AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models

José Antonio Hernández López*
University of Murcia
Murcia, Spain
joseantonio.hernandez6@um.es

Martin Weyssow*
DIRO, University of Montreal
Montreal, Canada
martin.weyssow@umontreal.ca

Jesús Sánchez Cuadrado
University of Murcia
Murcia, Spain
jesusc@um.es

Houari Sahraoui
DIRO, University of Montreal
Montreal, Canada
sahraouh@iro.umontreal.ca

ABSTRACT

The objective of pre-trained language models is to learn contextual representations of textual data. Pre-trained language models have become mainstream in natural language processing and code modeling. Using probes, a technique to study the linguistic properties of hidden vector spaces, previous works have shown that these pre-trained language models encode simple linguistic properties in their hidden representations. However, none of the previous work assessed whether these models encode the whole grammatical structure of a programming language. In this paper, we prove the existence of a *syntactic subspace*, lying in the hidden representations of pre-trained language models, which contain the syntactic information of the programming language. We show that this subspace can be extracted from the models' representations and define a novel probing method, the AST-Probe, that enables recovering the whole abstract syntax tree (AST) of an input code snippet. In our experimentations, we show that this syntactic subspace exists in five state-of-the-art pre-trained language models. In addition, we highlight that the middle layers of the models are the ones that encode most of the AST information. Finally, we estimate the optimal size of this syntactic subspace and show that its dimension is substantially lower than those of the models' representation spaces. This suggests that pre-trained language models use a small part of their representation spaces to encode syntactic information of the programming languages.

KEYWORDS

pre-trained language models, abstract syntax tree, probing, programming languages

*Both authors contributed equally to the paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556900

ACM Reference Format:
José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3551349.3556900

1 INTRODUCTION

The naturalness hypothesis of software [17] states that code is a form of human communication bearing similar statistical properties to those of natural languages. Since this key finding, much work has focused on the application of deep learning (DL) and natural language processing (NLP) to learn semantically meaningful representations of source code artifacts [4]. These representations are vectors that lie in hidden representation spaces of the DL model. These DL-based approaches have drastically improved the automation of a wide range of code-related tasks such as code completion [23, 35], code search [9, 18] or code summarization [8]. In particular, the state-of-the-art leverages pre-trained language model architectures such as those of BERT [13, 22] or GPT [7, 28, 29] to learn representations of source code. Among others, CodeBERT [14], GraphCodeBERT [15], CodeT5 [39] or Codex [10] have shown great performance and flexibility to lots of downstream tasks thanks to task-specific fine-tuning objectives.

A wide range of previous work has shown the great potential of these models in automating many tasks. However, what these models learn in their hidden representation spaces remains an open question. In this paper, we analyze the hidden representation spaces of pre-trained language models to broaden our understanding of these models and their use in automated software engineering. In the NLP field, *probing* emerged as a technique to assess whether hidden representations of neural networks encode specific linguistic properties [2, 5, 16]. A probe is a simple classifier that takes the model's hidden representations as input and predicts a linguistic property of interest. Recently, some previous works relying on probes have shown that the hidden representation spaces of these models encode linguistic properties of the input code and underlying programming language. Among others, Wan et. al [38] showed that unlabeled binary trees can be recovered from the representations of an input code. Even though this property is non-trivial, we foresee that hidden representations of pre-trained language models

encode more complex constructions of the programming languages. In particular, previous contributions have not proposed a probing method that can probe the whole grammatical structure, *i.e.*, the abstract syntax tree (AST), of a programming language in the hidden representation spaces of pre-trained language models. In this work, *we aim to bridge this gap and determine whether pre-trained language models encode the whole ASTs of input code snippets in their hidden representation spaces.*

We propose the *AST-Probe*, a novel probing approach that evaluates whether the AST of an input code snippet is encoded in the hidden representations of a given pre-trained language model. Our approach relies on the hypothesis that there exists a *syntactic subspace*, lying in the hidden representation spaces of the pre-trained language model, that encodes the syntactic information of the input programming language related to the AST. Using our probe, we aim to predict the whole AST of an input code and thus show that pre-trained language models implicitly learn the programming language's syntax. More specifically, the idea is to project the hidden representation of an input code to the syntactic subspace and use the geometry of this subspace to predict the code AST.

We evaluate the AST-Probe on five state-of-the-art pre-trained language models to demonstrate the generalizability of the probe and provide in-depth analysis and discussion. We also attempt to estimate the optimal size of the syntactic subspace, *i.e.*, its compactness, to understand how many dimensions are used by the pre-trained language models to store AST-related syntactic information of the programming languages.

The contributions of this work are the following.

(1) AST-Probe: A language-agnostic probing method able to recover whole ASTs from hidden representations of pre-trained language models to assess their understanding of programming languages syntax.
(2) A representation of ASTs as a compact tuple of vectors adapted from [34] that can be used for probing pre-trained language models.
(3) An extensive evaluation of the AST-Probe approach on a variety of pre-trained language models: CodeBERT [14], GraphCodeBERT [15], CodeT5 [39], CodeBERTa [41] and RoBERTa [22], and programming languages: Python, Javascript and Go.
(4) An estimation of the compactness of the optimal syntactic subspace by comparing different dimensions of possible subspaces.

Organization. Section 2 gives a brief overview of the technical background and discusses related work. In Section 3, we describe the AST-Probe in-depth. In Section 4 and 5, we go through our experimental setup and discuss the results of our experiments. Finally, in Section 6, we discuss the main findings of this work and broader analysis. We close this paper in Section 7 with future work and a conclusion.

2 BACKGROUND AND RELATED WORK

2.1 Pre-trained language models

Traditional NLP models were designed to be task-specific, *i.e.*, trained on specific data, and using specific learning objective functions. In recent years, this paradigm has shifted to pre-training large language models on large text corpora in a self-supervised fashion. These models are pre-trained using relatively general learning objectives to learn representations of the pre-training data. They can then be fine-tuned on a wide range of task-specific data and generally show impressive results [26].

Modern pre-trained language models are based on the Transformer architecture [37]. Pre-trained language models can be roughly classified into three categories [26]: *autoregressive language models, masked language models, and encoder-decoder language models*. Autoregressive language models' objective is to learn to predict the next word in a sequence by expanding a history made of the previous words through time steps (*e.g.*, GPT-3 [7], CodeGPT [24] or Codex [10]). Masked language models are trained with the objective to predict a masked word given the rest of the sequence as context (*e.g.*, BERT [13], CodeBERTa [41], CodeBERT [14] and GraphCodeBERT [15]). Finally, encoder-decoder language models can be trained on a sequence reconstruction or translation objective (*e.g.*, T5 [30], BART [30], CodeT5 [39] and PLBART [3]).

A language model takes a sequence of tokens as input and produces a set of contextualized vector representations. More concretely, let us denote $w_0, \ldots, w_n$ a sequence of $n+1$ tokens. A language model of $\mathcal{L}$ transformer layers consists of the calculus of the final embeddings in the following way:

$$H^{\ell} = \text{Transformer}_{\ell}(H^{\ell-1}), \forall\, \ell = 1, \ldots, \mathcal{L}$$

where $H^{\ell} = [h_0^{\ell}, \ldots, h_n^{\ell}]$, and each vector h_i^{ℓ} corresponds to the embedding of the token w_i. $\text{Transformer}_{\ell}$ is a transformer layer made of multi-head attention layers, residual connections, and a feed-forward neural network. H^0 refers to the initial embeddings computed as a sum of the positional encodings and the initial token embeddings. In this paper, we are interested in the syntactic information encoded in the vectors $h_i^{\ell} \in H^{\ell}$ for $\ell = 1, \ldots, \mathcal{L}$.

Finally, even though we focus on hidden vector representations in this work, it is worth mentioning that the attention layers of these transformer architectures contain valuable information. Several previous works have attempted to discover what information is encoded in the attention heads of transformer-based models of code. For instance, Wan et. al [38] showed that the attention is aligned with the motif structure of the AST, Sharma et. al [33] explored the attention layers of BERT pre-trained on code, and Paltenghi and Pradel [27] compared the attention weights with the reasoning of skilled humans. As we specifically aim at probing deep learning models, we focus the next two sections on this technique and its usage in the related work.

2.2 Probing in NLP

Pre-trained language models are powerful models that have achieved state-of-the-art results not only on NLP but also in code-related tasks. However, one issue that we aim to tackle in this paper is that it is not possible to get a direct understanding of to what extent these models capture specific properties of a language. In this view, *probing classifiers* are used to probe and analyze neural networks. A probe is a simple [1] classifier trained to predict a particular linguistic property from the hidden representations of a neural network [5]. If the probe performs well, it is said that there is evidence that the

[1] The term *simple* usually means minimally parameterised [25].

neural network representations embed the linguistic property. For instance, Belinkov et al. [6] used this framework to perform part-of-speech tagging using the representations of a neural machine translation model. Another example taken from Adi et al. [2] is trying to predict an input sequence length from its hidden vector representation.

In the context of syntax, syntactic probes are used to assess if word representations encode syntactic information of the input language. One example is the *Structural Probe* proposed by Hewitt and Manning [16] that is used to assess if dependency trees are embedded in the hidden representations of language models. The procedure consists of training a linear transformation whose target vector space has the following property: *the squared distance between two transformed word vectors is approximately the distance between these two words in the dependency parse tree*. Once the probe is trained, the dependency tree can be reconstructed using the distance between each pair of words in the linearly-transformed representation space [16, 25, 40].

In this paper, we adopt similar reasoning to assess if pre-trained language models of code encode the AST in their hidden representations. Our probe is also syntactic, but the target tree is of a different type. We consider ASTs, which are more complex constructions than dependency trees as they contain labeled non-terminal nodes that are not seen in the sequence of input tokens. Moreover, our probing approach is different as we consider an orthogonal projection rather than an arbitrary linear transformation. It allows us to thoroughly define the syntactic subspace (see Section 3).

2.3 Probing in pre-trained language models of code

The idea of probing languages models trained on code has been recently introduced, and thus only a few related works have been produced on this research topic over the past couple of years.

Karmakar and Robbes [19] proposed a first method for probing pre-trained language models of code. Through a set of four probing tasks, the authors analyzed the ability of pre-trained language models to capture some simple properties of programming languages. In their work, they concluded that these models implicitly capture some understanding of code in their hidden representations. For instance, they show that using their probes, the cyclomatic complexity of a program or the length of an input code can be predicted from a model's hidden representations. Subsequently, Troshin and Chirkova [36] extended their work by incorporating more models and more probing tasks to broaden the analysis. Finally, Wan et. al [38] designed a more complex probe that can extract unlabeled binary trees from code representations.

Although previous works have probed interesting linguistic properties, they have not focused on designing a probe that can determine if a pre-trained language model captures the full grammatical structure of programming languages. That is, previous works' probing methods do not allow to recover *whole ASTs* (including both terminal and non-terminal nodes). In particular, Troshin and Chirkova [36] assessed whether pre-trained models understand syntax by predicting several properties extracted from the AST such as token depths and paths. Wan et. al [38]'s probe is able to predict unlabeled binary trees at most, which are simplified versions of ASTs that do not encode non-terminal nodes.

In this work, we propose a probing method that allows recovering whole ASTs from pre-trained language models' hidden representations. The key idea of the probe is the assumption of the existence of a syntactic subspace in the hidden representation spaces of these models that encodes the programming languages' syntax. This assumption enables us to estimate the dimension of the subspace and draw conclusions about how compact this information is stored in the pre-trained language models. Thus, our method entails an advance in the state-of-the-art since it substantially improves the latter by assessing the models' understanding of the whole grammatical structure of programming languages.

3 THE AST-PROBE APPROACH

In this paper, we propose AST-Probe, a probe to determine if pre-trained language models implicitly learn the syntax of programming languages in their hidden representation spaces. To this end, we assess if ASTs can be reconstructed using the information from these hidden representation spaces that the models learn from sequences of code tokens.

The AST-Probe looks for a syntactic subspace [2] $\mathcal{S}$ in one of the hidden representation spaces of a model that is likely to contain most of the syntactic information of the language. We obtain the subspace $\mathcal{S}$ by learning an orthogonal projection of the representations of input code sequences to this subspace. Then, using the geometrical properties of $\mathcal{S}$, we show how it is possible to reconstruct the whole AST of an input code snippet. The AST-Probe must operate on real vectors to make predictions. Therefore, we first reduce the ASTs of input codes to compact tuples of vectors. The whole process is summarized in the following graph:

$$\text{AST} \overset{\S 3.1}{\longleftrightarrow} \text{binary tree} \overset{\S 3.2}{\longleftrightarrow} (\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}) \overset{\S 3.3}{\longleftarrow} \text{AST-Probe}$$

Given a code snippet, we extract its AST and convert it into a tuple of vectors $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ that compresses all the AST information. To this end, we adapt the approach proposed by Shen et al [34] for natural language constituency parsing. The conversion is performed in two steps: (1) we transform the AST into a binary tree (Sect. 3.1) and (2) we extract the tuple from the binary tree (Sect. 3.2). Note that these two transformations are bidirectional which allows us to recover the binary tree and the AST from the tuple of vectors. Finally, our AST-Probe aims at predicting the vectors $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ from the syntactic subspace $\mathcal{S}$ and reconstruct the AST of the code snippet (Sect. 3.3). In the following subsections, we go through these processes in detail, explain the nature of the vectors $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ and how they can be used to define a probe that recovers the whole AST from a hidden representation space of a model.

As a running example let us consider the following Python code snippet. Its associated AST is depicted in Fig. 1 and we use it to illustrate our approach.

Code Listing 1: Running example.

```
for element in l:
    if element > 0:
        c+=1
```

[2] We name this vector space the *syntactic subspace*, a term borrowed from [11] that best describes the targeted subspace.

```
    else:
        selected = element
        break
```

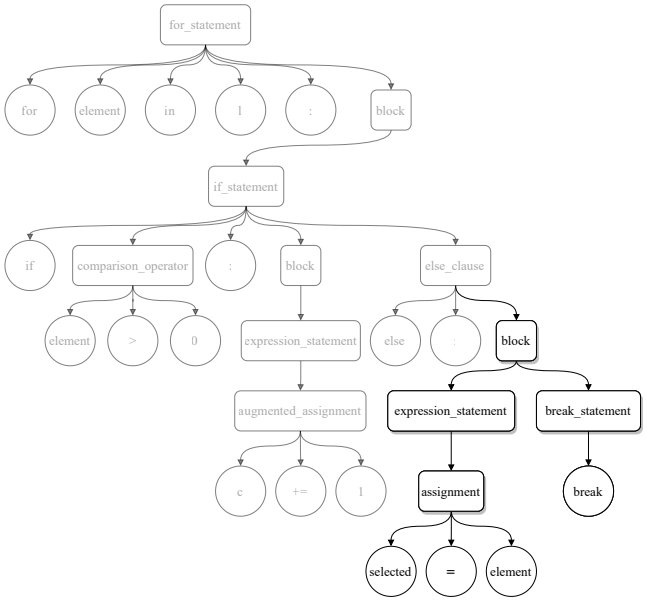

Figure 1: AST of our running example. The rounded rectangles are non-terminal nodes whereas the circles are terminal nodes.

3.1 AST to binary tree

First, to transform an abstract syntax tree into a binary tree, the unary and n−ary nodes need to be binarized. Since the binary tree associated to the full AST of our running example would not fit the paper, we plot the binary tree that corresponds to the *else block* in the non-shaded part of Fig. 2.

Given an AST, we use the following rules taken from [34] to convert it into a *binary parse tree*:

- If a non-terminal node is unary, it is merged with its child. An example of the application of this rule is shown in node ① of Fig. 2. In this example, the non-terminal *expression_statement* is merged with *assignment* to form a new non-terminal node.
- If a node is n−ary, a special node $\emptyset$ is added to binarize it. An example of the application of this rule is shown in node ② of Fig. 2. In this example, a non-terminal node $\emptyset$ is added to binarize the 3-ary non-terminal *assignment* node.
- If a non-terminal chain ends with a unique terminal node then the chain is deleted from the binary tree and its label is added to the terminal node. An example of the application of this rule is shown in node ③ of Fig. 2. The non-terminal *break_statement* is removed and its terminal node is labeled with its label.

Given a binary tree, recovering its corresponding AST is relatively straightforward. The process consists of removing the $\emptyset$ nodes, reconnecting the tree and expanding the unary chains.

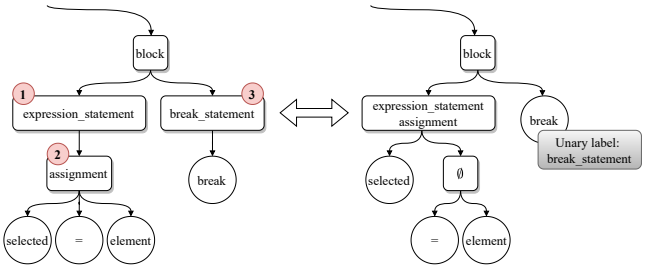

Figure 2: Excerpt of the binary tree associated to the AST of the running example (Fig. 1).

3.2 Binary tree to tuple

In this second step, the binary tree is transformed into the vectors tuple ($\boldsymbol{d}$, $\boldsymbol{c}$, $\boldsymbol{u}$). The vector $\boldsymbol{d}$ encodes the structural information of the AST whereas $\boldsymbol{c}$ and $\boldsymbol{u}$ encode the labeling information.

Encoding vector $\boldsymbol{d}$. To construct $\boldsymbol{d}$, we follow the definition of [34].

Definition 3.1. Given a binary parse tree whose leaves are the words $w_0, \ldots, w_n$. The syntactic distances of this tree is a vector of scalars $\boldsymbol{d} = (d_1, \ldots, d_n)$ such that

$$\text{sign}(d_i - d_j) = \text{sign}(\tilde{d}_i^{i-1} - \tilde{d}_j^{j-1}) \text{ for all } 0 \leq i, j \leq n,$$

where $\tilde{d}_i^j$ is the height of the lowest common ancestor for two leaves (w_i, w_j). This means that a vector $\boldsymbol{d}$ corresponds to the syntactic distances of a given tree if and only if it induces the same ranking order as $(\tilde{d}_1^0, \ldots, \tilde{d}_n^{n-1})$ [3].

Encoding vectors $\boldsymbol{c}$ and $\boldsymbol{u}$. Obtaining the vectors $\boldsymbol{c}$ and $\boldsymbol{u}$ is more straightforward. In particular, $\boldsymbol{c}$ is a vector of the same dimension as $\boldsymbol{d}$ that contains the label of the lowest common ancestor for every two consecutive tokens. Whereas, $\boldsymbol{u}$ is a vector whose dimension is the number of terminals containing the unary labels.

For sake of completeness, we replicate here the algorithms of [34] performing the required adaptations. The procedure to convert a binarized AST into a tuple is shown in Algorithm 1. Whereas the Algorithm 2 describes how to recover the AST from the tuple.

For instance, let us consider the excerpt in Fig. 2 (right). The tree has four leaves thus the size of the vectors $\boldsymbol{d}$, $\boldsymbol{c}$ and $\boldsymbol{u}$ are three, three and four respectively. In particular, the vector $(2, 1, 3)$ corresponds to $(\tilde{d}_1^0, \tilde{d}_2^1, \tilde{d}_3^2)$. Hence, every vector $\boldsymbol{d}$ that verifies the ranking $d_3 > d_1 > d_2$ is valid. On the other hand, the vectors $\boldsymbol{c}$ and $\boldsymbol{u}$ are (expresion_statement-assignment, $\emptyset$, block) and ($\emptyset$, $\emptyset$, $\emptyset$, break_statement) respectively. In the context of $\boldsymbol{u}$, the symbol $\emptyset$ means that the terminal node does not have an unary label.

3.3 Syntactic subspace and probing classifier

Let us denote $\mathcal{M}$ a deep model that receives a sequence l of code tokens $w_0^l, \ldots, w_n^l$ as input and outputs a vector representation for each token $h_0^l, \ldots, h_n^l \in \mathbb{R}^{m_1}$. Additionally, let us assume that the tuple ($\boldsymbol{d}$, $\boldsymbol{c}$, $\boldsymbol{u}$) is a valid tuple associated to the AST of the code sequence l.

Our hypothesis is that there exists a *syntactic subspace* $\mathcal{S} \subseteq \mathbb{R}^{m_1}$ that encodes the AST information (Fig. 3). The objective of

[3] Note that the vector $(\tilde{d}_1^0, \ldots, \tilde{d}_n^{n-1})$ also defines valid syntactic distances.

Algorithm 1 Binary tree to $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ function extracted from [34]

function TREE2TUPLE(node)
  if node is leaf **then**
    $\boldsymbol{d} \leftarrow []$
    $\boldsymbol{c} \leftarrow []$
    $h \leftarrow 0$
    if node has unary_label **then**
      $\boldsymbol{u} \leftarrow$ [node.unary_label]
    else
      $\boldsymbol{u} \leftarrow [\emptyset]$
    end if
  else
    $l, r \leftarrow$ children of node
    $\boldsymbol{d}_l, \boldsymbol{c}_l, \boldsymbol{u}_l, h_l \leftarrow$ TREE2TUPLE(l)
    $\boldsymbol{d}_r, \boldsymbol{c}_r, \boldsymbol{u}_r, h_r \leftarrow$ TREE2TUPLE(r)
    $h \leftarrow \max(h_l, h_r) + 1$
    $\boldsymbol{d} \leftarrow \boldsymbol{d}_l + +[h] + +\boldsymbol{d}_r$ ▷ The operator ++ means *concat*
    $\boldsymbol{c} \leftarrow \boldsymbol{c}_l + +$ [node.c_label] $+ +\boldsymbol{d}_r$
    $\boldsymbol{u} \leftarrow \boldsymbol{u}_l + +\boldsymbol{u}_r$
  end if
  return $\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}, h$
end function

Algorithm 2 $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ to binary tree function extracted from [34]

function TUPLE2TREE$(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$
  if $\boldsymbol{d} == []$ **then**
    node ← Leaf$(\boldsymbol{u}[0])$
  else
    $i \leftarrow \text{argmax}_i(\boldsymbol{d})$
    $\text{child}_l \leftarrow$ TUPLE2TREE$(\boldsymbol{d}_{<i}, \boldsymbol{c}_{<i}, \boldsymbol{u}_{\leq i})$
    $\text{child}_r \leftarrow$ TUPLE2TREE$(\boldsymbol{d}_{>i}, \boldsymbol{c}_{>i}, \boldsymbol{u}_{>i})$
    node ← Node$(\text{child}_l, \text{child}_d, \boldsymbol{c}[i])$
  end if
  return node
end function

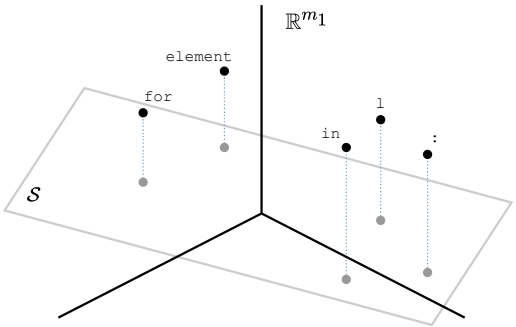

Figure 3: Visualization of the projection. The dotted blue lines represent the projection $P_{\mathcal{S}}$ of the token representations onto the syntactic subspace.

the AST-Probe is to learn a syntactic subspace $\mathcal{S}$ from a hidden representation space of the model $\mathcal{M}$ and assess how well the tuple

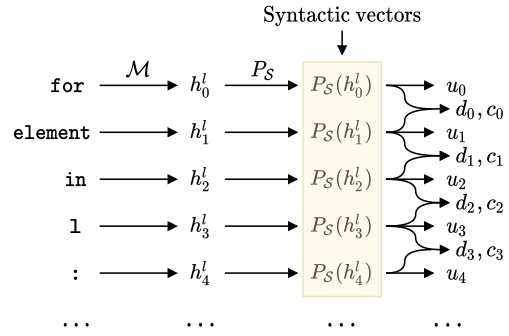

Figure 4: Overview of the AST-Probe. The syntactic vectors are obtained using the projection $P_{\mathcal{S}}$ (see Fig. 3).

$(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$ can be predicted from this subspace. We learn $\mathcal{S}$ using our probe by projecting the representations h_i^l onto $\mathcal{S}$. These projected representations are then used to predict the vectors $\boldsymbol{d}, \boldsymbol{c}$ and $\boldsymbol{u}$ using the geometry inside $\mathcal{S}$, *i.e.*, the dot product.

Knowing the subspace $\mathcal{S}$, each word vector can be decomposed into a sum of two projections:

$$h_i^l = P_{\mathcal{S}}(h_i^l) + P_{\mathcal{S}^\perp}(h_i^l),$$

$P_{\mathcal{S}}(h_i^l)$ contains the syntactic information of h_i^l, whereas $P_{\mathcal{S}^\perp}(h_i^l)$ contains the rest of the original word embedding information. The projection $P_{\mathcal{S}}(h_i^l)$ is what we call the *syntactic vector* of w_i^l.

An overview of the probe is shown in Fig. 4. First, we project the word vectors onto $\mathcal{S}$ to get the syntactic vectors. Then, they are used to infer $\boldsymbol{d}, \boldsymbol{c}$ and $\boldsymbol{u}$ using the square Euclidean distance, a set of vectors C and a set of vectors $\mathcal{U}$.

In the remaining of this section, we go through our probe in more detail. Let us consider B a matrix of dimension $m_2 \times m_1$ whose m_2 rows define an orthonormal basis of $\mathcal{S}$. The vector Bh_i^l corresponds to the coordinates of $P_{\mathcal{S}}(h_i^l)$ with respect to this basis. In the training procedure, we learn B and two sets of vectors $C, \mathcal{U} \subset \mathcal{S}$ forcing the following conditions:

(1) $\hat{d}_i := d\left(P_{\mathcal{S}}(h_{i-1}^l), P_{\mathcal{S}}(h_i^l)\right)^2 = \|Bh_{i-1}^l - Bh_i^l\|_2^2$ is a syntactic distance (see Def. 3.1) for all $1 \leq i \leq n$.
(2) For all $i = 1, \ldots, n$, $P_{\mathcal{S}}(h_{i-1}^l) - P_{\mathcal{S}}(h_i^l)$ is similar to the vector $v_{c_i} \in C$ *i.e.*, $\langle P_{\mathcal{S}}(h_{i-1}^l) - P_{\mathcal{S}}(h_i^l), v_{c_i} \rangle$ is high.
(3) For all $i = 0, \ldots, n$, Bh_i^l is similar to the vector $v_{u_i} \in \mathcal{U}$ *i.e.*, $\langle P_{\mathcal{S}}(h_i^l), v_{u_i} \rangle$ is high.

$|C|$ and $|\mathcal{U}|$ are the number of distinct labels that appear in the vectors $\boldsymbol{c}$ and $\boldsymbol{u}$, respectively. Since $C, \mathcal{U} \subset \mathcal{S}$, we define these vectors with respect to the orthonormal basis induced by B. Therefore, the number of total parameters in the probe is $m_1 \cdot m_2 + m_2(|C| + |\mathcal{U}|) \in O(m_2)$. It is important to note that the complexity of the probe is given by the dimension of $\mathcal{S}$ *i.e.*, m_2. Thus, our probe fits the definition of a simple probing classifier.

In order to achieve optimality of the subspace $\mathcal{S}$, we minimize a loss function composed of the sum of three losses and a regularization term: $\mathcal{L} = \mathcal{L}_{\boldsymbol{d}} + \mathcal{L}_{\boldsymbol{c}} + \mathcal{L}_{\boldsymbol{u}} + \lambda \cdot \text{OR}(B)$. The first loss is a

pair-wise learning-to-rank loss:

$$\mathcal{L}_{\boldsymbol{d}} = \sum_{i,j>i} \text{ReLU}\left(1 - \text{sign}(\tilde{d}_i^{i-1} - \tilde{d}_j^{j-1})(\hat{d}_i - \hat{d}_j)\right).$$

We use this loss since we only need the ranking information of $\boldsymbol{d}$ to recover the AST [34] *i.e.*, we want $\boldsymbol{d}$ to induce the same ranking as $(\tilde{d}_1^0, \ldots, \tilde{d}_n^{n-1})$. The second loss $\mathcal{L}_{\boldsymbol{c}}$ is defined as follows:

$$\mathcal{L}_{\boldsymbol{c}} = -\sum_i \log \frac{\exp\left(\left\langle P_{\mathcal{S}}(h_i^l) - P_{\mathcal{S}}(h_{i+1}^l), v_{c_i}\right\rangle\right)}{\sum_{v \in C} \exp\left(\left\langle P_{\mathcal{S}}(h_i^l) - P_{\mathcal{S}}(h_{i+1}^l), v\right\rangle\right)}.$$

Using the cross-entropy loss together with the softmax function, we ensure that the most similar vector to $P_{\mathcal{S}}(h_i^l) - P_{\mathcal{S}}(h_{i+1}^l)$ in the set C is v_{c_i}. $\mathcal{L}_{\boldsymbol{u}}$ is defined similarly:

$$\mathcal{L}_{\boldsymbol{u}} = -\sum_i \log \frac{\exp\left(\langle P_{\mathcal{S}}(h_i^l), v_{u_i}\rangle\right)}{\sum_{v \in \mathcal{U}} \exp\left(\langle P_{\mathcal{S}}(h_i^l), v\rangle\right)}.$$

Finally, the regualization term is defined as follows [21]:

$$\text{OR}(B) = \|BB^{\text{T}} - I\|_{\text{F}}^2,$$

where $\|\cdot\|_F$ denotes the Frobenius norm and I the identity matrix of dimension m_2. We add this component to the loss in order to force the basis, *i.e.*, the rows of B, to be orthonormal.

4 EXPERIMENTS

In this section, we go through our experimental setup in detail. In particular, we discuss the data and models used in our experiments as well as how we assess the effectiveness of our probe using proper evaluation metrics. To evaluate the relevance of our probe, we articulate our experiments around the following research questions:

– **RQ1**: *Can the AST-Probe learn to parse on top of any informative code representation* [4]*?*

We follow the same procedure as in [16] to check if the AST-Probe is valid. That is, we try to demonstrate that it does not deeply look for information and just exposes what is already present in the hidden representations [25]. To do so, we compare the accuracy of the AST-Probe using pre-trained language models including two baselines: a non-contextualized embedding layer and a randomly initialized language model. If the AST-Probe is valid, a performance gap should lie between the non-baseline models and the baselines.

– **RQ2**: *Which pre-trained language model best encodes the AST in its hidden representations?*

We compare a total of six pre-trained language models for three programming languages and assess which one best encodes the AST of input codes.

– **RQ3**: *What layers of the pre-trained language models encode the AST better?*

We apply our probe to specific hidden representation spaces of intermediate layers of the models and compare the probe effectiveness over the layers.

– **RQ4**: *What is the dimension of the syntactic subspace $\mathcal{S}$?*

To end our experiments, we are interested in how compact is the *syntactic subspace* in the hidden representation spaces of the pre-trained language models.

[4] *any informative representation* is a term used in [16] to refer to representations that do not contain lots of information and are simple.

Data availability. All experimental data and code used in this paper are available at https://doi.org/10.5281/zenodo.7032076.

4.1 Data

We choose CodeSearchNet dataset [18] to conduct our experiments. For all models, we assess their ability to capture the AST of code snippets from three programming languages: Python, Go, and Javascript. For each language, we extract a subset of CodeSearchNet following a ratio of 20000/4000/2000 samples for training, test, and validation, respectively. To extract the AST from the samples, we use the `tree-sitter` compiler tool [5]. We follow then the processes described in Sect. 3.1 and Sect. 3.2 to extract the tuples of vectors $(\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u})$.

4.2 Language models

We compare a broad range of pre-trained language models carefully selected from the state-of-the-art to perform a thorough analysis and bring meaningful discussions to the paper. We summarize the models used in our experiments in Table 1.

The first two rows describe baseline models. CodeBERT-0 refers to an embedding layer initialized with CodeBERT's weights. In this model, the code embeddings are uncontextualized as they are extracted before feeding them to the transformer layers of CodeBERT. In CodeBERTrand, the embedding layer is initialized similarly to CodeBERT-0, but the rest of the layers are randomly initialized. Here, the idea is to randomly contextualize CodeBERT-0. This type of baseline has been used in previous natural language processing related work as strong baselines to evaluate probes [11, 12, 16].

Our main models for comparison are CodeBERT, GraphCodeBERT, CodeT5, CodeBERTa and RoBERTa. Note that for CodeT5, we only consider the encoder layers of the encoder-decoder architecture. Finally, RoBERTa is an optimized version of BERT [13] trained on natural language texts. We include this model in our analysis to check if a model not trained on code is able to capture some understanding of the syntax of programming languages.

4.3 Evaluation metrics

The output of the AST-Probe is a prediction for the vectors $\boldsymbol{d}$, $\boldsymbol{c}$ and $\boldsymbol{u}$. To get a meaningful interpretation of the effectiveness of the models, we compare the predicted AST, *i.e.*, recovered from the vectors $\boldsymbol{d}$, $\boldsymbol{c}$ and $\boldsymbol{u}$, with the ground-truth AST. To this end, we compute the precision, recall and F_1-score over tree constituents [1]. These three metrics are used in the NLP literature to evaluate constituency parsers and are defined as:

$$\text{Prec} = \frac{|\text{ Const. in prediction }| \cap |\text{ Const. in ground-truth }|}{|\text{ Const. in prediction }|}$$

$$\text{Recall} = \frac{|\text{ Const. in prediction }| \cap |\text{ Const. in ground-truth }|}{|\text{ Const. in ground-truth }|}$$

$$F_1 = \frac{2 \cdot \text{Prec} \cdot \text{Recall}}{\text{Prec} + \text{Recall}}$$

In the context of ASTs, we define the constituents as (non-terminal, *scope*) where *scope* indicates the position of the first and last tokens. There is one constituent for each non-terminal node.

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

Table 1: Summary of the models used in our experiments.

Model	Architecture	Number of layers	Training data	Dataset
CodeBERT-0	embedding layer	1	code/doc bimodal	CodeSearchNet [18]
CodeBERTrand	CodeBERT, init. random	12	no training	–
CodeBERT [14]	transformer encoder	12	code/doc bimodal	CodeSearchNet [18]
GraphCodeBERT [15]	transformer encoder	12	code/doc bimodal + data flow	CodeSearchNet [18]
CodeT5 [39]	transformer encoder-decoder	12	code/doc bimodal	CodeSearchNet [18] + BigQuery C/C#
CodeBERTa [41]	transformer encoder	6	code/doc bimodal	CodeSearchNet [18]
RoBERTa [22]	transformer encoder	12	natural language text	BookCorpus [42] + English Wikipedia

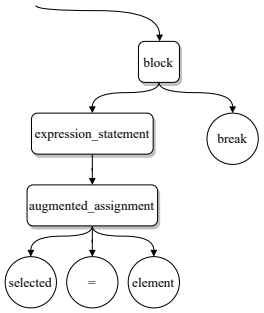

Figure 5: Excerpt of a predicted AST.

For instance, the constituents of the AST of Fig. 2 (left) are the following:

(1) (block, from *selected* to *break*)
(2) (expression_statement, from *selected* to *element*)
(3) (assignment, from *selected* to *element*)
(4) (break_statement, from *break* to *break*)

Let us assume that our predicted AST with respect to that block is the one of the Fig. 5. Thus, its constituents are the following (✓ represents a hit and ✗ a miss with respect to the true AST):

(1) (block, from *selected* to *break*) ✓
(2) (expression_statement, from *selected* to *element*) ✓
(3) (augmented_assignment, from *selected* to *element*) ✗

In that case, the precision and recall are 2/3 and 2/4, respectively.

4.4 Addressing the research questions

To answer RQ1/2/3, we train the AST-Probe for each combination of programming language, model and model layer while fixing the dimension of the syntactic subspace to $m_2 = 128$. And, we report the three evaluation metrics (precision, recall and F_1−score).

To answer RQ4, we select the layer of each model that yields the best F_1−score. We then train several configurations of the AST-Probe by varying the maximum number of dimensions of the syntactic subspace starting from eight up until 512 using powers of two.

4.5 Training details

All the considered models were pre-trained using *byte-pair encoding* (BPE) [32] which constructs the vocabulary of the model over subwords. Since our analysis is performed over whole-word tokens, we assign to each token representation the average of its subword representations [16, 38]. For all models, the dimension of the word embedding space is $m_1 = 768$. Finally, we set the orthogonal regularization term to a high value $\lambda = 5$ in order to ensure orthogonality of the basis induced by the matrix B (see Sect. 3.3).

To perform the optimization, we use the Adam optimizer [20] with an initial learning rate of 0.001. After each epoch, we reduce the learning rate by multiplying it by 0.1 if the validation loss does not decrease. We set the maximum number of epochs to 20 and use early-stopping with a patience of five epochs. A similar configuration of hyperparameters is used in [16] to train the syntax probe.

5 RESULTS

In this section, we report the results of our experiments and answer our four research questions.

5.1 RQ1 – Validity of the AST-Probe

For each model and language, we show in Table 2 the accuracy of the AST-Probe in the layer of the model that obtained the best F_1−score. We can notice a significant performance gap between the baselines and the rest of the models for all considered programming languages and across all metrics. This result validates the probe as it cannot generate ASTs from any informative code representations.

Additionally, we report in Fig. 7 the AST reconstructed by the probe using the representations of CodeBERTrand-10 (best of the baselines) and GraphCodeBERT-4 (best of the non-baseline models) for a Python code snippet. In this example, GraphCodeBERT performs very well, with a perfect precision score, and the predicted AST is very similar to the ground-truth. As for the baseline, we can observe that the predicted AST contains mistakes even though the chosen code snippet is very simple.

Answer to RQ1: We answer negatively to this research question. That is, the performance gap between the baselines and the other models shows that the AST-Probe is not able to learn to parse on top of any informative code representation. The AST-Probe is thus valid.

	Metrics		
Model-BestLayer	Precision	Recall	F_1
CodeBERT-0	0.3262	0.4003	0.3573
CodeBERTrand-10	0.3383	0.4167	0.3710
CodeBERT-5	0.7398	**0.7657**	0.7513
GraphCodeBERT-4	**0.7468**	0.7647	**0.7545**
CodeT5-7	0.6957	0.7097	0.7016
CodeBERTa-4	0.6620	0.6760	0.6679
RoBERTa-5	0.6724	0.6993	0.6841

(a) Python

	Metrics		
Model-BestLayer	Precision	Recall	F_1
CodeBERT-0	0.3358	0.4045	0.3658
CodeBERTrand-11	0.3327	0.4055	0.3642
CodeBERT-5	0.7092	**0.7297**	0.7186
GraphCodeBERT-4	**0.7131**	0.7277	**0.7196**
CodeT5-6	0.6650	0.6775	0.6706
CodeBERTa-5	0.6373	0.6561	0.6459
RoBERTa-8	0.6460	0.6724	0.6580

(b) JavaScript

	Metrics		
Model-BestLayer	Precision	Recall	F_1
CodeBERT-0	0.4337	0.4932	0.4589
CodeBERTrand-11	0.4403	0.5071	0.4692
CodeBERT-5	0.8029	0.8254	0.8134
GraphCodeBERT-5	**0.8135**	**0.8314**	**0.8218**
CodeT5-8	0.7710	0.7889	0.7792
CodeBERTa-4	0.7762	0.7944	0.7846
RoBERTa-5	0.7513	0.7819	0.7653

(c) Go

Table 2: Results of the AST-Probe for each model and language. We report the best layer of each model together with the baselines. For each metric, the highest value is in bold and the second highest is underlined.

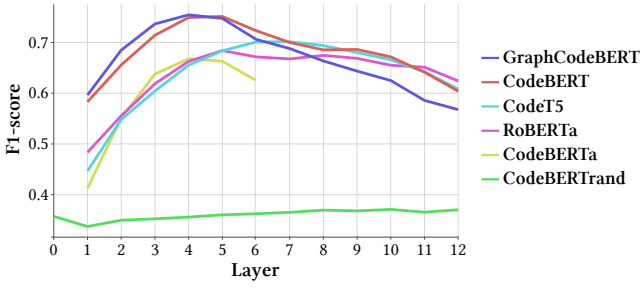

Figure 6: Result of the probe for each model according to their layers. The x–axis represents the layer number and the y–axis the F_1–score. The CodeBERTrand's layer 0 corresponds to CodeBERT-0.

5.2 RQ2 – Best model

Overall, the best models are CodeBERT and GraphCodeBERT for all programming languages in terms of F_1–score. As the difference in F_1–score between both models is very marginal, *e.g.*, +0.0032 F_1 in Python, +0.0010 F_1 in JavaScript, and +0.0084 F_1 in Go in favor of GraphCodeBERT, it is not possible to firmly conclude on which model is better than the other.

One interesting finding related to this RQ is the fact that, even though trained on text data, RoBERTa competes with models trained on code and is able to understand syntactic information of code. For all languages, RoBERTa achieves similar results with CodeT5. And, in the case of Python and JavaScript, it outperforms CodeBERTa.

Answer to RQ2: Among all the considered models, both CodeBERT and GraphCodeBERT best capture the AST in their hidden representations for all programming languages.

5.3 RQ3 – Best layers

Table 2 shows that for RoBERTa, the 5th and 8th layers are the best in terms of F_1–score. For CodeT5, the best F_1–score lies in the 6-7-8th layers. For CodeBERT, GraphCodeBERT, and CodeBERTa, the best F_1–score lies in the 4-5th layers. In Fig. 6, we plot the accuracy of the AST-Probe in terms of F_1–score for all models with respect to their hidden layer for Python [6]. For all the models, we can observe a peak in F_1–score in the middle layers.

Answer to RQ3: For all models, the AST information is more encoded in their middle layers' representations.

5.4 RQ4 – Estimating the dimension of $\mathcal{S}$

In Fig. 8, we plot the F_1–scores for each model by varying the dimension of the syntactic subspace for Python [7]. All these curves have a bell shape due to two reasons: **(1)** when m_2 is too small, the number of dimensions is not enough to encode the AST, and **(2)** when m_2 tends to m_1, and due to the orthogonality constraint, the projection $P_{\mathcal{S}}$ tends to produce a rotation of the initial representation space. Thus, if $m_2 \rightarrow m_1$ then $\langle Bx, By\rangle \rightarrow \langle x, y\rangle$ meaning that we are using the full 768-dimensional word vectors, *i.e.*, without preprocessing through a projection, when predicting the AST. It yields a bad performance because the full vectors encode lots of information not only related to syntax. And, we are only interested in the part of the representation space that encodes the syntax, which is ultimately extracted by the orthogonal projection.

The representation space has 768 dimensions, and the dimension of the syntactic subspace ranges between 64 and 128. It holds across all languages and all models. Only a small part of the representation space is used to encode the AST. It means that the pre-trained language models do not use too many resources to understand the language's syntax as the syntactic information is compactly stored in a relatively low-dimensional subspace.

[6] The shapes of the curves are similar for JavaScript and Go.
[7] The shapes of the curves are similar for JavaScript and Go.

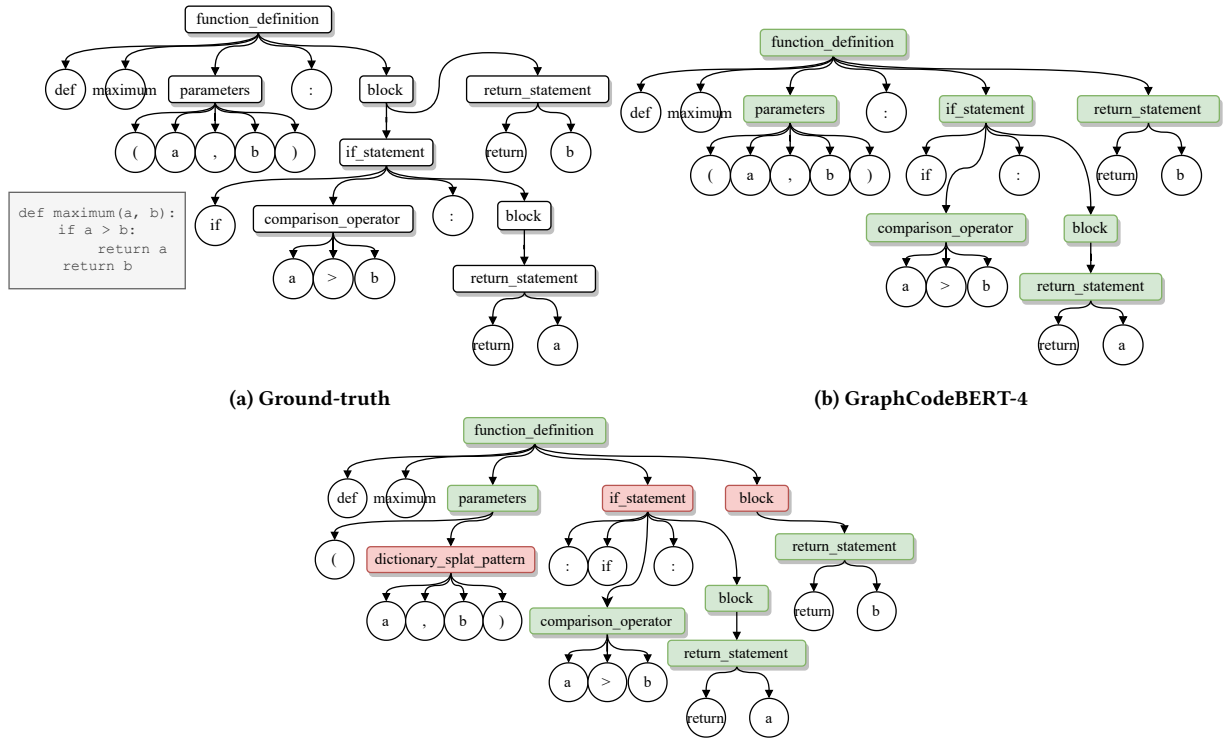

Figure 7: Recovered AST using the probe from the hidden representations of GraphCodeBERT-4 and CodeBERTrand-10. Green rounded rectangles represent constituent hits with respect to the ground-truth AST, and red ones represent misses.

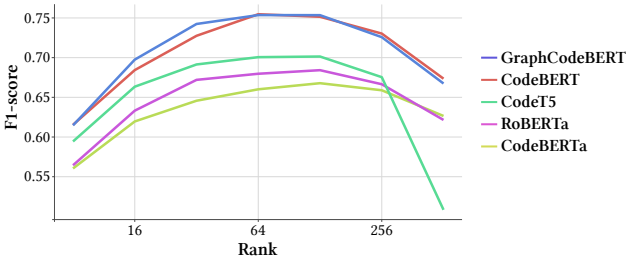

Figure 8: Result of the probe for each model and their best layer according to the dimension of the syntactic subspace for Python. The $x-$axis represents the dimension of the subspace and the $y-$axis the F_1-score.

Answer to RQ4: The dimension of the syntactic subspace ranges between 64 and 128.

6 DISCUSSION

In this work, we have successfully shown the existence of a *syntactic subspace* in the hidden representations of a set of five pre-trained language models that encode the information related to the code's AST. From these models, GraphCodeBERT and CodeBERT are the ones that best encode this syntactic information of the languages. Furthermore, we show that all the probed models mostly store this information in their middle layers. Finally, our last experiment highlights that the models store the information in a syntactic subspace of their hidden representation spaces with a dimension ranging between 64 to 128.

Our conclusion concerning the best layers is aligned with the results obtained by Troshin and Chirkova [36] and Wan et. al [38], where they apply different syntactic probes. In NLP, BERT encodes the syntactic knowledge of natural languages in its middle layers [31], which lines up with our conclusion for programming languages. An interesting future would be to continue developing probes to search for more properties intricated in the models' representation spaces. Then, try to determine what type of information the models learn in each of their hidden layers.

In their work, Wan et. al [38] proposed a structural probe also based on the notion of syntactic distance (see Def. 3.1) that can recover unlabeled binary trees from hidden representations and attention layers. Our probe tackles this limitation of their work by recovering full ASTs from hidden representations which is a more complex structure closer to the syntax of the language than unlabeled binary trees.

Furthermore, we claim that Wan et. al [38]'s probe presents two main issues. Firstly, they compare abstract syntax trees (ground-truth) with unlabeled binary trees (predicted using their probe).

The metric they use to compare both trees is difficult to interpret as they are not comparing trees of the same nature. The metric also contains a random component. We believe that our approach and experiments cope with this limitation as we compare ground-truth ASTs with ASTs induced using the AST-Probe. Furthermore, we use a metric widely used in the NLP literature to evaluate constituency parsers.

The second issue relates to the fact that they extract the unlabeled binary trees from the whole hidden representation spaces, *i.e.*, they take all their dimensions into account. Our analysis of the dimension of the *syntactic subspace* shows that few dimensions of the hidden representation spaces are used by the models to encode syntax. In their work, the authors report that unlabeled binary trees extracted from the attention information produce better results than those extracted from hidden representations. We claim that this can be explained by the fact that they extract unlabeled binary trees from the whole hidden representation spaces of the model. This approach may yield poor results as these representation spaces contain lots of information that are not only related to syntax.

In fact, in our last experiment, we conclude that the syntactic information is stored compactly in a low-dimensional space with a size ranging between 64 and 128. This means that between 8.3% (64/768) and 16.7% (128/768) of the dimensions of the full representation space are used to store AST-related information of the code. To the best of our knowledge, this is the first work that carries out this type of analysis. This finding raises questions about the nature of the information encoded in the rest of the dimensions which is an exciting direction for future work.

In this work, we report theoretical findings of the hidden representations of pre-trained language models. We envision that these findings can have broader practical implications. For instance, it seems that GraphCodeBERT recovers ASTs marginally better than CodeBERT. This is correlated with the fact that GraphCodeBERT performs better than CodeBERT in several downstream tasks (*e.g.,* code search, code translation, etc) [15]. Thus, a good research direction could be to highlight correlations between how well a model encodes syntax and its performance on downstream tasks. Such a practical finding could help us understand what pre-trained language models implicitly learn to perform well in practice. We aim to tackle this interesting question in future work.

In our experimentations, we measure the performance of recovering the full AST through the F_1 score. One possible related research direction would be to analyze which parts of the tree the probe fails to predict using metrics over constituents. It would provide hints on where the pre-trained models struggle to capture syntactical information.

Finally, our work presents several limitations that we plan to tackle in the future. First, we only consider pre-trained language models based on the transformer architecture. Nonetheless, our probe is agnostic of the architecture of the models and programming languages. It can be easily adapted to any type of model involving representation spaces. Then, we choose three common programming languages to analyze. The capacity of our probe to cover a programming language depends on the possibility of extracting the vectors $\boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}$ from the input code. However, the probe is also language agnostic as it only requires a definition of the language's grammar.

7 CONCLUSION AND FUTURE WORK

In this work, we presented a novel probing method, the *AST-Probe*, that recovers whole ASTs from hidden representation spaces of pre-trained language models. The AST-Probe learns a *syntactic subspace* $\mathcal{S}$ that encodes AST-related syntactic information of the code. Then, using the geometry of $\mathcal{S}$, we show how the AST-Probe reconstructs whole ASTs of input code snippets. We apply the AST-Probe to several models and show that the syntactic subspace exists in five pre-trained language models. In addition, we show that these models mostly encode this information in their middle layers. Finally, we estimated the compactness of the syntactic subspace and concluded that pre-trained language models use few dimensions of their hidden representation spaces to encode AST-related information.

In future work, we plan to extend our experiments by including more programming languages. We also want to investigate more pre-trained language models, and different neural network architectures. Furthermore, we plan to study whether a correlation exists between how well a model understands the programming languages' syntax and its performance on code-related tasks. Finally, we intend to compare syntactic subspaces before and after fine-tuning pre-trained language models, for instance, to assess if the models forget about the syntax.

ACKNOWLEDGMENTS

José Antonio Hernández López is supported by a FPU grant funded by the Universidad de Murcia. Martin Weyssow is supported by a Fonds de recherche du Québec - Nature et technologies (FRQNT) PBEEE scholarship (award #320621). We thank Dr. Istvan David and Dr. Bentley Oakes for their valuable feedback and comments that helped greatly improve the manuscript.

REFERENCES

[1] Steven Abney, S Flickenger, Claudia Gdaniec, C Grishman, Philip Harrison, Donald Hindle, Robert Ingria, Frederick Jelinek, Judith Klavans, Mark Liberman, et al. 1991. Procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the workshop on Speech and Natural Language*. 306–311.

[2] Yossi Adi, Einat Kermany, Yonatan Belinkov, Ofer Lavi, and Yoav Goldberg. 2016. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. *arXiv preprint arXiv:1608.04207* (2016).

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

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

[5] Yonatan Belinkov. 2016. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics* (2016), 1–12.

[6] Yonatan Belinkov, Nadir Durrani, Fahim Dalvi, Hassan Sajjad, and James Glass. 2017. What do neural machine translation models learn about morphology? *arXiv preprint arXiv:1704.03471* (2017).

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[8] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[9] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

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

[11] Ethan A Chi, John Hewitt, and Christopher D Manning. 2020. Finding universal grammatical relations in multilingual BERT. *arXiv preprint arXiv:2005.04511* (2020).

[12] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. *arXiv preprint arXiv:1805.01070* (2018).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[16] John Hewitt and Christopher D Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4129–4138.

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

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

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

[20] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[21] Tomasz Limisiewicz and David Mareček. 2020. Introducing orthogonal constraint in structural probes. *arXiv preprint arXiv:2012.15228* (2020).

[22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[23] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. *arXiv preprint arXiv:2203.07722* (2022).

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[25] Rowan Hall Maudslay, Josef Valvoda, Tiago Pimentel, Adina Williams, and Ryan Cotterell. 2020. A tale of a probe and a parser. *arXiv preprint arXiv:2005.01641* (2020).

[26] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heinz, and Dan Roth. 2021. Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey. *arXiv preprint arXiv:2111.01243* (2021).

[27] Matteo Paltenghi and Michael Pradel. 2021. 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, 867–879.

[28] A. Radford. 2018. Improving Language Understanding by Generative Pre-Training.

[29] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[31] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics* 8 (2020), 842–866.

[32] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[33] Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. 2022. An Exploratory Study on Code Attention in BERT. *arXiv preprint arXiv:2204.10200* (2022).

[34] Yikang Shen, Zhouhan Lin, Athul Paul Jacob, Alessandro Sordoni, Aaron Courville, and Yoshua Bengio. 2018. Straight to the tree: Constituency parsing with neural syntactic distance. *arXiv preprint arXiv:1806.04168* (2018).

[35] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.

[36] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Code. *arXiv preprint arXiv:2202.08975* (2022).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[38] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture?–A Structural Analysis of Pre-Trained Language Models for Source Code. *arXiv preprint arXiv:2202.06840* (2022).

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[40] Jennifer C White, Tiago Pimentel, Naomi Saphra, and Ryan Cotterell. 2021. A Non-Linear Structural Probe. *arXiv preprint arXiv:2105.10185* (2021).

[41] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[42] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*. 19–27.