine whether pre-trained models understand such inconsistencies in source code inputs.

We divide the incorrect-code tasks into two sub-categories: a) mistyped code probes (*TYP*, *REA*, *JBL*), and b) semantic replacement probes (*SRI*, *SRK*, *SCK*), which provide general indicators on the degree of syntactic and semantic understanding of the models.

Detect Invalid Types (TYP) For this task, negative samples have a single primitive datatype that is intentionally misspelled (e.g., *float* to *flaot*, or *folat*). This allows us to probe to which extent models are sensitive to mistypes that would be easily detected by a compiler. For each negative sample, we misspell a single datatype, while there may be other correct occurrences of the same datatype in the sample.

Relational operator to Assignment (REA) The Relational to Assignment Operator mistyped task evaluates whether pre-trained models can distinguish between the syntactic usages of relational and assignment operators. Some code snippets are mutated so that the relational operators are switched with assignment operators (e.g. `<=` may be switched with `+=`, or `!=` with `/=`). The mutated code may look superficially correct, therefore, such a change may be difficult to detect.

Jumbled Code Tokens (JBL) For this task, the invalid code snippets are mutated by swapping two consecutive source code tokens, chosen randomly (e.g., `int foo = 4; to int = foo 4;`). This task probes the sensitivity of models to the order of tokens: this is a small change at the level of a method, but a model that has knowledge of source code syntax should perform well in this task.

Switch Random Identifier (SRI) This is a semantic token replacement task where one occurrence of a valid identifier is replaced with another non-identical identifier from the same sample chosen randomly. The mutated code looks superficially correct: without additional information (e.g., which class, variables, and methods are in scope). Thus *SRI* probes models for their ability to determine if an identifier is *out of context*.

Switch Random Keyword (SRK) Invalid code snippets are mutated by replacing a language keyword such as *int*, with any random keyword such as *for*. Thus the probability that the token is not compatible to the context is high. This probes the models for their sensitivity to language rules—syntax and semantics—in a specific context.

Switch Compatible Keyword (SCK) Invalid code snippets are mutated by replacing a language keyword such as *double*, with another compatible keyword from the same keyword category such as *int* (primitive datatypes). Another example is switching *assert* with *throws*, both of which belong to the same keyword category (error handling).

4 THE INSPECT PROBING FRAMEWORK

INSPECT is the framework that we make available for researchers to perform probing studies. INSPECT is written in Python but it is programming language agnostic—it can work with datasets with samples in multiple programming languages. In our case, we use the JEMMA Dataset [54] to define probing tasks for the *Java* programming language.

Task definition. We use the JEMMA Dataset, derived from 50K-C Dataset [55], in conjunction with the JEMMA Workbench to gather data from a large corpus of *Java* source code. Once a dataset is created, INSPECT automates the rest of the work related to probing evaluation. We have a set of 15 probing tasks that can be readily used to probe further models. However, since tasks can vary widely depending on the code characteristic of interest and programming language of interest, users may also define additional tasks.

Get frozen representations. The first step that INSPECT automates is getting the learned representations from the pre-trained models. INSPECT makes it easy to interface with models that are on the HuggingFace model hub, the most common way to share Transformer-based models². Given a pre-trained model url, INSPECT will download it, pass each task sample (from training, validation, and test sets) through the model, and collect the model’s activations at each layer.

Train Probes. INSPECT then uses the vector frozen representations as input for each probe sample and collects the corresponding labels from the original datasets. Following which, INSPECT trains and evaluates the probing classifier on the probing tasks. The original model is not fine-tuned during this step

Compute results. INSPECT collects performance data of the probe from each model layer. The framework generates extensive data visualizations of the model’s performance, including raw and derived performance metrics exported as CSVs for further analysis, heatmaps of the model’s per-layer performance, and confusion matrices of predictions.

Time and space constraints. Once a new probing task is defined, the process is entirely automated. Depending on the task, dataset size, and number the models to probe on, the amount of space (to store the frozen representations), and time (to train the probe) needed may vary. In our case, the amount of space needed to store the frozen representations for each task is about 9-12 GB, considering just the representations from the models that are included in our evaluation. We estimate that running a single probe (in our case, a probing task with a sample size of 10k) on 8 source code models requires roughly 6 hours, including pre-trained feature extraction, hyperparameter tuning, and probing classification, on a single NVIDIA 2080Ti GPU. The time and space considerations multiply, when more probing tasks (and/or models) are considered.

Availability INSPECT is open-source and is available on GitHub at: <https://github.com/giganticode/inspect>. All probing task datasets used in this paper are also available in the same repository. We encourage researchers to evaluate their models with our probing suite, and to extend our suite with new probing task datasets exploring further characteristics.

2. <https://huggingface.co/models>

5 METHODOLOGY

5.1 Datasets, metrics and preprocessing

Datasets. To build the datasets for our probing tasks, we retrieved our code samples parsed at the method-level from the JEMMA Dataset. For each task dataset we carefully selected 10,000 suitable *Java* method samples, taking care that all of the classes are balanced (1,000 samples per class if the task has 10 classes; 2,000 if it has 5; 5,000 for binary classification). We exclude basic Java getters and setters to make the task less reliant on these easy methods. When possible, we select methods that are shorter, to avoid truncation; however some rare labels do not make this always possible. For models that have a window of 512 tokens, on average, 3% of methods are truncated. We also define a smaller dataset for each task, with 1,000 samples only, to study the impact of less data. For each probing task we split the data into training, validation, and test datasets in the following ratio: 60-20-20.

Performance metrics. Since all of our tasks are balanced, we use classification accuracy as the standard metric in this study. Due to the large number of tasks, accuracy also allows us to simplify the presentation compared to using multiple metrics such as recall, MSE, or other regression metrics.

We considered using regression for the metrics task but after judicious assessment elected not to do so, both for consistency and to reduce confounding factors, as explained in Section 3.2. Another justification is that the raw metrics are not as important in the case for probes, as much as the pattern of learning evident in the model layers, and the relative performance among models. In fact, the regression scores for such tasks show identical learning patterns across the layers for the evaluated list of models. When summarizing the results at a higher level, we simply rank the models.

Preprocessing. We tokenize the method samples, which removes the comments, tabs, and new line symbols, which transforms the raw source code into suitable input code representation for the pre-trained models. The method samples are truncated if the maximum sequence length of the model is exceeded, even though it is not a common phenomenon in the samples of our probing datasets.

Collecting feature data. We pass each sample through the model(s), and collect the vector representation of the sample at every layer. We do not collect the representation of individual tokens, as the features would be too large. We focus on the features of *summary tokens*, e.g., models that follow the BERT architecture have a special [CLS] token at the beginning of the sequence that is used as the overall representation of the entire input [3]. We follow the documentation of the models on the HuggingFace model hub or the respective papers in order to find the correct summary token if they do not have a BERT-style architecture.

Training. For training the linear probing classifiers, we train with a batch size of 1 for 20 epochs; we utilize early stopping with a tenacity of 5; we use the Adam optimizer to adapt the learning rate; and we use l2 regularization—optimizing the value of the coefficient during hyper-parameter tuning. The classifier itself is a single linear layer, with a classification

layer on top, which takes as input the frozen representation; the size of the layer matches the size of the input.

Training performance is evaluated on the validation set. Note that the hyper-parameters are simpler and less impactful since the only training that we do concerns linear probing classifier. We repeat this process for each model and each model layer. Finally, we gather evaluation data on the test set, which we use to present the results of this paper.

5.2 Pre-trained Transformer models

We evaluate 8 pre-trained source code models (Table 1), and compare their performance against a baseline model, BERT, which is **not** specifically trained on source code.

BERT is primarily pre-trained on the English language (BooksCorpus—800M words, Wikipedia—2.5B words), using the following objectives: Masked Language Modeling (MLM, predicting masked tokens), Next Sentence Prediction (NSP, classifying whether two sentences are related).

In contrast to the source code models evaluated in this study, BERT is an interesting reference point as a baseline, since we do not expect that it has knowledge of source code. We briefly discuss the source code models in the following paragraphs.

5.2.1 Encoders

CodeBERT [4] is trained unimodally and bimodally: on just source code, and, on source code with comments. The dataset used is CodeSearchNet [56], which has 2.1M bimodal and 6.4M unimodal data points, in six programming languages: *Java*, *Python*, *JavaScript*, *PHP*, *Ruby*, *Go*.

Training objective. The MLM objective is used for the bimodal data points, while a Replaced Token Detection (RTD) objective is used on all the data points. The RTD objective is used to classify whether tokens are original or substituted. CodeBERT uses n-gram models as generators, one for code, one for natural language.

CodeBERTa [57] is pre-trained on data from the CodeSearchNet dataset following the RoBERTa training objective [58]. Note that this model is considerably smaller than the other ones, with only 6 layers and 84 million parameters.

Training objective. RoBERTa keeps BERT’s MLM objective, but makes it dynamic (each training epoch masks different tokens); it also does not utilize BERT’s NSP objective.

GraphCodeBERT [7] is the model that uses the most structural source code information. During pre-training, GraphCodeBERT takes as input the nodes of the data-flow graph (DFG), in addition to source code and natural language comments. It is also trained on CodeSearchNet.

Training objective. It uses three pre-training objectives: MLM; DFG edge prediction (attention edges are masked for 20% of the nodes and should be predicted); and Node Alignment (predicting edges between code tokens and DFG nodes, for 20% of DFG nodes).

JavaBERT [59] is transformer-based source code model trained specifically on *Java* source code (~3M *Java* source code files). We use the *javabert-base-cased* checkpoint since *Java* is case-sensitive, and it showed better performance overall compared to the other checkpoint. With just 110M parameters it is one of the smallest models that we probe.

Training objective. JavaBERT is based on the BERT model with the same masked-language-modelling (MLM) training objective. Note that unlike other models, the input is not processed to be at the level of functions.

5.2.2 Encoder-Decoders

PLBART [60] is a bidirectional and auto-regressive transformer which is pre-trained on source code and natural language. It is based on the BART model [61]. The authors of the paper have made multiple models available: we use a version of PLBART that is trained specifically on *Java*.

Training objective. It is based on the BART model and pre-trained on a denoising objective: a corrupted data point must be reconstructed. Three noise sources were used: masked tokens, deleted tokens, and token infilling.

CodeT5 [62] uses the T5 architecture [63], which unifies all tasks as text generation tasks. CodeT5 is trained on CodeSearchNET, along with additional data for C and C#.

Training objective. CodeT5 inherits T5’s Masked Span Prediction (MSP) objective, which is similar to MLM, but the mask can hide a sequence of 1 to 5 tokens. CodeT5 is even pre-trained on text-to-code and code-to-text generation. CodeT5 also uses identifier-specific pre-training tasks: Identifier Tagging (classifying tokens as identifiers or not), and Masked Identifier Prediction (predicting all the identifiers in a snippet).

UnixCoder [8] is an unified encoder-decoder pre-trained model, which is trained in a cross-modal manner leveraging AST and code comments to enhance code representations. We probe the *unimodal* checkpoint in encoder-only mode since it showed better results for all the probing tasks.

Training objective. UnixCoder uses three NLP training objectives: MLM, classical left to right next-token language modelling, and MSP. It also uses two code-specific objectives at the level of the entire code fragment. The entire code fragment is encoded and used for cross modal generation (generating the text comment) and contrastive learning (finding the right text comment among several random comments from the same batch). UnixCoder is pre-trained on flattened ASTs, but uses the leaves of the AST (i.e., raw source code) for fine-tuning and inference. In addition, the model utilizes mask attention matrices with prefix adapters to control its behaviour (encoder-only, decoder-only, encoder-decoder).

CodeReviewer [64] is a model designed to understand code to assess logic, functionality, latency and other factors as part of code reviewing activities. The model is an encoder-decoder model based on the T5 architecture. It is the largest model we probe with 223M parameters.

Training objective. Distinct from other models, this model is pre-trained on code review diffs, rather than source code only, with four pre-training objectives: Diff Tag Prediction (DTP, was a line added, deleted, or kept), Denoising Code Diff (DCD, generating a code line based on context and diff tag), Denoising Review Comment (DRC, MSP for review comments), and Review Comment Generation (RCG, generate review comment given code and other comments).

| Model | Params. | Layers | Heads | H. Dim. | URL |
|---------------|---------|--------|-------|---------|---|
| CodeReviewer | 223M | 12 | 12 | 768 | https://huggingface.co/microsoft/codereviewer |
| CodeT5 | 220M | 12 | 12 | 768 | https://huggingface.co/Salesforce/codet5-base |
| PLBART | 140M | 6 | 12 | 768 | https://huggingface.co/uclanlp/plbart-multi_task-java |
| GraphCodeBERT | 125M | 12 | 12 | 768 | https://huggingface.co/microsoft/graphcodebert-base |
| CodeBERT | 125M | 12 | 12 | 768 | https://huggingface.co/microsoft/codebert-base |
| UniXCoder | 125M | 12 | 12 | 768 | https://huggingface.co/microsoft/unixcoder-base-unimodal |
| JavaBERT | 110M | 12 | 12 | 768 | https://huggingface.co/CAUKiel/JavaBERT |
| CodeBERTa | 84M | 6 | 12 | 768 | https://huggingface.co/huggingface/CodeBERTa-small-v1 |

TABLE 1
Probed models at a glance, ordered by size

| | KTX | IDN | LEN | TYP | REA | JBL | SRI | SRK | SCK | OCU | VCU | CSC | MXN | CPX | NPT |
|-----------------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Random | 10 | 25 | 20 | 50 | 50 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 20 | 10 | 10 |
| BERT | 54.0 | 67.8 | 81.3 | 89.9 | 65.3 | 51.4 | 54.6 | 62.4 | 61.5 | 20.4 | 22.5 | 31.8 | 59.5 | 34.6 | 29.0 |
| CodeBERT | 71.8 | 79.5 | 82.8 | 95.3 | 83.5 | 71.2 | 70.5 | 61.5 | 62.7 | 24.0 | 31.1 | 39.0 | 64.7 | 39.5 | 32.3 |
| CodeBERTa | 73.4 | 73.0 | 83.4 | 93.7 | 75.8 | 53.6 | 58.3 | 59.6 | 61.2 | 18.5 | 23.2 | 31.5 | 45.6 | 29.0 | 29.3 |
| CReviewer | 79.2 | 79.4 | 88.5 | 89.1 | 76.2 | 52.6 | 54.2 | 61.9 | 59.5 | 19.3 | 22.4 | 29.2 | 51.0 | 28.1 | 29.3 |
| CodeT5 | 70.8 | 75.3 | 86.3 | 93.8 | 80.0 | 54.8 | 62.2 | 64.2 | 65.0 | 20.5 | 24.1 | 32.5 | 57.4 | 31.1 | 31.5 |
| GCodeBERT | 66.2 | 78.8 | 80.0 | 95.9 | 80.8 | 69.4 | 71.0 | 64.3 | 61.9 | 23.9 | 33.7 | 39.5 | 63.4 | 36.8 | 35.4 |
| JavaBERT | 60.8 | 76.8 | 75.2 | 88.8 | 65.7 | 52.9 | 58.8 | 58.5 | 59.0 | 19.6 | 20.5 | 28.9 | 44.3 | 28.1 | 25.0 |
| PLBART | 72.7 | 76.5 | 86.7 | 89.3 | 62.1 | 50.1 | 50.5 | 59.0 | 58.3 | 15.1 | 18.0 | 26.8 | 36.8 | 25.1 | 24.8 |
| UniXCoder | 73.5 | 75.6 | 81.8 | 90.8 | 65.0 | 52.9 | 66.1 | 52.9 | 56.9 | 20.9 | 27.8 | 30.6 | 51.4 | 28.1 | 29.3 |
| Maximum (10k) | 79.2 | 79.5 | 88.5 | 95.9 | 83.5 | 71.2 | 71.0 | 64.3 | 65.0 | 24.0 | 33.7 | 39.5 | 64.7 | 39.5 | 35.4 |
| Rank | 5° | 4° | 2° | 1° | 3° | 6° | 7° | 10° | 8° | 15° | 14° | 11° | 9° | 11° | 13° |
| Std. dev. (exl. BERT) | 5.1 | 2.1 | 4.0 | 2.7 | 7.7 | 7.7 | 6.9 | 3.4 | 2.5 | 2.7 | 5.0 | 4.4 | 9.0 | 4.6 | 3.3 |
| Rank | 11° | 1° | 7° | 4° | 13° | 14° | 12° | 6° | 2° | 3° | 10° | 8° | 15° | 9° | 5° |
| Δ with BERT | 25.2 | 11.7 | 7.2 | 6.0 | 18.2 | 19.8 | 16.4 | 1.9 | 3.5 | 3.6 | 11.2 | 7.7 | 5.2 | 4.9 | 6.4 |
| Rank | 1° | 5° | 8° | 10° | 3° | 2° | 4° | 15° | 14° | 13° | 6° | 7° | 11° | 12° | 9° |

TABLE 2

Top: Overall accuracy scores across all tasks. The closer the score is to 100 the more green, the closer it is to random accuracy the more red it is. Bottom: Best performance per task (and rank); Standard deviation of code models (and rank); Delta between best model and BERT (and rank).

| | KTX | IDN | LEN | TYP | REA | JBL | SRI | SRK | SCK | OCU | VCU | CSC | MXN | CPX | NPT |
|-----------|------|------|------|------|------|------|------|-----|-----|-----|------|------|------|-----|-----|
| BERT | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| CodeBERT | 38.7 | 36.3 | 08.0 | 53.5 | 52.4 | 40.7 | 35.0 | | 3.1 | 4.4 | 11.1 | 10.6 | 12.8 | 7.5 | 4.6 |
| CodeBERTa | 42.2 | 16.1 | 11.2 | 37.6 | 30.3 | 4.5 | 8.1 | | 1.0 | | | | | | 0.4 |
| CReviewer | 54.8 | 36.0 | 38.5 | | 31.4 | 2.5 | | | | | | | | | 0.4 |
| CodeT5 | 36.5 | 23.3 | 26.7 | 38.6 | 42.4 | 7.0 | 16.7 | 4.8 | 9.1 | 0.1 | 2.1 | 1.0 | | | 3.5 |
| GCodeBERT | 26.5 | 34.2 | | 59.4 | 44.7 | 37.0 | 36.1 | 5.1 | 1.0 | 4.3 | 14.4 | 11.3 | 9.6 | 3.4 | 9.0 |
| JavaBERT | 14.8 | 28.0 | | | 1.2 | 3.1 | 9.3 | | | | | | | | |
| PLBART | 40.7 | 27.0 | 28.9 | | | | | | | | | | | | |
| UniXCoder | 42.4 | 24.2 | 2.7 | 8.9 | | 3.1 | 25.3 | | | 0.5 | 6.8 | | | | 0.4 |

TABLE 3

Comparison with the BERT baseline (normalized w.r.t the BERT accuracy as the lower limit and 100 as the upper limit for each task; cells are marked grey where accuracy is below the BERT baseline accuracy).

6 RESULTS

In this section, we discuss the probing results, in 3 parts: Task analysis, Model Analysis, and Layer Analysis.

Due to the large number of tasks, we often group them into the categories that they were presented in, in Section 3. The results in this section refer to Tables 2 and 3 below.

6.1 Task analysis

To determine how effective models are against the probes, we first take a look at their collective performance, and pose the following research questions below. This gives us an idea

as to which characteristics for which tasks, are well-encoded in the hidden layers of the models.

- RQ1.1 To what extent can current source code models encode intrinsic code characteristics?
- RQ1.2 To what extent do source code models encode code characteristics *better than* the baseline?

To answer RQ1.1, we look at the maximum model performance for each task, as well as the standard deviation of source code model accuracies for each task.

To answer RQ1.2, we compare the difference between highest accuracy recorded for a task and the BERT baseline accuracy, which is not trained on source code.

6.1.1 Collective performance on tasks

Token-based tasks: For *Token-based* tasks such as KTX (79.2%), IDN (79.5%), as well as LEN (88.5%), we observe good overall performance. In terms of the standard deviation among the source code models, IDN and LEN have low variance (*sigma* ranks 4 and 2), with KTX exhibiting somewhat higher variance (*sigma* rank 11).

Therefore, since the accuracies of the models are generally high, with a generally low variance in scores, this shows that identifier-based syntactic information and token-level surface information are well-encoded in all the models, although information pertaining to keywords is more model-dependent.

Mistyped tasks: For *Mistyped* tasks such as TYP (95.9%), REA (83.5%), and JBL (71.2%), we see considerable variations in performance. For the TYP task, all models perform considerably well, with a low variance across the model scores (*sigma* rank 4). In fact, if we look at the layer-wise performance of each model, almost all layers reflect high scores, indicating this is an easy task; and intrinsic information relevant to identifying *unnatural* misspelled types are well-encoded in the model hidden layers.

For REA even though the best reported score is high (83%) the variance among the model scores is the highest (σ rank 13) For JBL, the variance is slightly higher (σ rank 14), while the performance is somewhat lower (71.2%). This indicates that although some models may perform well or moderately well in term of accuracy, other models may struggle to make good predictions altogether. Indeed, the lowest scores for this task range from 50-60%, which is clearly not impressive considering that the random accuracy for these tasks is 50% (binary classification). This shows us that there is room for improvement for some models in encoding comprehensive semantic information relevant to these tasks.

Replacement tasks: *Replacement* semantic tasks such as SRI (71%), SRK (64.3%), and SCK (65%), report modest prediction accuracies. Moreover, the variance for the best-performing task, SRI, is high (σ rank 12). This indicates that while some of the models may have significant awareness of this type of semantic information, others struggle. In fact, several models have performance below 60% (some close to 50%), which is very low for a binary classification task with a random chance accuracy of 50%.

The more difficult tasks SRK and SCK start to show the limits of current source code models, as it is more challenging for them to distinguish these more difficult cases. While performance is relatively low, the variance is also low (*sigma* ranks 6 and 2), indicating that the models have more comparable performance. Overall, this indicates that there is significant room for improvement for this task category.

Count-based tasks: For *Count-based* tasks such as OCU (24.0%), VCU (33.7%), and CSC (39.5%), we see evidence that source code models struggle with surface-level and structural information. This is particularly for OCU, which is the worst-performing task overall. While the models score significantly better than the naive accuracy (10%), the overall accuracies demonstrate that these are harder tasks. For comparison, we see that KTX has a similar naive accuracy, but the models perform much better. The variance

between models is intermediate (*sigma* ranks: 3, 10, and 8), indicating that some models tend to struggle less than others.

Complexity-based tasks: Finally, the *Complexity-based* tasks show further evidence that source code models struggle with structural information. While MXN has relatively good performance (64.7%), CPX and NPT are on par with the *Count-based* tasks (respectively 39.5% and 35.4%). In terms of variance, the models are very spread out for MXN (the highest of all tasks), and moderate for CPX and NPT (*sigma* ranks 9 and 5). This shows that some models struggle more than others to encode structural information relating to the control-flow in code (particularly regarding indentation).

6.1.2 Comparison with the baseline

Comparing the performance of the models with BERT accentuates the emerging trend in RQ1.1. While for some tasks, the performance increase over BERT is very high (as high as 25%), for others, the source code models offer very little improvements (as low as 2%). In terms of task categories, *Token-based* and *Replacement* tasks are the ones where the largest improvements are seen, although there is variation across tasks. *Replacement* and *Count-based* tasks exhibit moderate improvements as a group (again with some variation), with *Complexity-based* tasks having the smallest improvements as a group.

Given the performance variation as a group, we briefly discuss the tasks with the highest and lowest improvements. The tasks with the largest improvement are KTX, JBL, and REA (tasks related to the semantic of individual keywords and of the method), followed by SRI, IDN, and VCU (all tasks that have to do with identifiers). On the other hand, the tasks with the smallest improvements are SRK, SCK, OCU, CPX, and MXN. These include two of the *Complexity-based* tasks, as well tasks relating to keywords in context, and operators. These tasks are also tasks where the models are struggling in the first place.

6.1.3 Summary

The overall picture that emerges is that source code models can do very well on syntactic token tagging tasks, and tasks that probe on the surface-level token-length, and on a subset of our semantic tasks such as *Mistyped* tasks.

For harder semantic *Replacement* tasks, the models perform moderately when identifiers are involved (SRI), and less well when keywords are involved. This shows that there is room for improvement for comprehensive semantic understanding in models. Furthermore, we observe that there are clear issues with tasks that involve more structural knowledge of source code, that is less easily extracted from a sequence of tokens.

Finally, a further sign that models are struggling is that for the more difficult tasks, the improvements over tend to be small overall.

Source code models tend to perform well on tasks that probe syntactic and some semantic characteristics, but they clearly struggle with tasks that probe more structural characteristics.

| Model | vs. BERT | Layers | Rank 1 | Rank 2 | Rank 3 | < BERT |
|---------------|----------|--------|--------|--------|--------|--------|
| GraphCodeBERT | | | | | | 1 |
| CodeBERT | | | | | — | 1 |
| CodeT5 | | | | | | 2 |
| CodeReviewer | | | | | — | 9 |
| UniXCoder | | | — | | | 6 |
| BERT | — | | — | — | | — |
| PLBART | | | — | | — | 12 |
| CodeBERTa | | | — | — | | 6 |
| JavaBERT | | | — | — | — | 10 |

TABLE 4

Model Performance summary: task performance sparklines, layer performance sparklines, and medal tally

6.2 Model analysis

In this section, we inspect the performance of the models individually across tasks, and task categories. We pose the following research questions as to ascertain which models encode the most information across tasks:

- RQ2.1 Which models are the best in each category?
- RQ2.2 What are the best models overall?

6.2.1 Performance on task categories.

For RQ2.1, we focus on the performance of the models in each task category, and the performance against the BERT baseline. Table 3 shows the performance of the source code models, normalized against the BERT baseline accuracy.

Token-based tasks: For token-based tasks such as KTX and IDN, all source code models comfortably exceed the BERT baseline accuracy. For KTX, CodeReviewer is the best performing model, while, for IDN, CodeBERT and CodeReviewer report the best probing accuracies.

The BERT accuracies for KTX and IDN are the lowest among the other evaluated models, indicating that, as expected, source code models are as a whole much more proficient in learning such code-specific characteristics related to keyword and identifier syntax than BERT. Almost all models outperform BERT on KTX by more than 10%, and all models exceed BERT’s performance on IDN by at least 5%, with several exceeding 10%.

Although BERT does quite well for the LEN task, most code models outperform BERT significantly, suggesting that they have a greater knowledge of surface-level information that are encoded in their hidden layers. Six source code models exceed the BERT baseline by as much as 7% for the LEN task. CodeReviewer is the best performing model, while PLBART and CodeT5 are competitive. Interestingly, two BERT-based source code models are under-performing on LEN (GraphCodeBERT and JavaBERT), while the top three models all have an encoder-decoder architecture, suggesting that the encoding of intrinsic information relevant to LEN may be influenced by model architecture.

We note that for KTX, IDN, and LEN tasks, CodeReviewer is consistently among the best performing models.

Mistyped tasks: For incorrect code tasks, almost all source code models exceed the BERT baseline. In all cases, CodeBERT and GraphCodeBERT vie for the top two positions. CodeT5 is consistently third, with CodeBERTa and CodeReviewer being occasionally competitive. Importantly, while all but one models exceed BERT’s performance for JBL, only CodeBERT and GraphCodeBERT do so with a consequent margin (19.8% and 18%), outlining that this is a more challenging task. On the other hand, five models outperform BERT by margin that exceeds 10% on REA.

Replacement tasks: For the semantic-replacement task SRI, two source code models outperform the BERT-baseline by at least 15%: GraphCodeBERT, CodeBERT. They are followed by UniXCoder and CodeT5, which have also sizeable improvements over BERT. Other models are closer to BERT, if not worse. For both SCK and SRK however, most models perform worse than BERT. Only CodeT5, GraphCodeBERT, and CodeBERT can surpass BERT’s performance, and do so by small margins even in the best of cases (less than 2% for SRK, 3.5% for SCK). Most models are close together (except UniXCoder on SRK). If there is a silver lining, it is that the difference is larger on SCK. Since SCK requires that models find replacements of compatible rather than random keywords, it is a task that requires more source code knowledge than SRK. Indeed, BERT does worse on SCK than SRK, while some models such as CodeT5 or CodeBERT do slightly better. BERT actually ranks *third* on the SRK task (fourth on SCK).

GraphCodeBERT, CodeT5, and CodeBERT are consistently among the best for Mistyped and Replacement tasks.

Count-based tasks: For OCU, VCU, and CSC, most source code models perform worse than BERT. Only two models perform consistently well (GraphCodeBERT and CodeBERT), with UniXCoder a distant third. A silver lining is that on VCU and CSC, GraphCodeBERT and CodeBERT outperform BERT by noticeable to consequent margins (7 to 11%). In particular, GraphCodeBERT does comparatively well on VCU (+11.7% over BERT). One reason for this might be that reasoning about variables is very well aligned with GraphCodeBERT’s original training objective. That said, in absolute terms, GraphCodeBERT’s performance is limited (barely a third of correct predictions).

Complexity-based tasks: For Complexity-based tasks, only two models outperform BERT for CPX and MXN. In both cases, CodeBERT is first, followed by GraphCodeBERT. Thus, while at first glance the performance on MXN appears to be relatively good at 64.7%, most source code models are actually far below that level (ranging from 36 to 57%). Surprisingly, the situation is somewhat better for the more challenging NPT task: more source code models outperform BERT. The top three (GraphCodeBERT, CodeBERT, and CodeT5) do so by a good margin (particularly for GraphCodeBERT). Three more models barely exceeds the baseline’s performance. We were surprised that BERT ranks *third* in two of the three complexity tasks.

While CodeBERT and GraphCodeBERT perform best, no model performs well for Count and Complexity tasks.

6.2.2 Overall model performance

For RQ2.2, we focus on the ranking of models across all tasks, as presented in Table 4. The table summarizes the relative performance amongst the models by counting the number of times the models rank first, second, or third on any given task. We also provide summary of each model’s performance via a sparkline inspired graph [65]: tasks are ordered by categories (*Token-based* in blue, *Mistyped* in green, *Replacement* in cyan, *Count-based* in orange, *Complexity-based* in red). The layers are normalized with the baseline BERT performance as the lower bound and the maximum performance for the task as the upper bound.

We can see that two models clearly stand out: CodeBERT and GraphCodeBERT. They are essentially tied. While CodeBERT ranks second in more tasks (6 versus 5), GraphCodeBERT arrives in the top 3 in 13 out of 15 tasks, compared to 12 for CodeBERT. Both models perform below BERT only once (on LEN for GraphCodeBERT, and on SRK for CodeBERT).

CodeT5 is a clear, but distant third: it is placed in the top 3 models in 8 tasks (SCK, SRK, REA, JBL, CSC, TYP, LEN, and NPT), while performing worse than BERT on only MXN and CPX. CodeReviewer is fourth, ranking in the top 3 on 3 tasks (twice first, on KTX and LEN, and second on IDN), but performs worse than BERT on 9 tasks. UnixCoder follows, placing second on KTX and third on SRI, OCU, and VCU, while faring worse than BERT on 6 tasks.

The three remaining models (PLBART, JavaBERT, and CodeBERTa) all perform *worse overall than BERT*. This leaves BERT as sixth overall, in particular finishing third on *four tasks*: SRK, and two of the three *Complexity-based* tasks (MXN and CPX).

CodeBERT and GraphCodeBERT are by far the best performing model, followed by CodeT5. Surprisingly, three source code models perform worse than the BERT baseline.



6.3 Layer analysis

Studies in the field of NLP have shown for the case of language, that pre-trained models such as BERT tend to represent surface characteristics of language best in early layers, syntactic characteristics in the middle layers, and higher-level semantic characteristics in the later layers [31]. These studies contribute to our scientific understanding of these models, which is why we perform a similar study in the case of source code models. To understand if learning patterns exist across tasks for a given model, or across models for a given task, we analyze the layer-by-layer performance of the models. We aggregate the model scores into a single inline graph which provides a glimpse into the overall task learning. Even though the learning patterns may differ from model to model at an individual-level, the collective performance indicators help us understand where relevant characteristics may be encoded.


For the previous analyses, we always considered the best performing layer for each model. In this section, we inspect the performance of the models at the layer-level. We divide our analysis into two parts, layer analysis by tasks and by models. We pose the following research questions




to determine in which layers learning has taken place most effectively for the probed characteristics.





- RQ3.1 For a given task, which layers show the most effective learning of code characteristics?
- RQ3.2 Is the final layer universally suitable across tasks?






For space and clarity reasons, we simplify the presentation by focusing on trends and patterns, instead of presenting a large amount of numbers corresponding to individual layer-wise accuracies of each model, across fifteen tasks. We present inline graphs inspired by sparklines [65] that show the relative performance of each layer. Detailed layer-wise heatmaps of model accuracies for all tasks are available in Appendix A. For each task and each model, we first rank the layers to homogenize the performance of models and tasks (higher scores are better) and reduce the influence of outliers. The ranks are then averaged over all 12-layer models for a single task RQ3.1 (e.g., LEN: ) , or averaged over all tasks for a single model RQ3.2 (e.g., CodeBERT: ) , before being depicted in sparklines.




6.3.1 Layer performance by task

Although the very last layer often performs well, and better than its immediate predecessors—leading to a “last-layer peak” as shown in e.g., CSC  —, we note that the performance is clearly *not* always best in the latest layers. Overall, we can group tasks based on whether the best performance arises in early, early-middle, middle, or late layers.

Early layers. Three tasks show aggregate best performance in the early layers, in two distinct fashions. First, KTX  clearly peaks at the first layer. Second, LEN  and TYP  have a broad peak in the first half of layers, with clear lower scores in the last layers.

Early-Middle layers. Nine tasks exhibit their highest performance in the middle layers. Of these, four show a comparatively earlier peak, leading to this classification. These are CSC , CPX , NPT  and REA .

Middle layers. Three tasks show best scores in the middle layers, with the last layer usually also performing well. These are VCU , SRK , and MXN . Additionally, OCU  and IDN  show similar, albeit less-pronounced patterns.





Later layers. Finally, three tasks show a trend of almost continuous improvement from the early layers to the final layers. These are JBL , SRI , and SCK .





Relation to task categories. We note that two *Token-based* tasks peak in early layers (KTX, LEN), while IDN, which relies more on identifier semantics, peaks in the middle layers. For *structural* tasks (*Count-based* and *Complexity-based*), three tasks peak in the early-middle layers (CSC, CPX, NPT), and three others in the middle layers (VCU, OCU, and MXN). Finally, the more *semantic* tasks (*Mistyped* and *Replacement*) have one task peaking in early layers (TYP—the easiest), one each in the early-middle (REA) and middle (SRK) layers, and three tasks peaking in the late layers (JBL, SRI, and


SCK). Thus, while we caution against over-interpreting these results, we note a tendency for structural tasks to perform better in the middle layers, and for more semantic tasks to perform better in the later layers, with some tasks that do not fulfill this model.

6.3.2 Layer performance by model

For the models, we also see several patterns of performance across layers.

Middle layers. JavaBERT , GraphCodeBERT , and to a lesser extent CodeBERT  and CodeT5  encode the probed task characteristics in the middle layers. All models are BERT variants. Interestingly, GraphCodeBERT has a very pronounced pattern, with very strong middle layers, while layer 10 is almost always one of the worst performing layers.

Late layers. CodeReviewer , and CodeBERTa  show a progressive learning pattern with the best performance generally originating in the last layer of the model. Similarly, BERT  show a progressive learning pattern, but since the average performance drops in the last layer, the penultimate layer encode the most information. PLBART  clearly performs best in the second half of the layers, but the best performing one is closer to the middle.

In contrast, UniXCoder  shows a learning pattern which is unique, with several individual layers much better than their neighbours (especially layers 2, 8, and 12), with layer 8 performing best overall.

It is a common misnomer that the final layer of the model is the best layer. However, in reality we observe a diversity learning patterns for source code models. Indeed some models progressively improve over the layers with the final layer producing the best results, yet, we also notice some models encoding significant information predominantly in the middle layers, as well as in the first layers.

Source code models show a variety of learning patterns, and do not always perform best in the last layer.

7 DISCUSSION

7.1 Model performance

Overall, both CodeBERT and GraphCodeBERT shows promising results: they are the most consistent models, improving upon the BERT baseline in almost all of the tasks. In fact, they are the only two models to consistently improve over BERT on the *structural* tasks, as well as being the only ones to offer sizeable improvements over BERT in a challenging task such as JBL. GraphCodeBERT underperforms the baseline only on LEN. We hypothesize that since the LEN task does not depend on source code structure, or on the data-flow, the modelling paradigm in GraphCodeBERT extracts the surface-level information relevant to the number of tokens less effectively compared to other models. A deep dive into the task analysis promises to yield more concrete evidence. CodeBERT underperforms on SRK; while we have no clear hypothesis of why that is, we note that this is the least source code specific of the three *Replacement* tasks.

7.2 The competitiveness of BERT

We chose to include BERT as a baseline specifically for its *lack* of source code knowledge. LEN is only task not about source code per se; thus LEN was the only task where we expected BERT to be competitive. Also, our expectations could have been that BERT might do well on tasks that rely on identifiers. Identifiers are predominantly composed of concatenated English words, so BERT might have been able to use its English knowledge. We expected it to underperform on all code-specific tasks, particularly the structural ones, for which source code differs most from English.

In fact, BERT's knowledge of English did not particularly help it for tasks relying on identifiers such as IDN or SRI. However, it showed competitive performance for structural tasks. But this surprising performance on structural tasks is not due to BERT performing well; rather, it is more due to the source code models performing surprisingly *poorly*.

Clearly, all source code models struggle on structural tasks. One possible reason for this is that pre-trained source code models using the Transformer architecture do not fully exploit the structure of source code. Transformers inherit by default positional embeddings, popular for NLP tasks, that emphasize the sequential nature of tokens. While this can be changed, few source code models do it, or adopt a training objective that takes into account the specificities of source code (a point discussed further in Section 7.3).

7.3 Factors influencing performance

We have included several models in this study, but each model varies in several ways from the other models, which prevents us from unambiguously identifying the factors that could influence the results. We nevertheless outline our hypotheses for several such factors.

Influence of the language. One hypothesis could be that models trained on a single language (*Java*-specific) could perform better, since they do not have to dedicate capacity to other programming languages. We find no evidence of this, so far. In fact, monolingual models are among the worse performing models in this study (JavaBERT, PLBART³). But a real test of this would be, for instance, to compare a monolingual GraphCodeBERT with a multilingual one.

Influence of the training data. Unlike the other models, CodeReviewer is trained on diff changes rather than functions. It performs best on KTX, IDN, LEN, and, to a lesser extent, on REA tasks. One could think that these identifier level tasks are closer to its training data. On the other hand, it struggles with tasks that require to reason on an entire function (e.g., CPX, VCU), which are understandably farther from its training data.

Performance of decoders. We initially also wanted to include *decoder-only* models, such as CodeGPT or CodeGen. This would have been especially valuable as the largest models such as Codex or AlphaCode are decoder only models (both are not available for this study: AlphaCode is not accessible, and Codex does not allow us to access

3. The PLBART model checkpoint we have probed is trained specifically on Java on top of the PLBART base model which is multilingual.

model weights). However we found these models to be under-performing. One possible reason could be that decoders function differently than encoders, and thus could be harder to probe. The performance on `LEN` points (albeit weakly) to encoder/decoders behaving differently already: three encoder/decoder outperform the `BERT` baseline by a sizeable margin, while all the encoders are close to `BERT` or underperform it. Of note, `UniXCoder`, which has encoder/decoder capabilities, but that we use in encoder-only mode, does not outperform `BERT`. A second reason could be that these models are trained on a classical language modelling objective as a continuous stream of *files*, rather than encoding single *functions*. As such, they may have a very distinct vision of their input, and not have such a clear boundary of functions, which are purposed as probe samples.

Influence of Model size. It is hard to extrapolate from the limited data. The best performing models (`CodeBERT`, `GraphCodeBERT`) are not particularly large. `CodeT5` is among the largest models, and performed relatively well. `CodeReviewer` is also larger, but performed less well than `CodeT5`, although the different training modality played a role. Similarly, `PLBART` is somewhat larger than `CodeBERT` or `GraphCodeBERT`, and yet is one of the worst performing models. On the other hand, the two other under-performing models, `JavaBERT` and `CodeBERTa`, are on the smaller end. Another aspect is the number of layers: both `CodeBERTa` and `PLBART` have only 6 layers, while the other models have 12. Overall, there is insufficient evidence at this stage to determine how model size would impact performance. A stricter approach with probes that fixes all parameters but varies model size would be necessary to study the influence of model size on encoding of intrinsic code characteristics.

Influence of the training objective. We note that all the models that under-perform on the `BERT` baseline have a training objective that is derived from NLP, but is not source code specific. On the other hand, `GraphCodeBERT`'s training objective forces it to reason about the data flow; `CodeT5` has several identifier-specific pre-training tasks; `UniXCoder` uses AST information while training; while `CodeReviewer` has training objectives related to code changes. The only exception is `CodeBERT`. `CodeBERT` uses a pre-training objective (RTD—Replaced Token Detection) that is not *explicitly designed* for source code. On the other hand, RTD can *implicitly* provide a very source-code specific objective. With RTD, `CodeBERT` learns to discriminate between real source code and fake, but plausible, source code (source code where some tokens are replaced by the output of a bidirectional n-gram model [4]). We also note that `GraphCodeBERT` performs comparatively well on the `VCU` task, which is close to its training objective (reasoning on the data flow of variables). The fact that models that have language-specific objectives (explicitly or implicitly) tend to overperform, while more generic models underperform, constitutes in our view evidence that the training objective is a key factor in the model's performance, although further studies should confirm this. We note that Troshin and Chirkova's study (discussed in Section 7.6) points towards similar conclusions.

7.4 Additional Observations

Performance on structural tasks. We also examine the confusion matrices to better understand the performance of models on the tasks. For the *Count-based* or *Complexity-based* tasks, we note that the models perform much better at finding the class "0" than any other classes. It appears that models can differentiate well between the absence of a phenomenon and its presence, but counting is farther out of reach. The models also tend to over-represent one class with a low count, and another one with a higher count, showing some ability to differentiate between low and high counts.

From 1K to 10K. We also examined the change in performance when training on only 1,000 samples, rather than 10,000. On average, the models improve their accuracy by close to 5% when training on 10,000 samples. While this is good, this also means that the models need a significant number of samples (at least 10K) for effective learning, which means that the representations are not that easy to access. The best performing models improved the most (`GraphCodeBERT`, followed by `UniXCoder` and by `CodeBERT`). Another crucial aspect observed when training the probes with 1,000 and 10,000 samples is that the irregularity (uneven trend) in accuracies across the model layers is stabilized as more samples are provided. This gives a more robust characterization of the model learning for the different intrinsic code characteristics. We have observed this for all tasks in our probing suite.

7.5 Implications

Using NLP-based models as baseline. We were surprised by `BERT`'s performance in some tasks, which provides very valuable context for the performance of the source code models. As such, comparing with a NLP-based baseline that is not specifically trained on code, and that is *not* expected to perform well on source code, may yield unexpected results, and help researchers better frame their results.

INSPECT extension. We defined a suite of probing tasks, but this is by no means final. Our framework allows for addition of new tasks and new languages. Then they can be used to run additional experiments. For instance, releasing similar tasks in other programming languages would allow to test whether multi-lingual model have similar quality of representations in all the languages that they were trained on. Extending `inspect` to another language would not drastically change `inspect`, but significant data gathering and pre-processing might need to take place before this. Additional tasks would allow to probe for representations of additional concepts that our initial suite could not cover, and potentially shine a light on additional shortcomings of source code models. We have in mind several possible tasks that could be probed but did not include as of now (see Appendix B), and we have no doubts the community would have many more. Another way to extend our framework is to support model architectures beyond Transformers. This would allow us to probe for additional models that represent more explicitly the structure of programs in their architecture, such as GNNs [66], models that use path-based representations such as `Code2Seq` [67], or models

that augment path-based representations with precise flow information such as Flow2Vec [68] and ContraFlow [69].

Further research on training objectives. Our results provide evidence that the training objective influences the model’s representation of source code concepts. Moreover, our results show that if models perform well on semantic tasks that can be solved by reasoning at the level of tokens or sequence of tokens, the models struggle with probing tasks that require them to reason about source code structure. This holds even for the best performing models such as GraphCodeBERT. Clearly, this calls for additional investigation into whether training objectives that take the structure of code into account would impact performance, and in which way. For instance, it is possible that such training objective would come with tradeoffs: if GraphCodeBERT is one of the best models in the structural tasks, it is one of the worst models on KTX, for instance. Designing novel training objectives, investigating if such tradeoffs occur, and if they can be mitigated, are all ample avenues for future work. Some particularly interesting training objectives to investigate include ones that model program paths, such as Flow2Vec [68], that leverage contrastive learning such as ContraFlow [69], or approaches that combine multiple models such as Fix-Filter-Fix [70]. However some of these approaches would need INSPECT to be extended.

Transfer to downstream tasks. Finally, our probing task suite investigates only the pre-training stages of source code models. Whether better representations after pre-training translate to increased performance on end tasks, or if fine-tuning is enough should be systematically investigated.

In addition, our task suite could be used to investigate to which extent fine-tuned models conserve their representations of the probed concepts, or if they are subject to “catastrophic forgetting”. Troshin and Chirkova have some initial results that point towards this in their study [45]. Finally, we note that the paradigm for the largest language models such as Codex or AlphaCode is at the moment to forego or de-emphasize fine-tuning and to rely on prompt engineering instead. While we do not have access to the inner weights of these models to probe them, such a use case constitute an additional incentive to probe these models, and to perform research in finding the right pre-training objective for source code models.

Towards a better understanding of source code models. Diagnostic tasks, taken in isolation or together as a whole, can be used to further our scientific understanding of what source models learn. In a context where these models are both notoriously opaque, but, at the same time, taking an increasingly larger place in the world, we think that this is an important endeavor. A very interesting direction for future work is to study the link between diagnostic tasks and practical tasks. Practical tasks are more complex than diagnostic tasks, which probe for very specific characteristics. However, we hope that diagnostic tasks can be used to form hypotheses on the performance of practical tasks (e.g., that performance on a practical task might be subpar on some parts of the data due to a model’s lack of structural understanding). Needless to say, extensive further work in this direction would be needed to confirm that it is practical.

7.6 Relationship to other studies

As mentioned in Section 2, several studies have also analyzed source code models since our first study. We briefly discuss their results in relation with ours.

Comparison with Troshin and Chirkova’s study. We find both some similarities and some differences, which we discuss here. At a high level, we find that the models performed considerably worse on our tasks (e.g. in Algorithm the “worse” task from Troshin and Chirkova, some source code models exceed 75% accuracy). This could be due both to the selection of tasks (we are testing for source code characteristics that are less well represented), or the data selection (starting with more data, we were able to select a more challenging subset, e.g. enforcing that it is balanced).

Since our selection of tasks is different, we can not do a detailed per-task comparison. However, we note that there is a degree of similarity between their Variable Misuse task, and our SRI task. In both tasks, there is a considerable gap between BERT’s performance and the best-performing models. In both tasks, GraphCodeBERT, CodeBERT, and CodeT5 perform well (although CodeT5 performed less well on SRI task, and PLBART performed considerably worse—it seems that our version of the task is more discriminative).

Turning to the high-level conclusions, Troshin and Chirkova observe that “[BERT] performs worse than the models pretrained on code in all tasks except the semantic-related Readability and Algorithm tasks, where all pretrained models perform similarly” (one caveat is the size of the datasets for Algorithm and Readability, 934 and 200 datapoints). We find that we have significantly more tasks where BERT performs comparably or better than some or most pre-trained models; in fact, BERT achieves third place in 4 occasions out of 15 tasks, including all tasks related to code complexity. In only 7 tasks out of 15, we find that the majority of source code models do better than BERT. If both studies point out strengths and weaknesses of source code models, our study has more conservative conclusions.

Troshin and Chirkova also consider whether models pretrained with code-specific objectives perform better. For instance, they find that GraphCodeBERT performs best in their DFG Edge Prediction task, which is similar to its pretraining objective. Likewise, we find that GraphCodeBERT performs best in our VCU task, which bears some similarity to its pretraining objective. On difference is that we observe consistently worse performance for models with basic NLP objectives (some of them not in Troshin and Chirkova’s study), which provides us with additional evidence supporting the conclusion, but from the other side of the spectrum. Overall, the fact that both studies find evidence that code-specific pre-training objectives have a positive impact on probing performance increases our confidence in this conclusion.

Troshin and Chirkova also investigate performance across layers (discussed below). Finally, they also consider model size: they were able to compare two variants of the CodeT5 and PLBART models on their task suite. However, they find mixed evidence of the benefit of model size (it is positive in 6 of 8 tasks for CodeT5, and 4 of 8 tasks for PLBART). This echoes our lack of conclusion on model size.

Layer-wise performance. Several studies have found that structural information was best represented in the middle layers. This is the case of the AST-Probe of Lopez *et al.*, which finds that the representation of the AST is better defined in the middle layers [44]. This find is echoed by the study of Wan *et al.* [41], who also measure the representation of the AST by source code models. Our findings for the structural tasks go in this overall direction, as we find that models tend to perform best with the middle layers for these tasks. However we also encountered several tasks for which early or late layers were the ones where the models were performing best, indicating that not all tasks behave similarly. Troshin and Chirkova also investigate performance across layers; they find that for most tasks, the middle layers (4 to 10) have the highest performance [45]. They also find that some tasks related to variable names and misuse perform best on the last layer. We also find that structural tasks perform best in middle layers, and that semantic tasks tend to perform best in the last layers. The work by Chen *et al.* [42] defines the CAT metric, which performs best in the early layers; however, the CAT score is based on the attention.

Model performance. Looking at the performance of specific models, the study of Lopez [44] compares several models in common with ours. We note that our results go in the same direction: For both CodeBERT and GraphCodeBERT, the AST-Probe metric peaks in earlier layers than CodeT5. The shape of the AST-probe curve for CodeBERTa is also similar to our results. We also note that CodeBERT and GraphCodeBERT perform the best with respect to the AST-Probe, which, again, echoes our results. In Wan’s study [41], GraphCodeBERT outperforms CodeBERT. In terms of CAT-Score [42], GraphCodeBERT is usually ahead, followed by CodeBERT, while UnixCoder struggles to compete with a RoBERTa baseline. In Troshin and Chirkova’s probing tasks, GraphCodeBERT is often, but not always the best performing; CodeT5 is often competitive and performs the best in their “is variable declared” and “algorithm” task; CodeBERT is often competitive as well. They also find that BERT is competitive in the “readability” and “algorithm” tasks. Finally, if somewhat less related, another work observed surprisingly good performance for BERT on source code tasks, when BERT is extended with source-code specific adapters [71].

Overall, our findings contribute to a growing body of evidence about the way source code models represent various source code aspects, and in which layers they do so. Our findings are in general in agreement with other studies. However, our extensive selection of tasks and our comparison with the BERT baseline allows us to highlight the limitations of the current crop of source code language models, as well as suggesting ways forward in terms of new experiments, and potential solutions with new training objectives.

8 LIMITATIONS

In this section, we discuss some of the limitations of our study. These may be addressed in future works as we collectively develop the probing paradigm further, ultimately, to better understand the inner workings of large language

models (LLMs), particularly in our case, large-scale source code language models.

Mono-lingual probes. The majority of the probed models are pre-trained on multiple programming languages, and probing them on a single language (e.g. *Java*) is just a first step. Further probes in more languages should be designed to understand the code aspects learned by the pre-trained models. Our work presents the probing framework and releases the relevant code with which different models can be probed on other languages. The work to gather and pre-process the necessary data in other languages should still be done, however, and is likely to require some effort.

Possible confounding factors. While we did our best to make sure our tasks are reliable, there can be an influence from confounding factors. In particular, we can think of method size as a possible confounding factors for some of the structural tasks, as it is well known that there is often a relationship between lines of code and source code complexity in general [72]. This is why we chose to formulate the task as a classification task, rather than a regression task. We think that the impact of this compounding factor, if present, is limited, since we observe very different performance for LEN compared to the *Complexity-based* and *Count-based* tasks (e.g., PLBART is one of the best performing models for LEN, but one of the weakest on the *Complexity-based* and *Count-based* tasks). Moreover, should this confounding factor be more significant, this would only further highlight that the source code models, by overly relying on length, have even weaker abilities to model source code structure than we previously thought.

Unknown characteristics. We limited this study to fifteen tasks, in an attempt to balance the broadness of characteristics necessary to study source models, and the need to reduce the complexity of the presentation. Nevertheless, code understanding may depend on several source code characteristics not covered in this study; further exploration beyond probing for our current task suite is needed to explore additional characteristics. We present a few possibilities for further probing tasks in Appendix B, as well as some tasks that we excluded to streamline the presentation.

Method-level code. Our tasks all focus on the method as a unit (save for IDN which focuses on single identifier tokens). We note that broader contexts beyond methods are still challenging for all but the largest source code models [73] [54], due to the limited input size of the transformer. This is why we restrict ourselves to methods only. Tasks that encompass a larger context should, in time, be developed.

Focus on source code. Our tasks primarily focus on source code, while many models can handle both source code and natural language. We chose to focus primarily on source code, since probing natural language itself has been extensively investigated [28], [35]. However, additional probing tasks that investigate both source code and natural language would be a worthwhile addition.

Focus on encoders rather than decoders. We probed two decoder-only models, CodeGPT [74] and CodeGen [11]. However, our initial results found that these models were severely under-performing, so we preferred to exclude them from the study as we suspected more factors could be at

play (see Section 7). While the current study is specifically probing for *encoded* code characteristics in the model, probing on vector representations from decoders in the future may yield interesting results. Even though early evidence indicates that vectors from the encoders extract more information than the decoders [45], yet, probes on decoders may be useful, and perhaps more fair for certain types of models.

Model availability. Our study is subject to the availability of suitable models to probe. We can not control which models are trained, nor how they were trained. This necessarily limits our conclusions, as the models that are available may not have all the characteristics that we would like to probe for. For instance, we would have liked to probe more mono-lingual models, but the research community has principally focused on multi-lingual language models. We would have also liked to perform more systematic study of model sizes, but this is also limited by model availability. The fact that each model varies in multiple factors prevents us, at this time, to emit strong conclusions as to which factors influence performance. Likewise, the largest and most successful source code language models (e.g., Codex, AlphaCode) are not available with the necessary degree of access to include them in our study.

9 CONCLUSION

Large-scale pre-trained models for code have been shown to perform spectacularly well on a range of Software Engineering (SE) tasks leading to the release of a number of popular new tools e.g. Tabnine, IntelliCode [14], or Github Copilot [10], among many others. As more of such pre-trained models and derivative tools are introduced to the SE community, it becomes imperative to improve our understanding of their capabilities and weaknesses.

In this paper, we have used the probing paradigm to gain insight into the capabilities of eight state-of-the-art publicly-available pre-trained source code models. We gauged their capabilities on a set of fifteen tasks specifically designed for this study to evaluate a broad set of source code characteristics, including identifiers, structural, and semantic characteristics. We show how probes can help us uncover the strengths and weaknesses of a model, to understand the role played by the individual hidden layers in model performance, to verify the linear extractability of properties, and overall to peek into the “black boxes” that are large-scale pre-trained models.

In summary, we observe that GraphCodeBERT is best performing model across most tasks, encoding more of the syntactic, semantic, and structural information than any other model. While it is hard to isolate all factors unambiguously, we think that the training objective is one of the most important factors that impact performance.

More importantly, we notice that models struggle with structural tasks. Even GraphCodeBERT’s improvement over BERT—a model that should *not* have much source code knowledge—is slim. This suggests that there is room for further research in architecting more advanced source code models that can more effectively leverage source code knowledge. Additionally, we observed a diversity of learning patterns in the model layers, indicating that care must

be taken as to determine which layers encode the most knowledge for specific tasks.

We introduced a probing framework, INSPECT, for the intrinsic evaluation of large-scale pre-trained models of source code. Our probing framework automates the entire probing process and can be used with any model available on the Huggingface model hub. Furthermore, it is also extensible with additional tasks. As future work, we plan to construct probing datasets in multiple languages since the majority of the pre-trained code models are multi-lingual. We also plan to define additional tasks to cover more source code characteristics.

In the long run, such an extensive suite of probing tasks could be used to thoroughly evaluate novel pre-trained source code models, thereby forming a pseudo-benchmark during the development phase, making sure that these models do encode important source code characteristics.

REFERENCES

- [1] A. Karmakar and R. Robbes, “What do pre-trained code models know about code?” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.
- [2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [3] *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019.
- [4] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [5] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Pre-trained contextual embedding of source code,” *arXiv preprint arXiv:2001.00059*, 2020.
- [6] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “Unified pre-training for program understanding and generation,” *arXiv preprint arXiv:2103.06333*, 2021.
- [7] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” *arXiv preprint arXiv:2009.08366*, 2021.
- [8] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” *arXiv preprint arXiv:2203.03850*, 2022.
- [9] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.
- [10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “A conversational paradigm for program synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.
- [12] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, “InCoder: A generative model for code infilling and synthesis,” *arXiv preprint arXiv:2204.05999*, 2022.
- [13] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, “Competition-level code generation with alphacode,” *arXiv preprint arXiv:2203.07814*, 2022.
- [14] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intelli-code compose: Code generation using transformer,” *arXiv preprint arXiv:2005.08025*, 2020.
- [15] N. A. Ernst and G. Bavota, “Ai-driven development is here: Should you worry?” *IEEE Software*, vol. 39, no. 2, pp. 106–110, 2022.

- [16] H. M. Johnson, "Clever hans (the horse of mr. von osten): A contribution to experimental, animal, and human psychology," *The Journal of Philosophy, Psychology and Scientific Methods*, vol. 8, no. 24, pp. 663–666, 1911.
- [17] P. Kavumba, N. Inoue, B. Heinzerling, K. Singh, P. Reiser, and K. Inui, "When choosing plausible alternatives, clever hans can be clever," *arXiv preprint arXiv:1911.00225*, 2019.
- [18] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021, pp. 610–623.
- [19] A. Karmakar, J. A. Prenner, M. D'Ambros, and R. Robbes, "Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation," *arXiv preprint arXiv:2212.02684*, 2022.
- [20] G. Alain and Y. Bengio, "Understanding intermediate layers using linear classifier probes," *arXiv preprint arXiv:1610.01644*, 2018.
- [21] X. Shi, I. Padhi, and K. Knight, "Does string-based neural MT learn source syntax?" in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1526–1534.
- [22] Y. Adi, E. Kermany, Y. Belinkov, O. Lavi, and Y. Goldberg, "Fine-grained analysis of sentence embeddings using auxiliary prediction tasks," *arXiv preprint arXiv:1608.04207*, 2017.
- [23] Y. Belinkov, N. Durrani, F. Dalvi, H. Sajjad, and J. Glass, "What do neural machine translation models learn about morphology?" in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 861–872.
- [24] M. E. Peters, M. Neumann, L. Zettlemoyer, and W. Yih, "Dissecting contextual word embeddings: Architecture and representation," *arXiv preprint arXiv:1808.08949*, 2018.
- [25] I. Tenney, P. Xia, B. Chen, A. Wang, A. Poliak, R. T. McCoy, N. Kim, B. V. Durme, S. R. Bowman, D. Das, and E. Pavlick, "What do you learn from context? probing for sentence structure in contextualized word representations," *arXiv preprint arXiv:1905.06316*, 2019.
- [26] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni, "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)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 2126–2136.
- [27] I. Tenney, P. Xia, B. Chen, A. Wang, A. Poliak, R. T. McCoy, N. Kim, B. V. Durme, S. R. Bowman, D. Das, and E. Pavlick, "What do you learn from context? probing for sentence structure in contextualized word representations," *arXiv preprint arXiv:1905.06316*, 2019.
- [28] Y. Belinkov, "Probing classifiers: Promises, shortcomings, and advances," *Computational Linguistics*, vol. 48, no. 1, pp. 207–219, 2022.
- [29] T. McGrath, A. Kapishnikov, N. Tomašev, A. Pearce, D. Hassabis, B. Kim, U. Paquet, and V. Kramnik, "Acquisition of chess knowledge in alphazero," *arXiv preprint arXiv:2111.09259*, 2021.
- [30] J. Hewitt and P. Liang, "Designing and interpreting probes with control tasks," *arXiv preprint arXiv:1909.03368*, 2019.
- [31] G. Jawahar, B. Sagot, and D. Seddah, "What does BERT learn about the structure of language?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 3651–3657.
- [32] J. Hewitt and C. D. Manning, "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)*, 2019, pp. 4129–4138.
- [33] E. Matthew, "Peters, mark neumann, mohit iyyer, matt gardner, christopher clark, kenton lee, luke zettlemoyer. deep contextualized word representations." in *Proc. of NAACL*, 2018.
- [34] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does bert look at? an analysis of bert's attention," *arXiv preprint arXiv:1906.04341*, 2019.
- [35] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in bertology: What we know about how bert works," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 842–866, 2020.
- [36] J. Z. Forde, C. Lovering, G. Konidaris, E. Pavlick, and M. L. Littman, "Where, when & which concepts does alphazero learn? lessons from the game of hex," in *AAAI Workshop on Reinforcement Learning in Games*, vol. 2, 2022.
- [37] J. Cao, Z. Gan, Y. Cheng, L. Yu, Y.-C. Chen, and J. Liu, "Behind the scene: Revealing the secrets of pre-trained vision-and-language models," in *European Conference on Computer Vision*. Springer, 2020, pp. 565–580.
- [38] M. Schrimpf, I. A. Blank, G. Tuckute, C. Kauf, E. A. Hosseini, N. Kanwisher, J. B. Tenenbaum, and E. Fedorenko, "The neural architecture of language: Integrative modeling converges on predictive processing," *Proceedings of the National Academy of Sciences*, vol. 118, no. 45, p. e2105646118, 2021.
- [39] C. Caucheteux, A. Gramfort, and J.-R. King, "Long-range and hierarchical language predictions in brains and algorithms," *arXiv preprint arXiv:2111.14232*, 2021.
- [40] C. Caucheteux, A. Gramfort, and J. R. King, "Deep language algorithms predict semantic comprehension from brain activity," *Scientific reports*, vol. 12, no. 1, pp. 1–10, 2022.
- [41] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?—a structural analysis of pre-trained language models for source code," *arXiv preprint arXiv:2202.06840*, 2022.
- [42] N. Chen, Q. Sun, R. Zhu, X. Li, X. Lu, and M. Gao, "Cat-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure," *arXiv preprint arXiv:2210.04633*, 2022.
- [43] K. Zhang, G. Li, and Z. Jin, "What does transformer learn about source code?" *arXiv preprint arXiv:2207.08466*, 2022.
- [44] J. A. H. López, M. Weyssow, J. S. Cuadrado, and H. Sahraoui, "Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models," *arXiv preprint arXiv:2206.11719*, 2022.
- [45] S. Troshin and N. Chirkova, "Probing pretrained models of source code," *arXiv preprint arXiv:2202.08975*, 2022.
- [46] M. Paltenghi and M. Pradel, "Thinking like a developer? comparing the attention of humans with neural models of code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 867–879.
- [47] M. Paltenghi, R. Pandita, A. Z. Henley, and A. Ziegler, "Extracting meaningful attention on source code: An empirical study of developer and neural model code exploration," *arXiv preprint arXiv:2210.05506*, 2022.
- [48] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.
- [49] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: Indentation as a proxy for complexity metric," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 133–142.
- [50] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.
- [51] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, pp. 188–200, 1988.
- [52] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.
- [53] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1073–1085.
- [54] A. Karmakar, M. Allamanis, and R. Robbes, "JEMMA: an extensible java dataset for ml4code applications," *Empir. Softw. Eng.*, vol. 28, no. 2, p. 54, 2023.
- [55] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 1–5.
- [56] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," *arXiv:1909.09436 [cs, stat]*, Sep. 2019, arXiv: 1909.09436.
- [57] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Hugging-face's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.
- [58] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

- [59] N. T. De Sousa and W. Hasselbring, "Javabert: Training a transformer-based model for the java programming language," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2021, pp. 90–95.
- [60] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.
- [61] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.
- [62] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
- [63] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.
- [64] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Codereviewer: Pre-training for automating code review activities," *arXiv preprint arXiv:2203.09095*, 2022.
- [65] E. Tufte, "Sparklines: Intense, simple, word-sized graphics," *Beautiful Evidence*, vol. 1, pp. 46–63, 2004.
- [66] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.
- [67] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2019.
- [68] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
- [69] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 519–531.
- [70] H. Hong, J. Zhang, Y. Zhang, Y. Wan, and Y. Sui, "Fix-filterfix: Intuitively connect any models for effective bug fixing," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3495–3504.
- [71] D. Goel, R. Grover, and F. H. Fard, "On the cross-modal transfer from natural language to code through adapter modules," *arXiv preprint arXiv:2204.08653*, 2022.
- [72] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, W. Charles *et al.*, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 03, p. 137, 2009.
- [73] J. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 5050–5067, 2022.
- [74] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.



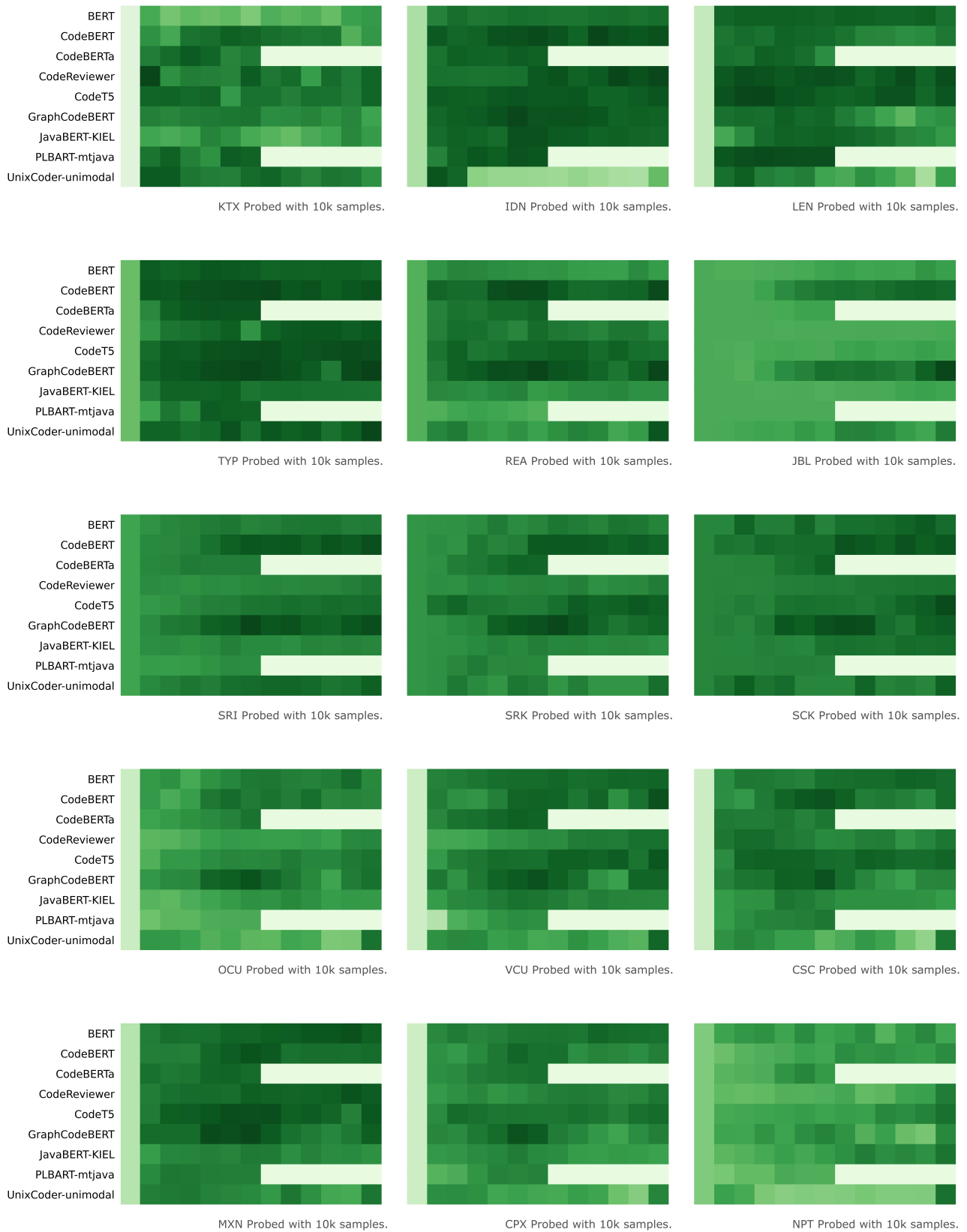
Anjan Karmakar received his Ph.D. and M.Sc. in Computer Science from the Free University of Bozen-Bolzano, Italy. His research interests include applications of machine learning in Software Engineering, and evaluation of Large Language Models (LLMs) of source code.



Romain Robbes is a Senior Scientist (Directeur de Recherche) at the CNRS, working at the LaBRI and hosted by the University of Bordeaux, since February 2023. Before that, he was: an Associate Professor at the Free University of Bozen-Bolzano, (2017–2023); an Assistant, then Associate Professor at the University of Chile (2010–2017); a Ph.D. student, then post-doc, at the University of Lugano (2004–2009). His research interests include Empirical Software Engineering, Mining Software Repositories, Software Maintenance and Evolution, and the intersection of Machine Learning with Programming Languages and Software Engineering.

APPENDIX A

Fig. 2. Heatmaps illustrating layer-wise model performance across tasks



APPENDIX B

CANDIDATE TASKS

We briefly describe tasks that we considered for inclusion, but finally did not include, as well as tasks that we could consider including in the future.

Discarded tasks

During the course of this work, we developed and experimented with several tasks, but the following are some of which did not make the final crop of probes.

AST and TAN. Our previous work [1] included a single token-level AST Node Tagging task. It mixed identifiers with keywords, and did not test for keyword generalization. We replaced it with the current token-level tasks: IDN and KTX. This allows to separately probe keyword and identifier tagging, which are more nuanced than AST node tagging.

As for TAN, it was a binary classification task where models were made to predict the most common AST node type (between identifiers and keywords) in a method. However we found that all models performed well on the task, which led us to de-prioritize it in order to simplify the presentation.

JMB and JFT. We experimented with two variants of JBL. While JBL swaps a single token pair, the variants swapped 50% (JMB) and 100% (JFT) of the tokens. Both variants were too easily solved by the models, indicating that the models are able to detect such drastic code changes.

OCT and VCT. They were variants of OCU and VCU that focused on the total count of operators and variables, rather than the unique count. We chose the unique variant of each as we thought it was less sensitive to the confounding factor of code size, and to reduce the number of tasks; .

NML and NMS They are descendent tasks of CSC where the number of loops (NML) and the number of if structures (NMS) are probed separately in each task. The predictions from the NML and NMS tasks were altogether aligned with the results from the CSC task, therefore, we decided to only include CSC to minimize the number of similar tasks.

Possible Future tasks

We also discussed additional tasks, that we have not included at this time to limit the amount of tasks.

AST level tasks. We thought that characteristics based on the source code's AST, such as AST depth, could be a valuable addition. For the moment, the code complexity tasks cover the most similar concepts. Similarly, we considered classifying methods according to some patterns in the AST structure (e.g., "has nested loops", "has an if in a loop", *etc.*). However, we thought the diversity of possible patterns might be too high, so we used complexity metrics instead.

Variant incorrect-code tasks. We considered a variant of SRI where *all* the occurrences of an identifier would be swapped with a random identifier, further emphasizing sensitivity to context. However we thought such a task might be better suited as a future work. We also considered higher level variants of the "code jumbling" tasks, such as jumbling statements or entire code blocks. These would be extremely interesting, but we thought that they might be too complex for the current breed of models.