Cross-lingual Performance of CodeGPT on the Code Completion Task

Nadine Kuo[1]

Supervisor(s): Arie van Deursen[1]**, Maliheh Izadi, Jonathan Katzy**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

The development of contemporary source code auto-completion tools have significantly boosted productivity and efficiency of developers. In 2021, the GPT-2-based Transformer CodeGPT was developed to support code completion and text-to-code generation. Similarly to most code models however, CodeGPT was trained on a limited set of widely-used languages (Java, Python) - leading to constrained efficacy in lower-resource languages. This motivated us to research CodeGPT's performance on the token-level code completion task across *high- and low-resource* languages. We investigate in which scenarios CodeGPT predicts incorrect tokens with high certainty using a *tuned lens*, followed by studying attention patterns that underlie the observed behaviour. Our findings indicate that CodeGPT is most competent in Java and Python code (Top-1 accuracies: 69.2% and 68.2% respectively). It generates false predictions with highest confidence when it encounters unfamiliar constructs in low-resource languages, or code structures that cannot be predicted from left context only. Moreover, we find a positive correlation between null attention and model confidence.

Keywords

Automatic Code Completion, Large Language Models, Pre-Trained Language Models, Transformers, Multi-headed Self-Attention

1 Introduction

Ever since the first Transformer-based LLMs GPT [1] and BERT [2] were introduced in 2018, multiple efforts have been made to train language models on code to support programming tasks. Earlier works have indeed demonstrated that the Transformer's self-attention mechanism - allowing for learning valuable contextual relationships across input tokens - is effective in the context of source code [3, 4]. In fact, the incorporation of code completion tools such as *GitHub Copilot*[1] within IDEs and online editors has led to a substantial boost in productivity and efficiency of developers [5].

Multiple studies have attempted to explain *why* these code models are successful. By structurally analyzing CodeBERT [2] and GraphCodeBERT [6], Wan et al. [3] revealed that the syntax structure of code is preserved in the hidden model layers. Furthermore, Chen et al. [4] invented CAT-probing to demonstrate that attention scores relate to distances between AST nodes. Despite the state-of-the-art performance exhibited by LLMs across a diverse range of code intelligence tasks [7], it is crucial to acknowledge that these models are predominantly trained on limited sets of widely-used programming languages. Consequently, their efficacy may be constrained when applied to lesser-known languages - such as Objective-C and Julia. However, none of the earlier works have shed light onto performance of code models on low-resource languages.

[1]https://github.com/features/copilot

Hence, in this work we aim to analyze the *cross-lingual* performance of the GPT-2-based model CodeGPT[2] - whilst performing the single-token code completion task on both high-resource languages (Java, Python, C++) *and* low-resource languages (Kotlin, Go, Julia). By identifying cross-lingual disparities, we aspire to establish a foundation of knowledge upon which to construct models for low-resource languages in the future.

The main research question this paper aims to answer is: **How does CodeGPT perform on the token-level code completion task across multiple languages?**

To guide this research, two sub-questions will be addressed:

1. **In what scenarios does CodeGPT predict incorrect tokens with high confidence?**
2. **Are there patterns in CodeGPT's attention mechanism across different languages?**

The main contributions of this work can be summarized as:

- A cross-lingual investigation of weak areas of CodeGPT during code completion, demonstrating it generates false predictions with high confidence (**RQ1**) when encountering unfamiliar constructs or those that cannot be predicted from left context only. (Sections 3.3, 5.1)
- An attention investigation, revealing a positive correlation between null attention and model confidence (**RQ2**). (Sections 3.4, 5.2)
- An evaluation of CodeGPT's cross-lingual performance on the code completion task (**main RQ**), showing it performs most accurately and confidently for Java, thereby achieving Top-1 accuracy 69.2% and MRR 16.9% (Section 6).

2 Background and Related Work

In this section, we outline background work within the area of code completion and Transformer-based LLMs for code intelligence tasks.

2.1 Automatic Code Completion

Automatic code completion, or auto-completion, refers to the task of predicting subsequent tokens based on an input code sequence, serving as the context for given language model [8]. Various types of predictions exist:

- Token-level predictions (TLP): single tokens of code
- Statement-level predictions (SLP): lines of code
- Block-level predictions (BLP): methods or functions

In the scope of this two-months research project, token-level code completion will be investigated.

More generally, code completion falls under the umbrella of *code generation* - along with code repair and code translation [2]. Another growing category of code intelligence is *code understanding*, which includes tasks such as code search and code clone detection.

[2]Note that we have used a multilingual version of CodeGPT trained on Java, Python, JavaScipt, PHP, Go and Ruby.

In 2019, Svyatkovskiy et al. introduced *Pythia* [9] - a LSTM-based code completion tool generating ranked suggestions for methods and APIs in Python. Later, the Transformer-based *IntelliCode Compose* [10] was presented as a general-purpose, multilingual version capable of predicting entire sequences of arbitrary code tokens. It had been trained on the widely-used languages JavaScipt, TypeScript, C# and Python.

In June 2021, *GitHub Copilot*[3] was made publicly available as code completion tool developed by OpenAI and GitHub. It was built on top of Codex [5] - a descendant of Transformer GPT-3 that was mostly trained on Python code. Other successful, commercial code completion tools are *Tabnine*[4] and *ChatGPT*[5].

All of the above tools have been integrated to VSCode IDE extensions to enhance developer productivity. Yet, the fact that they have been predominantly trained on popular languages may lead to limited performance on lesser-known languages.

2.2 Transformers: from Natural Language to Code

The Transformer [11] is a widely-researched deep learning model architecture that has repeatedly achieved promising results across a range of NLP tasks [12, 13]. It fully relies on its self-attention mechanism to attend to different parts of the input sequence and thus raise more apt predictions. In contrast to the existing RNNs and LSTMs, Transformers have the ability to process full input sequences by exploiting parallel computation. Moreover, they introduced an alternative to recurrence: positional embeddings can now be learned to encode information specific to a token position.

The potential of Transformers for unsupervised, multitask learning became evident when OpenAI introduced the LLM GPT-2 [14], which outperformed other language models on a diverse range of downstream NLP tasks in a zero-shot setting. This adaptability inspired researchers to train language models on large-scale code corpora, leading to the support for software engineering tasks such as code summarization [15], code search [16] or code completion [17]. Through transfer learning, general models were fine-tuned to predict future tokens given some input code sequence - also referred to as causal language modelling.

For instance, BERT (Bidirectional Encoder Representations from Transformers) [13] and its descendant RoBERTa [18] demonstrated the power of exploiting left and right context whilst performing NLP tasks such as question answering. In fact, both LLMs achieved state-of-the-art performance on the *GLUE* and *SQuAD* test benches. Inspired by their success, CuBERT [19] - pre-trained on 7.4M Python source files - was presented as BERT-like model for code understanding tasks. Later, Feng et al. presented CodeBERT [20], which supported natural language code search tasks as well as code completion.

GPT (Generative Pre-trained Transformer) [1] diverged from the BERT-like architecture by adopting a unidirectional, decoder-only design which suffices for code completion - where the goal is to predict the next token or sequence given the *preceding* context only. The effectiveness of this architecture has been demonstrated by various GPT-based code models including CodeFill [21] and CodeGPT [7], both having achieved superior results on token- and statement-level code completion tasks.

2.3 Transformers for Code: Analyzing their Inner Mechanisms

Given the widespread technological advances invoked by Transformer-based LLMs, more recent studies have focused on interpreting their inner mechanisms.

In an attempt to find out *why* code models are successful, Wan et al. [3] performed structural analysis on CodeBERT [2] and GraphCodeBERT [6] - thereby using Python, Java and PHP as target languages. They discovered that self-attention weights align with the syntax structure of code, implying that they capture the *motif structure* in code ASTs. This was especially apparent in the deeper layers of the model. In addition, there exists position-based heads that do not take into account context. Through structural probing they also demonstrated that the syntax structure of code is preserved in the contextual word embedding vectors learned across hidden layers. In fact, these pre-trained code models are capable of inducing syntax trees of code without training, to a certain extent.

Given the lack of quantitative measures on the performance of models on code representation learning in previous studies, Chen et al. [4] proposed a novel technique called CAT-probing. Specifically, they measured how attention scores relate to distances between AST nodes whilst performing code summarization in Python, Java, JavaScript and Go. After having conducted layer-wise probing on RoBERTa [18], CodeBERT [2], GraphCodeBERT [6] and UniXCoder [22], they figured that the specific token types code models focus on vary across languages. In fact, attention distribution on Python code differed from other languages, due to Python having fewer token types.

Furthermore, they discovered that the ability to capture code structure dramatically differs with layers, as attention heads start focusing on special tokens in deeper layers. They also noted that the middle layers may be contributing significantly to the transfer of general structural knowledge into task-specific structural knowledge. These findings were confirmed by Vig et al. [23], whom investigated the structure of attention in GPT-2 and found that attention aligns with dependency relations most strongly in the middle layers - where attention distances are the lowest.

Clearly, there has been limited research on the performance and internal workings of code models when applied to low-resource languages. Hence, we aim to fill this gap in knowledge by examining the *cross-lingual* performance of CodeGPT on the token-level code completion task.

[3]https://github.com/features/copilot
[4]https://www.tabnine.com/
[5]https://openai.com/chatgpt

Model parameters	**No. or size**
No. of layers	12
Max. context length	1,024
Embedding size	768
No. of attention heads	12
Attention head size	64
Vocabulary size	50,261
No. of parameters	124M

Table 1: Model parameters of CodeGPT (GPT-2-based).

3 Approach

To investigate how CodeGPT performs on the code completion task across different programming languages, a *tuned lens* investigation (Section 3.3) followed by an attention investigation (Section 3.4) will be conducted. Figure 2 depicts the overall experiment pipeline, which is elaborated on after Section 3.1 has presented the multilingual CodeGPT model assessed in this work.

3.1 CodeGPT

CodeGPT, a GPT-2-based [14] Transformer model released in 2021 by Microsoft for research on source code generation, supports code completion and text-to-code generation[6] tasks. Inherently, it shares natural language understanding ability and model parameters with GPT-2 as displayed in Table 1.

Given an input sequence (see Figure 1), the model generates predictions auto-regressively, i.e. one token at a time. Before input is fed to its stack of decoder blocks $h_0 - h_{11}$, positional embedding and tokenization is applied. Specifically, the Byte-Pair Encoding (BPE) tokenization algorithm is used, which merges frequently occurring pairs of bytes to effectively handle out-of-vocabulary token combinations [24].

Each transformer decoder layer consists of a multi-head self-attention block, enabling the model to learn contextual relationships between input tokens [25, 26]. Due to the fact that attention is masked, the model is unable to ”cheat” by attending to future input tokens. Attention is followed by a feedforward neural network, which applies non-linear transformations to each token allowing the model to understand more sophisticated features and thus make better predictions. Layer normalization and residual connections are added to each block to mitigate vanishing gradients [27].

After all decoder layers have processed the input, the language model head applies an unembedding matrix to project the final hidden state back to human-readable vocabulary, thereby assigning logits (unnormalized probabilities) to each token. The final softmax layer ultimately normalizes these logits to produce a finite probability distribution over the vocabulary. Generally, the token associated with the highest probability is chosen as output prediction - also referred to as *greedy decoding*.

To assess CodeGPT's cross-lingual performance, we use a multilingual CodeGPT model[7] that was initialized as GPT-2, but fine-tuned on the *CodeSearchNet* dataset [28] containing

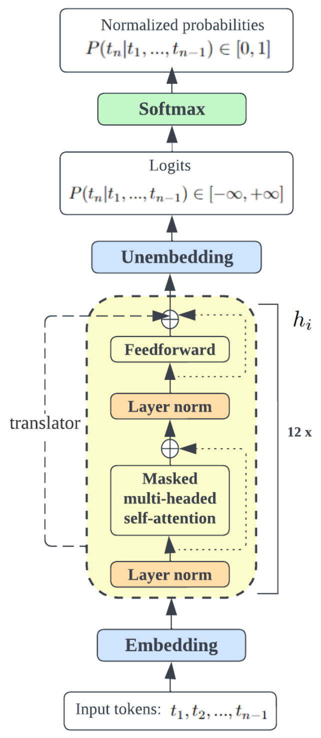

Normalized probabilities
$P(t_n|t_1,...,t_{n-1}) \in [0,1]$
Softmax
Logits
$P(t_n|t_1,...,t_{n-1}) \in [-\infty,+\infty]$
Unembedding
h_i
Feedforward
Layer norm
translator
12 x
Masked multi-headed self-attention
Layer norm
Embedding
Input tokens: $t_1, t_2, ..., t_{n-1}$

Figure 1: Model architecture of CodeGPT (GPT-2-based).

open-source Github projects in the following six programming languages: Java, Python, Javascript, Go, Ruby and PHP.

3.2 Pre-processing

Prior to our experiments, a pre-processing phase (see Figure 2) starts off by removing comments and documentation within source files across all languages. Despite the fact that CodeGPT was trained on source code containing comments and documentation, we are merely interested in performance on *code*. Furthermore, files containing fewer than 1,024 tokens (max. context length) were filtered out to ensure sufficient left context for reliable predictions. At the same time, files having more than 1,024 + C tokens - where C is the number of token predictions to make - were randomly sliced to reduce overhead. Afterwards, the processed input code is tokenized into numeric identifiers using BPE, which can be fed to the model.

3.3 Experiment 1: Tuned Lens Investigation

The model's hidden states are passed to a *tuned lens*[8], to inspect how intermediate model predictions vary across languages whilst performing code completion. This lens trains a translator (see Figure 1) that maps intermediate layers h_1 - h_{10} to the final layer h_{11}. Ultimately, this leads to smooth transition between hidden states h_i and the unembedding layer [29].

For each hidden layer h_i, the *tuned lens* outputs logits over the model vocabulary. By greedily picking the highest logit,

[6]Refers to generating code via a natural language desciption.
[7]https://huggingface.co/AISE-TUDelft/CodeGPT-Multilingual

[8]https://github.com/jkatzy/tuned-lens

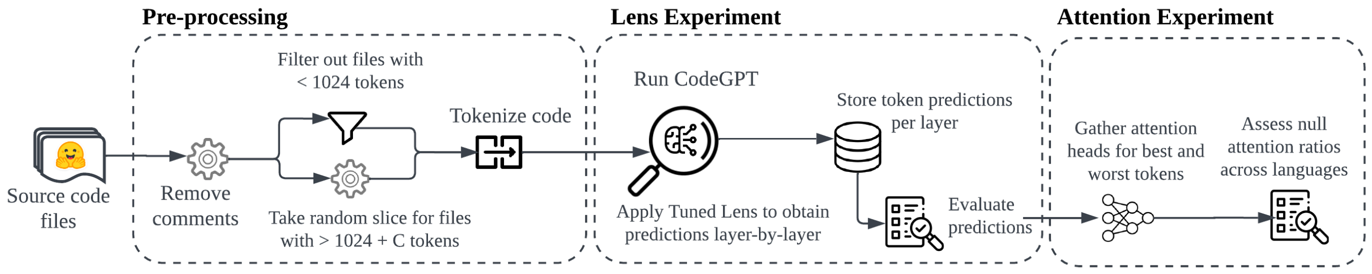

Pre-processing
Lens Experiment
Attention Experiment
Source code files
Remove comments
Filter out files with < 1024 tokens
Take random slice for files with > 1024 + C tokens
Tokenize code
Run CodeGPT
Apply Tuned Lens to obtain predictions layer-by-layer
Store token predictions per layer
Evaluate predictions
Gather attention heads for best and worst tokens
Assess null attention ratios across languages

Figure 2: Data pipeline used for our lens and attention experiments.

we obtain human-readable token predictions. The final decoder layer h_{11} produces the resulting token prediction made by the model. Hence, we can compare these final predictions against the ground truth (i.e. subsequent input token) to examine the overall accuracy of CodeGPT.

More importantly however, the intermediate layer-by-layer predictions provide us with information about the model's confidence. For instance, token predictions that were correct from the first layer h_0 (highest depth) are the most *confident* and *correct* model predictions. We consider these predictions the "best".

In our cross-lingual study of CodeGPT, we aim to examine in which scenarios it predicts *incorrect* tokens with *high confidence*. We refer to these as the "worst" predictions, as the model emits unjustly high confidence. For comparison purposes, we locate the top-10 worst and best predicted tokens - serving as areas of interest for our follow-up attention experiment.

3.4 Experiment 2: Attention Investigation

As demonstrated by earlier works [3, 4, 23], analyzing attention is helpful for understanding how the model focuses on various input tokens - despite the fact that they may not provide full explanations for individual predictions [30].

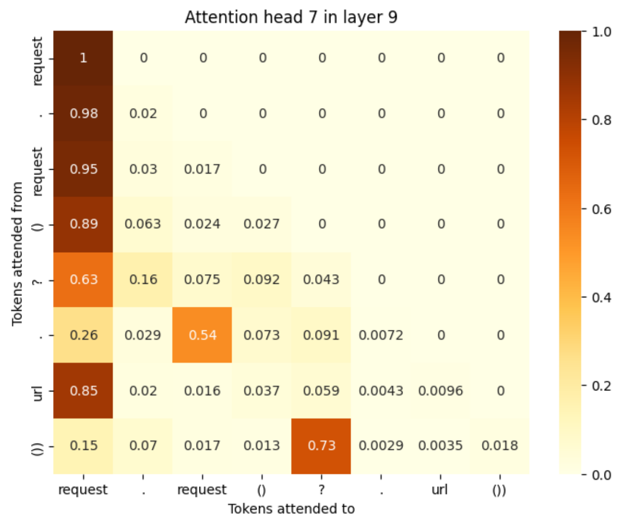
Specifically, the self-attention mechanism allows every token x_i within input sequence x of size N to assign an attention score $\alpha_{i,j}(x) \in [0, 1]$ to some preceding token x_j - as summarized in (1).

$$A(x_i) = (\alpha_{i,0}(x), \alpha_{i,1}(x), ..., \alpha_{i,i}(x)) \qquad (1)$$

where $\sum_{j \leq i} \alpha_{i,j}(x) = 1$. This set of attention weights $A(x_i)$ is uniquely learned by all 12 x 12 heads, in the form of a lower-triangular matrix of size N x N (see example in Figure 3).

Using the attention visualization tool *Bertviz* [31], one could infer a subtle triangular pattern captured by this attention head (see Figure 4), implying that a high proportion of attention is assigned to the first token `request`. In previous work carried out by Vig et al. [23], attention assigned to the first token was referred to as "null attention". As the first token is attended to by default when no useful tokens are found in the input, it is considered uninformative.

Inspired by his work, the aim of our attention investigation is to correlate cross-lingual proportions of null attention with differences in behavior exhibited by CodeGPT. Given



Tokens attended from	request	.	request	()	?	.	url	())
request	1	0	0	0	0	0	0	0
.	0.98	0.02	0	0	0	0	0	0
request	0.95	0.03	0.017	0	0	0	0	0
()	0.89	0.063	0.024	0.027	0	0	0	0
?	0.63	0.16	0.075	0.092	0.043	0	0	0
.	0.26	0.029	0.54	0.073	0.091	0.0072	0	0
url	0.85	0.02	0.016	0.037	0.059	0.0043	0.0096	0
())	0.15	0.07	0.017	0.013	0.73	0.0029	0.0035	0.018

Figure 3: Heatmap depiction of attention matrix in head 7, layer 9 - given example input sequence `request.request()?.url()`).

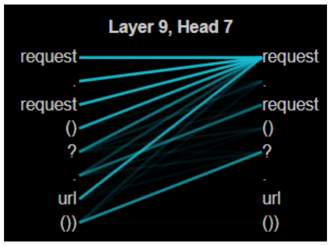


Figure 4: *Bertviz* visualization of attention matrix displayed in Figure 3, in which thickness of token-to-token lines indicate intensity of attention scores.

a language, we compute these ratios across attention heads associated with the top-10 *best and worst predicted tokens* - capturing the setting in which the model was most confident. To compare this with average model behavior, we also analyze attention heads involved in *arbitrary predictions*. Given that multiple earlier findings [3, 23, 31] have demonstrated that attention distribution varies across layers, we conduct a layer-by-layer breakdown of null attention in order to detect latent patterns within the model.

4 Experimental Setup

This section presents the test bench we have utilized to investigate CodeGPT's cross-lingual performance on code completion, as well as the setup used during both our lens and attention experiments. Moreover, we describe the chosen metrics to evaluate model performance.

4.1 Test Datasets

All input source files have been adapted from *the Stack* [32], a 3.1 TB pre-training dataset for code LLMs. It covers permissively-licensed open-source Github projects across 30 different programming languages, gathered from 137.4M unique repositories published between 1 January 2015 and 31 March 2022.

For our study, we have randomly selected 512 files for three high-resource (*Java, *Python, C++) and three low-resource (Kotlin, *Go, Julia) languages. Throughout this paper, * is used to indicate languages CodeGPT was fine-tuned on.

It may be noteworthy to mention that the *CodeSearchNet* corpus used for fine-tuning contains 1.6M, 1.2M and 0.7M functions for Java, Python and Go respectively. This implies that the latter low-resource language we assess the model on is underrepresented by approximately 50%.

4.2 Model Inference

To ensure that all intermediate predictions generated by the *tuned lens* are based on left context only (i.e. unbiased), all input files to be fed to CodeGPT were split into C windows of 1,024 tokens (max. content length). Essentially, each window corresponds to a single token prediction out of the total C predictions to be made[9].

Within a file we only start predictions from token index 1,024 onward, to ensure optimal left context and equally sized input as expected by the model. More importantly, this guarantees all attention matrices are of size 1,024 x 1,024.

To make model inference more efficient, these windows were fed to the model in batches - thereby exploiting parallel computation. As prior work [33] has shown that batch size affects prediction results, we ensured the same size across all source files and languages.

4.3 Gathering Attention Heads

As initial step towards our attention investigation, we feed CodeGPT with 60 (out of 512) randomly sampled source files of a given language - associated with the top-10 *best and worst predicted tokens*. To ensure diversity in input context fed to the model, we sample files in such a way that each token of interest is represented by three distinct files.

Subsequently, for each file, we let the model perform single-token prediction on the corresponding token of interest ($C = 1$). Out of the resulting 12 x 12 attention heads of each prediction, we arbitrarily select 12 - one head per model layer[10]. This approach yields diversity in terms of attention heads, which is crucial as the capturing of long-distance relationships across tokens varies significantly between heads - especially in deeper model layers [23].

A similar methodology was followed for collecting attention heads associated with *arbitrary* predictions. Thus, in total 2 sets x 60 files x 12 heads = 1440 attention heads of size 1,024 x 1,024 have been assessed for each language.

4.4 Evaluation Metrics

For assessing CodeGPT's overall performance on the token-level code completion task across languages, the following metrics were used:

$$\text{Top-1 Accuracy} = \frac{|Q|}{C} \tag{2}$$

where Q is the set of correctly predicted tokens and C is the number of tokens to be predicted within an input sequence of size N.

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{depth} \tag{3}$$

The top-1 accuracy (2) provides a decent indication of accuracy across the cross-lingual input files, but this metric does not consider the confidence of the model - or depth of predictions. Hence, the Mean Reciprocal Rank (MRR) is included to take into account intermediate predictions as well. Essentially, we have defined the MRR as the reciprocal of the average depth of correct predictions (3). This implies that the lower the depth, the more confident the model is about a correctly predicted token - making it a more reliable prediction.

To compute the ratio of null attention within a given attention matrix A, we consider the proportion of attention scores from all N tokens x_i to the first token x_0 being at least 0.9 as can be seen in (4).

$$\text{Null Attention Ratio} = \frac{1}{N}\sum_{i=0}^{N}\alpha_{i,0}(x) \geq 0.9 \tag{4}$$

where N is the total number of input tokens, i.e. the length and width of A.

In the example attention head depicted in Figure 3, this ratio would be $3/8$, or 37.5%.

4.5 Implementation and Configuration

Access to our CodeGPT model and its corresponding lens[11] was facilitated by *Huggingface*. Additionally, the open-source *PyTorch* framework provided the backbone that enables efficient inference of transformer models.

Expensive computations were delegated to *DelftBlue*'s[12] servers, providing 32GB Tesla V100S GPUs.

[9]For our *tuned lens* experiment, we set C to the maximal no. of predictions possible from token 1,024 onward.

[10]Due to limited memory resources for handling large amounts of 1024 x 1024 matrices.

[11]https://huggingface.co/AISE-TUDelft/CodeGPT-Multilingual-lens

[12]https://www.tudelft.nl/dhpc/system

	Top-1 Accuracy (%)	MRR (%)
***Java**	69.2	16.93
***Python**	68.22	14.74
C++	64.52	11.55
***Go**	67.94	12.35
Kotlin	58.25	11.76
Julia	65.06	11.24

Table 2: Cross-lingual performance of CodeGPT on the single-token code completion task, averaged over 512 input files per language.

5 Results

Below, we present our findings obtained by the lens- and attention experiments conducted. These results are evaluated later in Section 6.

5.1 Results Experiment 1: Tuned Lens Investigation

The overall cross-lingual performance of CodeGPT on the token-level code completion task is presented in Table 2, based on the Top-1 accuracy and MRR averaged over 512 source code files per language.

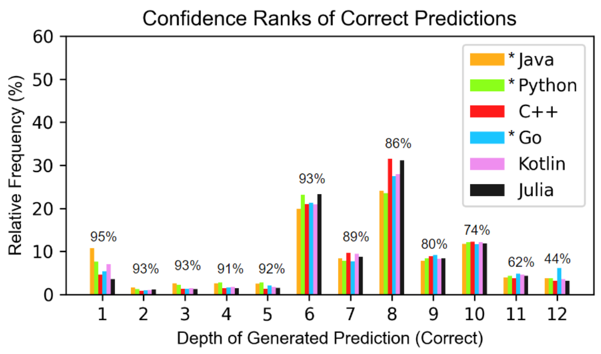
Next, Table 3 displays the top-10 best and worst predicted tokens by CodeGPT across all languages. Here, ”best” entails correctly predicted with the highest confidence possible, i.e. the model was able to generate the correct token from the very first layer. The same notion can be applied to ”worst” predictions, which are associated with incorrect predictions however.

All tokens have been categorized into one of the following types: user-defined elements (transparent), language-defined elements (yellow), common code structures (blue) and punctuation (orange). Moreover, we have excluded a defined set of commonly over-dominating tokens: newlines, underscores, tabs, parentheses, commas, and dots. This facilitated the exploration of novel patterns across programming languages.

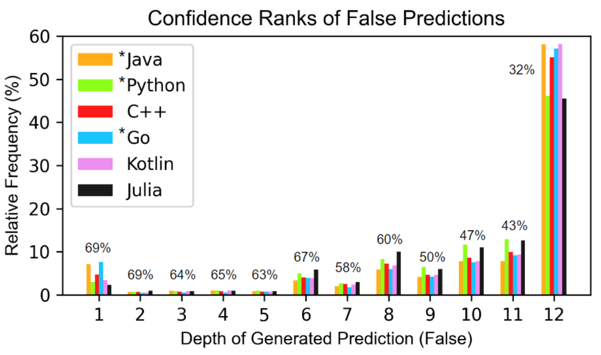
Finally, Figure 5 visualizes the cross-lingual spread of prediction depths across all 12 network layers - separately for correct[13] and false predictions. In both cases holds that depth $d = 1$ is associated with the ”best” and ”worst” predictions. Higher depths signify lower model certainty, since more hidden layers were required to generate the output prediction - which could be either correct or incorrect.

In our work, the notion of ”model confidence” revolves around prediction *depths*. Nonetheless, one could also consider the *probability* assigned to the resulting token generated by the output layer of the model: $P_{output} \in [0, 1]$. In Figure 5, each depth d is annotated with the obtained P_{output} (%) averaged over all (correct or false) predictions of depth d. To illustrate, P_{output} = 69% for d = 1 in 5b implies that the worst token predictions (false with highest confidence) had an average probability of 0.69.

[13]This histogram corresponds to the reciprocal of MRR scores displayed in Table 2.



(a) Cross-lingual spread of depths of correct token predictions.



(b) Cross-lingual spread of depths of false token predictions.

Figure 5: The cross-lingual spread of confidence levels of predictions by CodeGPT. Each depth d is annotated with P_{output} (%) averaged over predictions of depth d.

5.2 Results Experiment 2: Attention Investigation

We depict the cross-lingual, layer-by-layer ratios (%) of null attention as heatmap in Figure 6. For each language, the layer-specific ratios are averaged over 60 randomly sampled attention heads of size $1,024$ x $1,024$ found in that layer.

Additionally, we display the mean (%) and standard deviation (SD) of null attention ratios per language - computed over the last model layers 6-12. We deem the first layers negligible due to the relatively uniform values across all languages. This choice can be justified by Vig et. al [23], who found that attention heads in the initial model layers typically assign small attention scores to the first token.

Furthermore, we have computed average ratios of null attention for the top-10 *best and worst* predicted tokens (see Table 3) across all languages. In Appendix B we display for each individual token, the mean (%) and SD over ratios computed across layers 6-12. Tokens that are among *both* the best and worst predictions of a language have not been considered, as we aim to study tokens that are consistently being predicted correctly or incorrectly by the model.

*Java		*Python		C++		*Go		Kotlin		Julia	
Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst
get	}	self	def	;	}	x	}	get	}	1	end
;	public	s	1	x	if	:	or	s	return	0	1
<	return	get	class	get	{	n	?	qual	if	2	2
als	if	0	:	n	->	for	!	s	get	t	@
\\	@	1	=	0	return	n	<	ator	"	[	[
Exception	VK	if	if	1	#	!!	case	n	get	ices	:
String	private	n	@	s	::	son	<	els	"	[	[
or	assert	'	[	int	case	s	is	point	get	n	{
of	this	format	2	or	void	n	java	1	function	]	else
adata	{	args	0	r	log	ert	to	els	var	estamp	]

Table 3: Top-10 tokens predicted correctly (best) or incorrectly (worst) by CodeGPT with highest confidence - obtained by performing the token-level code completion task across 512 files per language. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

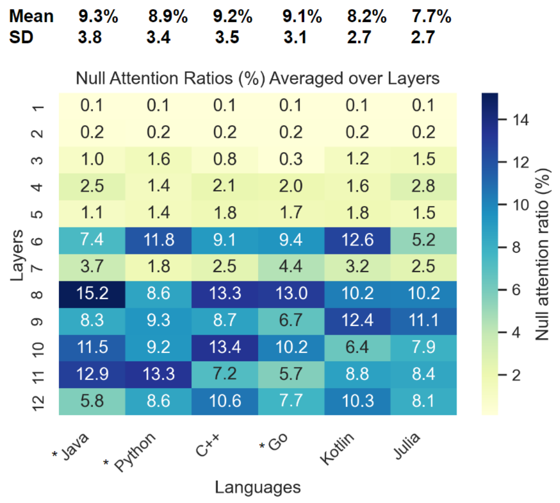


Figure 6: Cross-lingual ratios of null attention for arbitrary attention heads of size 1,024 x 1,024, broken down by layer.

6 Discussion

Below we will present our interpretation to the results presented previously, thereby evaluating the limitations of CodeGPT's performance on the token-level code completion task across high- and low-resource languages.

6.1 Strong and Weak Areas

As can be seen by the Top-1 accuracy and MRR scores in Table 2, **CodeGPT performs most reliably for Java and Python input** - the two high-resource languages the model was primarily trained on. The slightly better performance in Java can be attributed to the fact that the *CodeSearchNet* corpus contains 1.6M Java methods, whereas it covers 1.2M Python functions. Similarly, the number of Go functions was only 0.7M, which probably explains its third rank. Low-resource languages that the model was not trained on, Kotlin and Julia, obtain the worst performance.

Overall, **the *best* predictions for languages the model was not trained on seem to be endings of user-defined conjunction literals** such as `mbol` (Table 3). The fact that this is natural language related suggests that CodeGPT does not comprehend code syntax specifics of these (mostly low-resource) languages. Instead, the model appears to be relying on its uni-directional architecture as well as its inherited natural language understanding through GPT-2. In contrast, the best token predictions for Java and Python mostly contain common code structures and language specific elements.

The *worst* predictions for languages CodeGPT was not fine-tuned on involve language-specific constructs. For languages it *was* trained on, most of the worst predictions are constructs that cannot be predicted from left context only. Hence, this is due to the inherent limitations of CodeGPT's architecture[14] rather than lack of knowledge on language-specifics.

6.2 Model Confidence

The spread of model confidence levels appears to be consistent across all six languages. Specifically, it can be observed that **a high proportion of *correct* predictions have depths between 6 and 8** (see Figure 5a), implying mediocre model confidence. In fact, this observation aligns with previous findings [4, 23] suggesting that attention aligns most strongly with dependency relations in the middle layers.

Interestingly, the spread of confidence levels for *false* predictions follows a rather left-skewed pattern: **50-60% of all false predictions had depth 12, i.e. the lowest model confidence possible.** This observation backs findings by Halawi et al. [34] who revealed that given inaccurate contexts, language models seem to produce accurate predictions up until some "critical" layer - after which it changes its mind and replaces this with an incorrect prediction that is more fitting given the "false" context.[15]

Furthermore, for all languages it became evident that the average probabilities assigned to predicted tokens P_{output} decrease as the prediction depths increase. Thus, confidence in terms of prediction *depths* and *probabilities* correlate positively.

[14]This also explains why '}' is the number one worst predicted token across most languages.

[15]This phenomenon can be seen in Appendices A.2 to A.6.

6.3 Null Attention Patterns

When considering the average ratios of null attention within CodeGPT (Figure 6), a common pattern can be observed across all languages. Namely, **the deeper model layers have the highest null attention ratios on average.** Whereas this trend was presented in an earlier study by Vig et al. [23] into LMs for *NLP* tasks, the relation between null attention and capturing *code* syntax has not been analyzed so far.[16]

Moreover, we observe a **correspondence between this distribution of null attention ratios and confidence levels of correct predictions** (Figure 5a). In both cases, there is a noticeable shift from layer 6 onward, which can be attributed to earlier revelations [3, 4] that deeper layers are involved in targeting specific code constructs and longer-range dependency relations. Meanwhile, most attention heads in the lower layers attend to token *position* without considering content. Intuitively, the model no longer has to "occupy" all attention heads from the middle layers onward, leading to higher null attention ratios.

We can also infer a **positive correlation between variance of null attention ratios across layers and model confidence** when considering the ratios associated with the *best and worst* predicted tokens.[17] In this setting in which the model is *most confident*, the SDs are far higher[18] than those presented in Figure 6. This variation can be explained by the finding that construct-specific attention heads cluster by layer depth [23].

Overall, the proportion of null attention is highest for the languages CodeGPT was trained on (Java, Python, Go), as well as C++ which has similarities with Java in terms of syntax. These are also the languages having the highest SD across layers. Whereas this observation seems to suggest that CodeGPT is most confident and accurate in these languages, we cannot deduce a *causal* relationship from the correlations revealed above.

To substantiate the above statement - within a language, the highest means and SDs of null attention do not necessarily correspond to correctly predicted tokens. In fact, the overall magnitude and variance null attention ratios are similar *between* the best and worst tokens, as well as between token categories.

6.4 Threats to Validity

Threats to Internal Validity: Relate to the threshold we have set for defining null attention to 0.9 - inspired by earlier research [23]. Moreover, we have excluded a predefined set of frequently-occurring punctuation tokens, leading to lower performance *statistics* when compared to earlier findings [7] on CodeGPT's performance.[19] However, we applied the same methodology across all languages under investigation, giving grounds for our cross-lingual evaluation.

[16]In fact, Vig et al. filtered out null attentions in his study.

[17]Displayed in Appendix B for optional inspection.

[18]The average SD over all languages and layers in a regular setting (Figure 6) is 3.2, whereas this is 10.2 in the setting corresponding to highest confidence (Appendix B).

[19]At least, for Java and Python - the two languages CodeGPT was assessed on in earlier work.

Threats to External Validity: Our results can be generalized to the "real world", as the test corpus used (*the Stack*) contains permissively licensed source code collected from Github repositories. Whereas it contains varying numbers of source files across languages, we have arbitrarily sampled 512 files for all six languages - thus ensuring equal representation across languages. Given the currently available literature, it cannot be concluded whether there is overlap between *the Stack* and *CodeSearchNet* dataset used for fine-tuning CodeGPT. Future studies can improve generalizability by using more input files across languages. Similarly, one may benefit from inspecting *all* attention heads - rather than randomly selecting a few per layer.

Threats to Construct Validity: The performance metrics we use, Top-1 accuracy and MRR, ensure diversity in the sense that we consider both the model's output layer and hidden layers. Both metrics occur frequently in literature [21, 35–37]. Also, our finding that confidence in terms of prediction *depth* and *probability* correlate positively, signifies the suitability of MRR - which we have defined in terms of *depth*.

7 Conclusions and Future Work

Most current code language models are predominantly trained on widely-known programming languages, which limits their performance when applied to lower-resource languages. In this work, we have evaluated the GPT-2-based CodeGPT on the token-level code completion task, across both high- (*Java, *Python, C++) and low-resource languages (*Go, Kotlin, Julia).

Through our *tuned lens* investigation, we found that CodeGPT generates most confident and accurate token predictions for the two high-resource languages it was mainly fine-tuned on: Java and Python. It achieves Top-1 accuracies (69.2%, 68.2%) and MRR scores (16.9, 14.7) respectively.

Moreover, the type of tokens being predicted *incorrectly* with *highest certainty* differs across languages depending on whether or not the model was fine-tuned on it. For languages familiar to CodeGPT, these "worst" tokens are primarily constucts that cannot be predicted from left context only - which can be attributed to the inherent uni-directional model architecture. In contrast, the worst predicted tokens for languages the model was *not* fine-tuned on involve language-specific elements which do not occur in the other languages.

From our attention investigation, it became evident that there is a positive correlation between model *accuracy* and *magnitude* of null attention ratios. Moreover, model *confidence* correlates positively with *variance* of null attention across layers. However, investigating null attention is not sufficient to explain cross-lingual differences between the *best and worst* predictions, nor *token categories*. Future research may perform clustering on attention heads to inspect common or unique patterns beyond null attention. Additionally, as we have encountered limitations of CodeGPT's unidirectional architecture, future studies can benefit from analyzing bidirectional models as well. Finally, this study can be extended by also investigating statement- and block-level code completion performance.

8 Responsible Research

Below we will reflect on the ethical aspects of this research, including reproducibility and transparency regarding the experiment methodology adopted.

8.1 Transparency

In this paper, we have ensured to be transparent about what external test corpus was used to evaluate the performance of CodeGPT on the token-level code completion task across different programming languages. We have also documented and visualized how this raw data was pre-processed before being fed to the model.

Furthermore, we have clearly indicated the number of input source files and attention heads that were involved in our investigation. Earlier in Section 6.4, we had noted that generalizability of this study can be improved by incorporating larger amounts of data.

We have also made sure to critically assess secondary sources of input data. It may be noteworthy to mention that the datasets used for fine-tuning and evaluating the model in this work both contain source files from open-source Github repositories. From available literature on these datasets, it is not clear whether these two datasets are overlapping - as mentioned in Section 6.4.

Additionally, we have clarified how CodeGPT's performance was measured - including what exact metrics we used and what experimental settings were adopted to obtain the results presented. We also believed it was crucial to be transparent about the way we have interpreted these results.

Overall, by being transparent we strive to make evident to peers how the research was conducted and what the various phases of the process were.

8.2 Reproducibility

Firstly, ensuring reproducibility extensively relies on safeguarding transparency regarding our research methodology.

Moreover, we made sure that all processes involving random sampling were done so using a random seed based on the corresponding sample identifier - defined by the original, publicly-available test corpus *the Stack*.

Finally, we have open-sourced our Github repository used for this study at: https://github.com/AISE-TUDelft/CodeShop. Other resources required to reproduce this study have been made public via *Huggingface*:

- The multilingual CodeGPT model assessed in this work
- CodeGPT-specific lens
- Java dataset adapted from the Stack
- Python dataset adapted from the Stack
- C++ dataset adapted from the Stack
- Go dataset adapted from the Stack
- Kotlin dataset adapted from the Stack
- Julia dataset adapted from the Stack

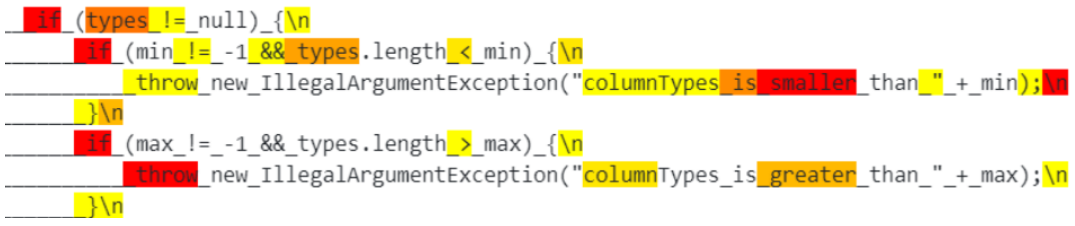
Acknowledgements

I would like to thank Jonathan Katzy and Maliheh Izadi for supervising us throughout this research project, and helping us out during challenging moments. Furthermore, I would like to express my appreciation to Arie van Deursen, for providing valuable feedback. Finally, I am grateful for Erik Mekkes, Miranda Keeler and Razvan Popescu for being open to brainstorm on ideas and share insights.

A Tuned Lens Visualizations and Model Confidence

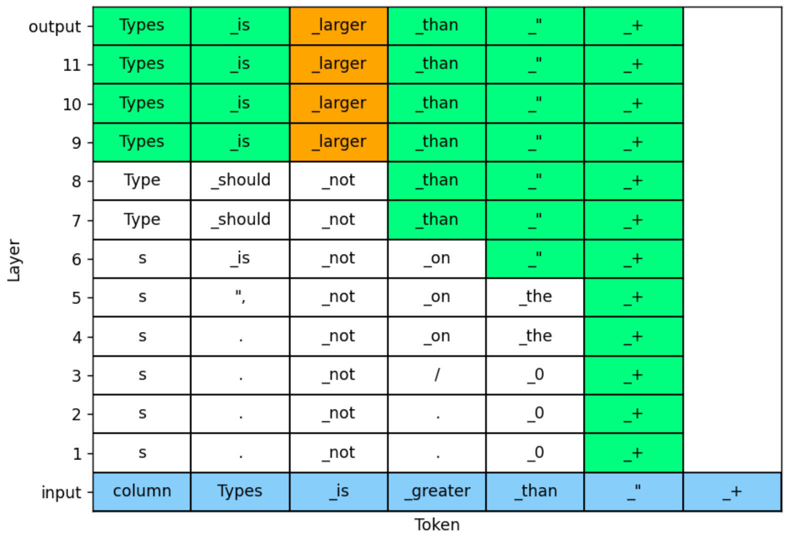
In this section, we depict results obtained by applying the *tuned lens* on hidden states of CodeGPT involved in the token-level code completion task across six languages: *Java, *Python, C++, Kotlin, *Go, Julia (see Section 3.3 for more details). Essentially, below is displayed a layer-by-layer visualization of the model's intermediate token predictions given some input[20] - indicated in blue at the bottom.

A.1 *Java



```
__if_(types_!=_null)_{\n
______if_(min_!=_-1_&&_types.length_<_min)_{\n
__________throw_new_IllegalArgumentException("columnTypes_is_smaller_than_"_+_min);\n
______}\n
______if_(max_!=_-1_&&_types.length_>_max)_{\n
__________throw_new_IllegalArgumentException("columnTypes_is_greater_than_"_+_max);\n
______}\n
```

Figure 7: Java code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).

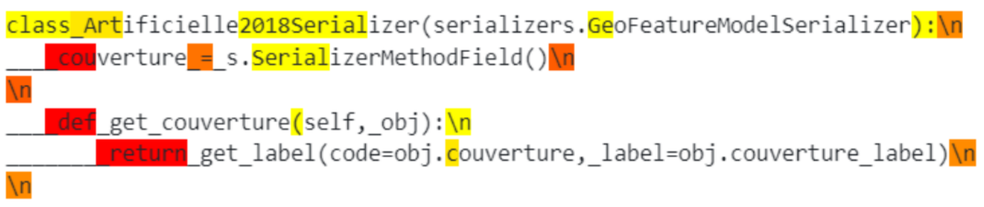


Layer							
output	Types	_is	_larger	_than	_"	_+	
11	Types	_is	_larger	_than	_"	_+	
10	Types	_is	_larger	_than	_"	_+	
9	Types	_is	_larger	_than	_"	_+	
8	Type	_should	_not	_than	_"	_+	
7	Type	_should	_not	_than	_"	_+	
6	s	_is	_not	_on	_"	_+	
5	s	",	_not	_on	_the	_+	
4	s	.	_not	_on	_the	_+	
3	s	.	_not	/	_0	_+	
2	s	.	_not	.	_0	_+	
1	s	.	_not	.	_0	_+	
input	column	Types	_is	_greater	_than	_"	_+
	Token						

Figure 8: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

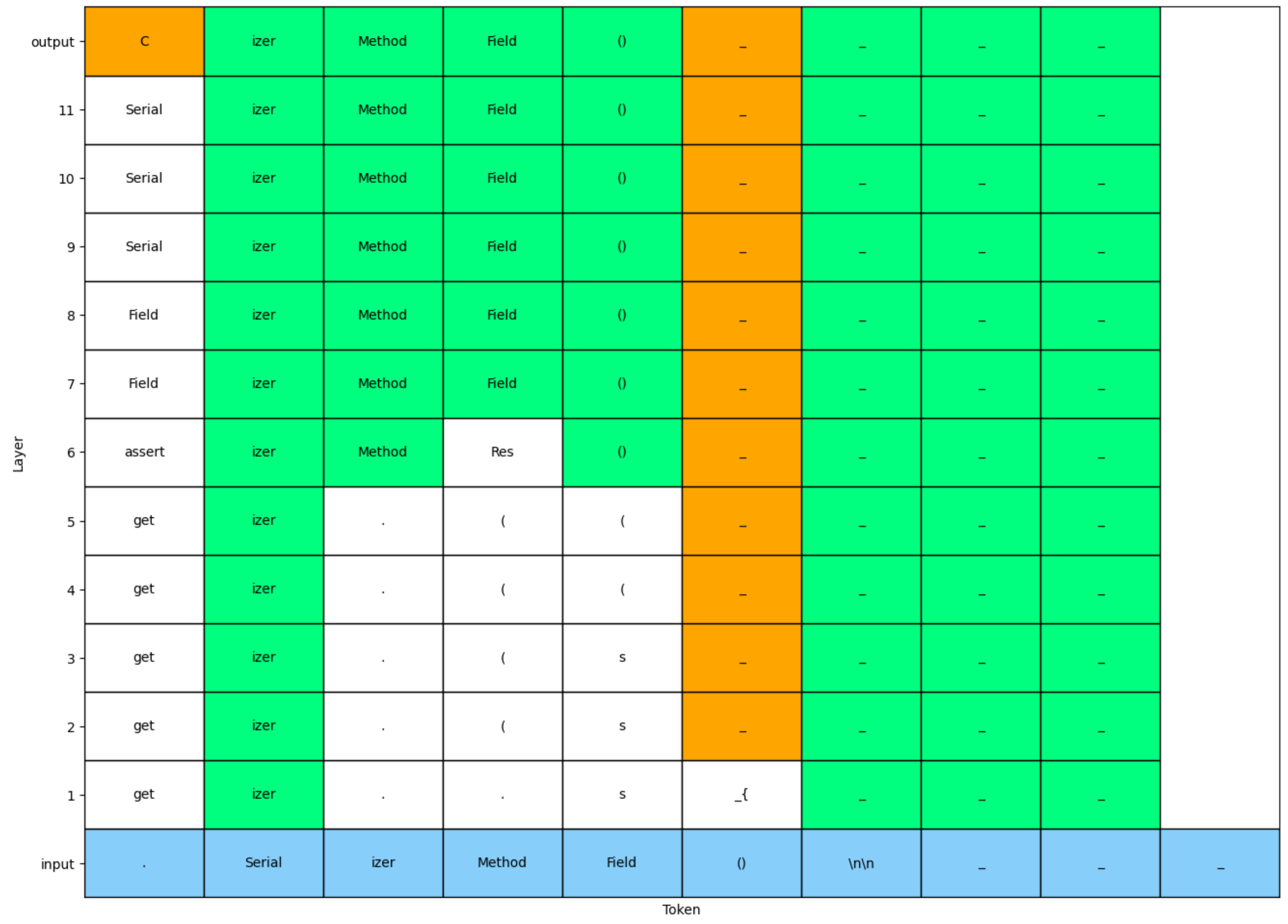
[20]Since this is merely a toy visualization, the inputs used are rather short. In our study, we assessed inputs of size 1,024.

A.2 *Python



```
class_Artificielle2018Serializer(serializers.GeoFeatureModelSerializer):\n
____couverture_=_s.SerializerMethodField()\n
\n
____def_get_couverture(self,_obj):\n
________return_get_label(code=obj.couverture,_label=obj.couverture_label)\n
\n
```

Figure 9: Python code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).

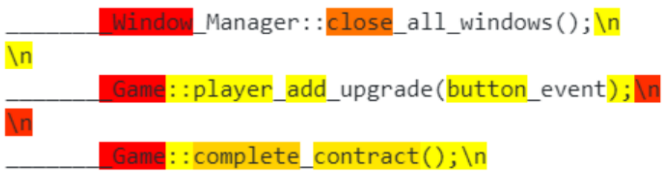


Layer										
output	C	izer	Method	Field	()	_	_	_	_	
11	Serial	izer	Method	Field	()	_	_	_	_	
10	Serial	izer	Method	Field	()	_	_	_	_	
9	Serial	izer	Method	Field	()	_	_	_	_	
8	Field	izer	Method	Field	()	_	_	_	_	
7	Field	izer	Method	Field	()	_	_	_	_	
6	assert	izer	Method	Res	()	_	_	_	_	
5	get	izer	.	(	(	_	_	_	_	
4	get	izer	.	(	(	_	_	_	_	
3	get	izer	.	(	s	_	_	_	_	
2	get	izer	.	(	s	_	_	_	_	
1	get	izer	.	.	s	_{	_	_	_	
input	.	Serial	izer	Method	Field	()	\n\n	_	_	_

Token

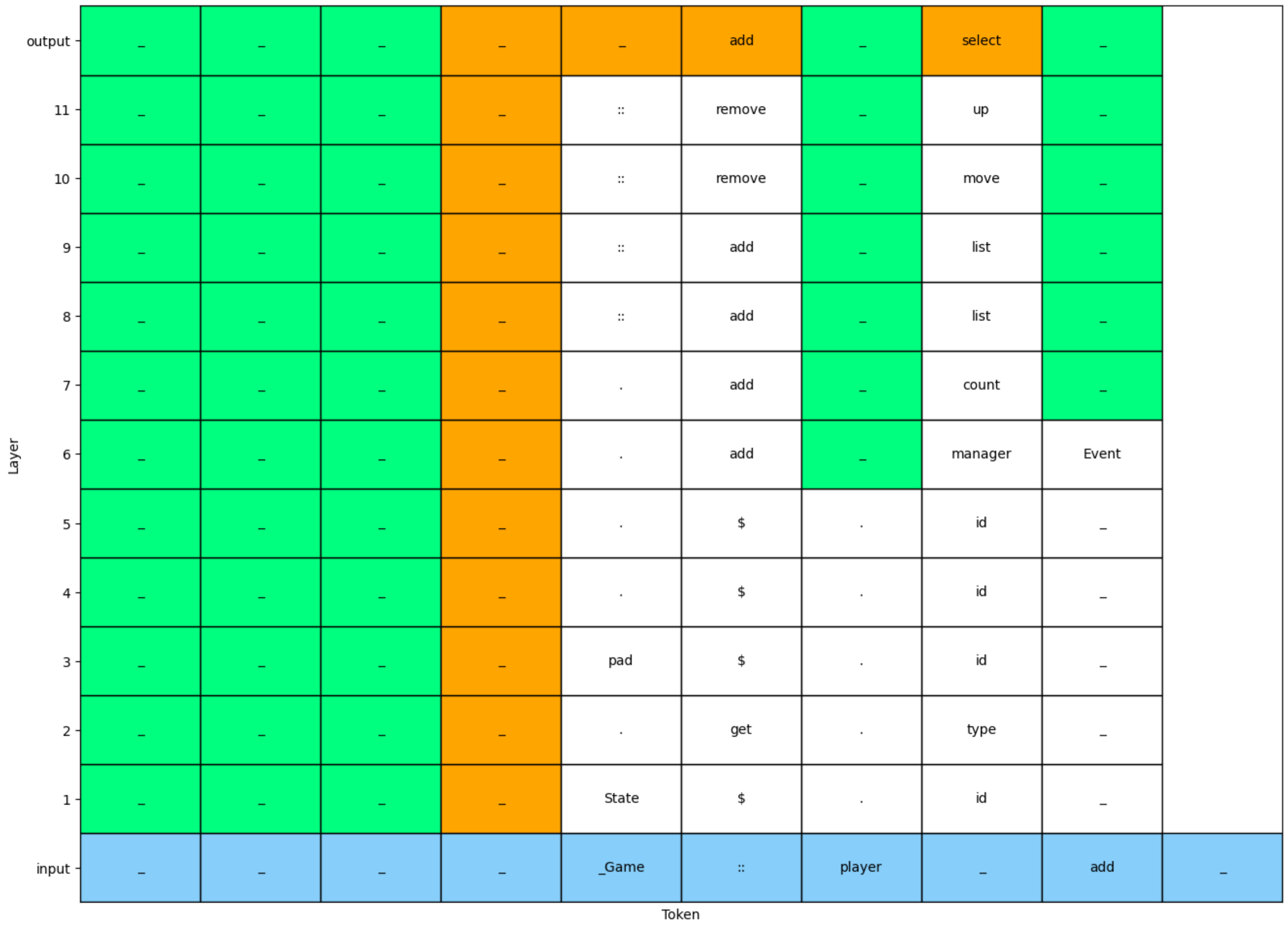
Figure 10: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

A.3 C++



```
________Window_Manager::close_all_windows();\n
\n
________Game::player_add_upgrade(button_event);\n
\n
________Game::complete_contract();\n
```

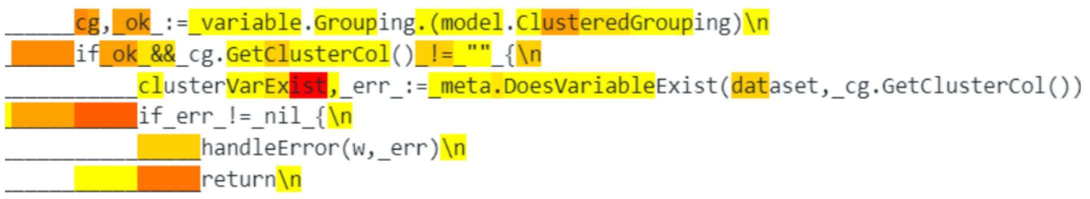
Figure 11: C++ code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).



Layer										
output	_	_	_	_	_	add	_	select	_	
11	_	_	_	_	::	remove	_	up	_	
10	_	_	_	_	::	remove	_	move	_	
9	_	_	_	_	::	add	_	list	_	
8	_	_	_	_	::	add	_	list	_	
7	_	_	_	_	.	add	_	count	_	
6	_	_	_	_	.	add	_	manager	Event	
5	_	_	_	_	.	$	.	id	_	
4	_	_	_	_	.	$	.	id	_	
3	_	_	_	_	pad	$	.	id	_	
2	_	_	_	_	.	get	.	type	_	
1	_	_	_	_	State	$	.	id	_	
input	_	_	_	_	_Game	::	player	_	add	_

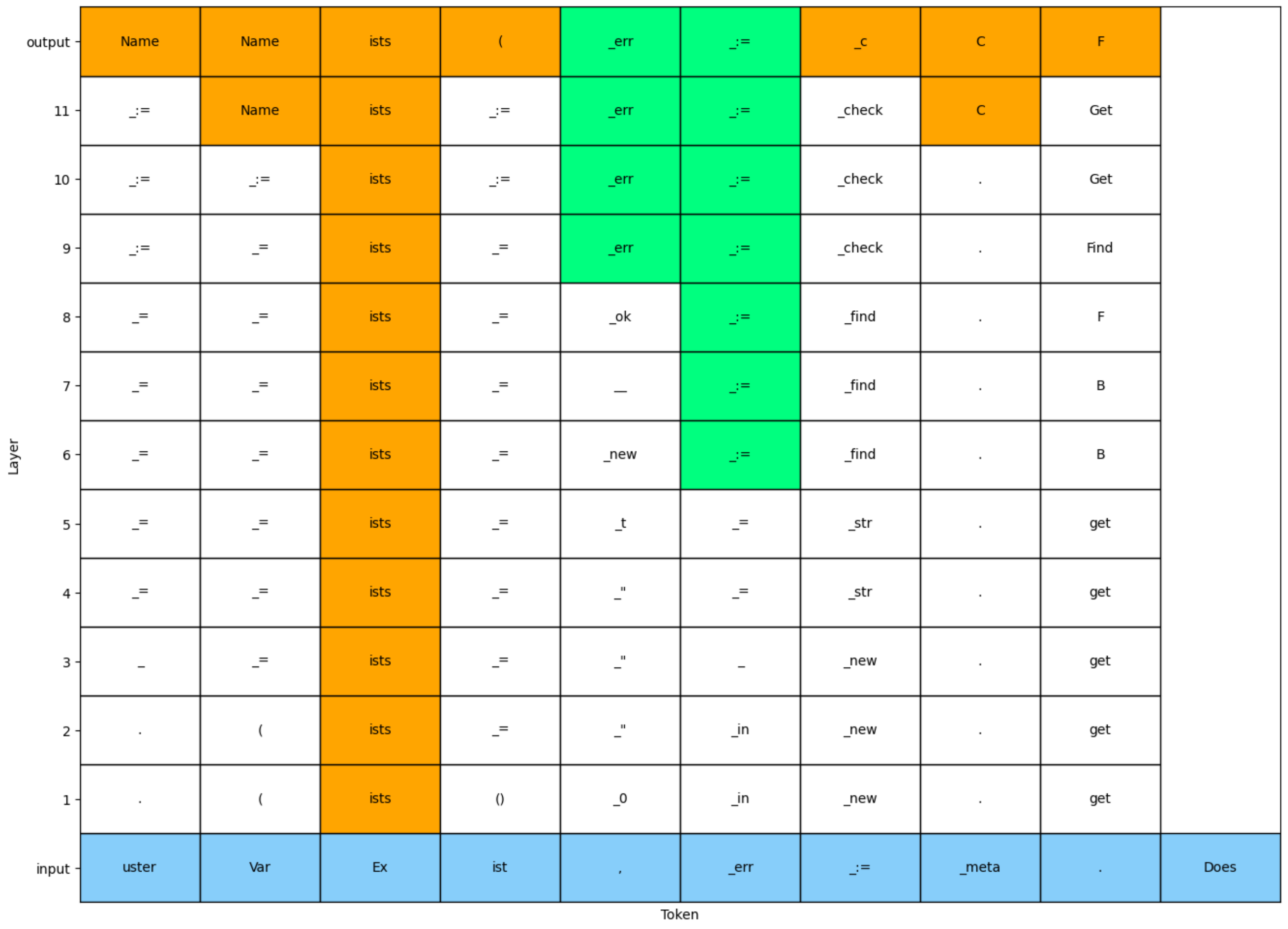
Figure 12: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

A.4 *Go



```
______cg,_ok_:=_variable.Grouping.(model.ClusteredGrouping)\n
______if_ok_&&_cg.GetClusterCol()_!=_""_{\n
___________clusterVarExist,_err_:=_meta.DoesVariableExist(dataset,_cg.GetClusterCol())
___________if_err_!=_nil_{\n
_________________handleError(w,_err)\n
_________________return\n
```

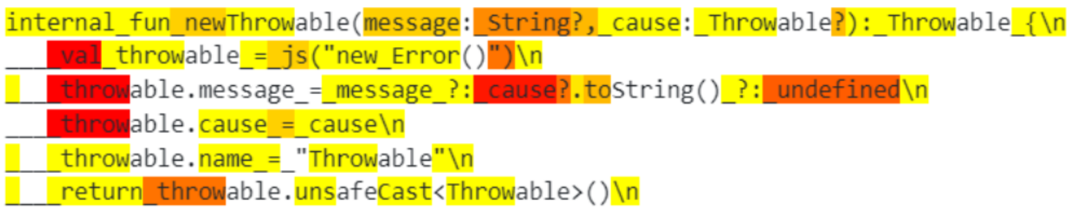
Figure 13: Go code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).



Layer										
output	Name	Name	ists	(	_err	_:=	_c	C	F	
11	_:=	Name	ists	_:=	_err	_:=	_check	C	Get	
10	_:=	_:=	ists	_:=	_err	_:=	_check	.	Get	
9	_:=	_=	ists	_=	_err	_:=	_check	.	Find	
8	_=	_=	ists	_=	_ok	_:=	_find	.	F	
7	_=	_=	ists	_=	__	_:=	_find	.	B	
6	_=	_=	ists	_=	_new	_:=	_find	.	B	
5	_=	_=	ists	_=	_t	_=	_str	.	get	
4	_=	_=	ists	_=	_"	_=	_str	.	get	
3	_	_=	ists	_=	_"	_	_new	.	get	
2	.	(	ists	_=	_"	_in	_new	.	get	
1	.	(	ists	()	_0	_in	_new	.	get	
input	uster	Var	Ex	ist	,	_err	_:=	_meta	.	Does

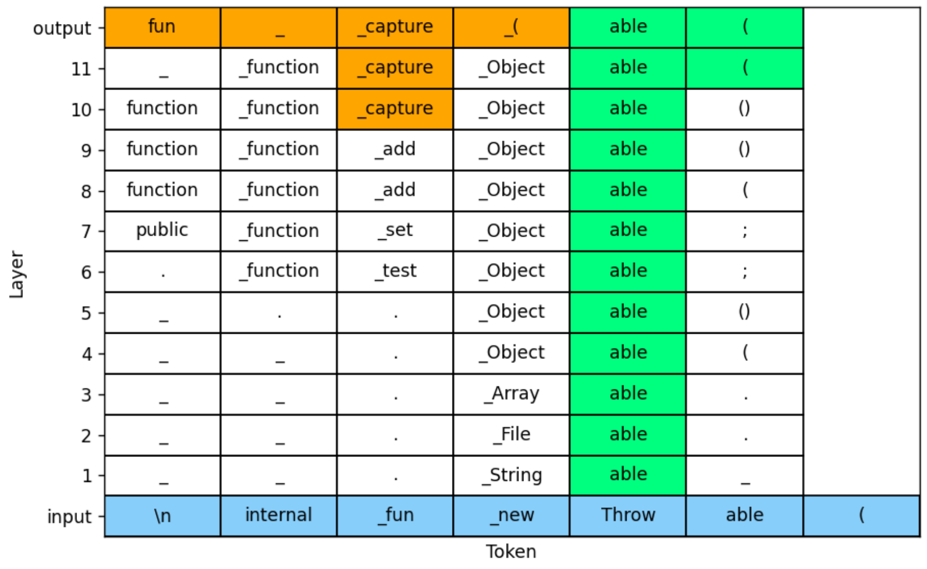
Figure 14: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

A.5 Kotlin



```
internal_fun_newThrowable(message:_String?,_cause:_Throwable?):_Throwable_{\n
____val_throwable_=_js("new_Error()")\n
____throwable.message_=_message_?:_cause?.toString()_?:_undefined\n
____throwable.cause_=_cause\n
____throwable.name_=_"Throwable"\n
____return_throwable.unsafeCast<Throwable>()\n
```

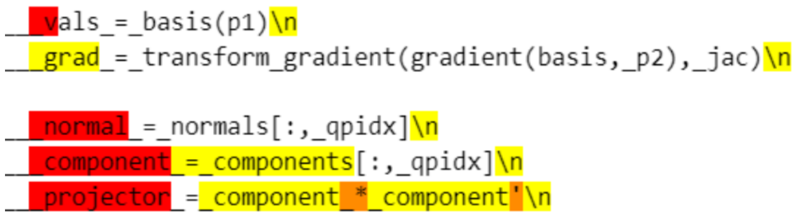
Figure 15: Kotlin code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).



Layer							
output	fun	_	_capture	_(	able	(	
11	_	_function	_capture	_Object	able	(	
10	function	_function	_capture	_Object	able	()	
9	function	_function	_add	_Object	able	()	
8	function	_function	_add	_Object	able	(	
7	public	_function	_set	_Object	able	;	
6	.	_function	_test	_Object	able	;	
5	_	.	.	_Object	able	()	
4	_	_	.	_Object	able	(	
3	_	_	.	_Array	able	.	
2	_	_	.	_File	able	.	
1	_	_	.	_String	able	_	
input	\n	internal	_fun	_new	Throw	able	(
	Token						

Figure 16: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

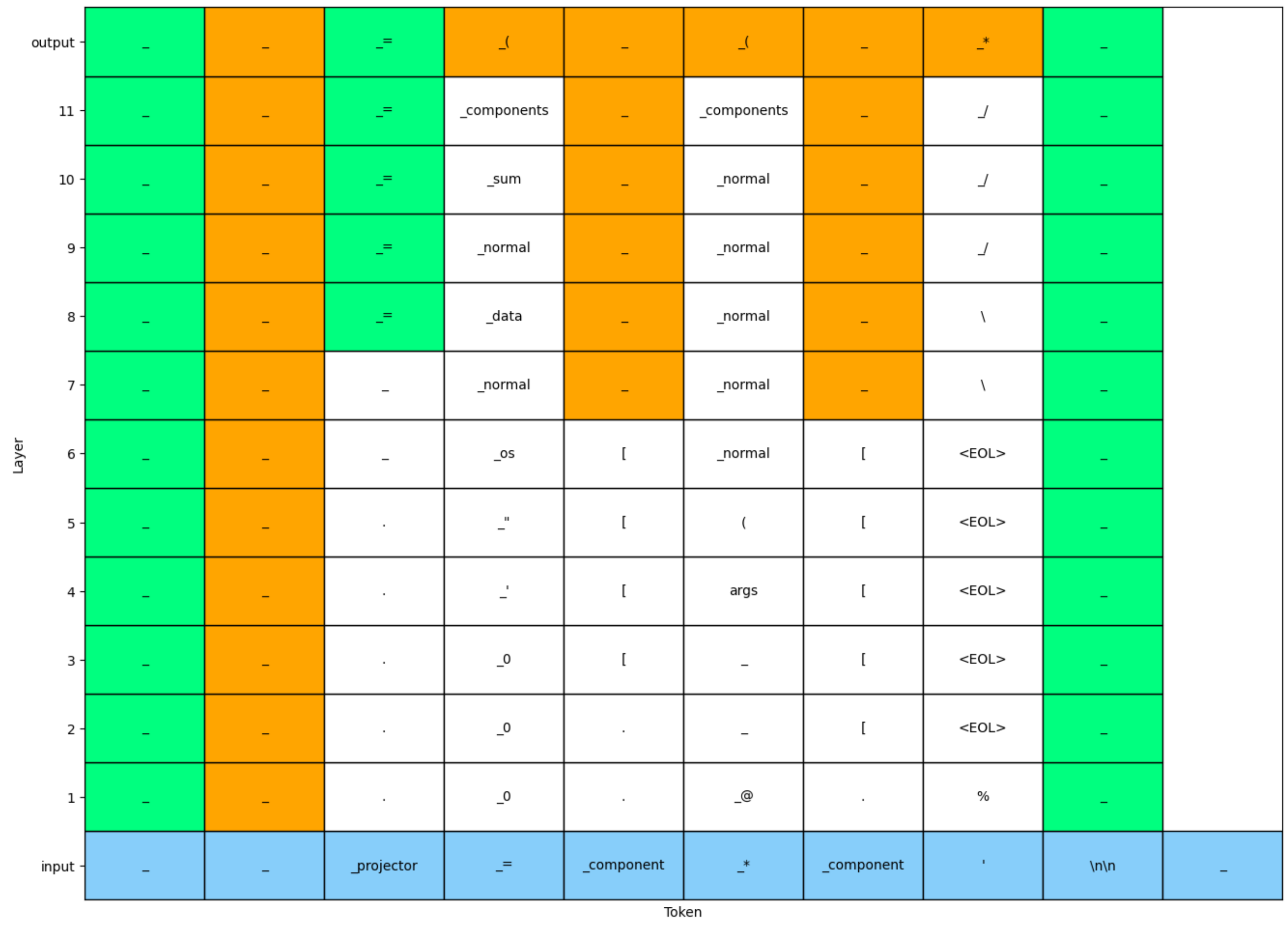
A.6 Julia



```
___vals_=_basis(p1)\n
___grad_=_transform_gradient(gradient(basis,_p2),_jac)\n

___normal_=_normals[:,_qpidx]\n
___component_=_components[:,_qpidx]\n
___projector_=_component_*_component'\n
```

Figure 17: Julia code snippet containing the input sequence fed to the *tuned lens*, for which the results are displayed below. All *incorrectly* predicted tokens are highlighted using a color indicating the depth (or confidence level) of the model. Here, bright red corresponds to the highest confidence level (lowest depth and worst), whereas yellow signifies the lowest confidence level (highest depth).



Layer										
output	_	_	_=	_(	_	_(	_	_*	_	
11	_	_	_=	_components	_	_components	_	_/	_	
10	_	_	_=	_sum	_	_normal	_	_/	_	
9	_	_	_=	_normal	_	_normal	_	_/	_	
8	_	_	_=	_data	_	_normal	_	\	_	
7	_	_	_	_normal	_	_normal	_	\	_	
6	_	_	_	_os	[	_normal	[	<EOL>	_	
5	_	_	.	_"	[	(	[	<EOL>	_	
4	_	_	.	_'	[	args	[	<EOL>	_	
3	_	_	.	_0	[	_	[	<EOL>	_	
2	_	_	.	_0	.	_	[	<EOL>	_	
1	_	_	.	_0	.	_@	.	%	_	
input	_	_	_projector	_=	_component	_*	_component	'	\n\n	_
	Token									

Figure 18: Layer-by-layer visualization of *tuned lens* output. Green and orange colors are used to indicate depths of correct and false token predictions respectively. Note that the ground truth (i.e. the next input token to predict) can be obtained by looking at the blue cell one column to the right.

B Null Attention Ratios for Best and Worst Tokens

For all of the six investigated languages (*Java, *Python, C++, Kotlin, *Go, Julia), we have randomly sampled attention heads across 60 distinct source files - corresponding to the best and worst predicted tokens as presented in Table 3.

Over these attention heads, we have calculated the mean and SD of null attention ratios across layers 6 up until 12 [21]. The maximum and minimum values are in boldprint.

As explained in Section 3.4, these null attention ratios denote the proportion of tokens (out of all 1024 input tokens) that assign an attention score of at least 0.9 to the first token in given input sequence. As the first token is attended to by default, null attention is not deemed informative.

B.1 *Java

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
’get’	10.4	9.7	’}’	8.4	8.6
’;’	8.9	8.3	’public’	11.7	9.2
’<’	9.6	11.9	’return’	8.5	7.8
’als’	5.6	6.5	’if’	10.8	7.2
’\\’	**2.73**	**1.44**	’@’	11.12	9.14
’Exception’	6.9	5.7	’VK’	**22.0**	**15.8**
’String’	11.7	8.5	’private’	6.1	4.8
’or’	8.7	9.0	’assert’	15.4	12.2
’of’	16.8	10.8	’this’	9.6	10.7
’adata’	13.7	9.5	’{’	7.0	7.7
Average	9.5	8.1	**Average**	11.0	9.3

Table 4: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in Java. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

B.2 *Python

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
’self’	8.6	9.2	’def’	7.6	7.8
’s’	7.7	7.9	’1’	-	-
’get’	5.8	5.7	’class’	5.3	5.6
’0’	-	-	’:’	4.3	4.8
’1’	-	-	’=’	12.5	11.3
’if’	-	-	’if’	-	-
’n’	6.2	5.8	’@’	10.1	9.1
” ’ ”	6.3	5.0	’[’	11.5	11.0
’format’	**15.4**	**13.8**	’2’	**4.4**	**4.1**
’args’	8.5	9.3	’0’	-	-
Average	8.4	8.1	**Average**	8.0	7.7

Table 5: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in Python. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

[21] As mentioned earlier in Section 3.4, the first five model layers are roughly uniform across languages.

B.3 C++

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
’;’	5.9	6.3	’}’	11.4	9.3
’x’	**3.67**	5.07	’if’	5.8	7.1
’get’	9.9	6.8	’{’	9.6	13.8
’n’	5.0	5.0	’->’	7.3	6.5
’0’	9.8	7.8	’return’	13.8	14.3
’1’	5.22	**4.0**	’#’	9.2	8.3
’s’	12.0	15.8	’::’	13.1	**19.4**
’int’	10.6	7.67	’case’	10.1	9.1
’or’	13.5	9.7	’void’	5.1	4.9
’r’	9.9	9.1	’else’	**15.7**	15.0
Average	8.5	7.7	**Average**	10.1	10.8

Table 6: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in C++. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

B.4 *Go

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
’x’	5.1	5.1	’}’	8.4	6.4
’s’	12.2	12.4	’return’	9.3	9.9
’qual’	8.0	9.7	’if’	7.8	6.3
’0’	9.4	10.4	’t’	**4.9**	**4.1**
’pace’	11.2	**13.6**	’{’	6.0	4.7
’n’	9.4	6.1	’case’	12.9	9.7
’ator’	9.9	8.5	’ ” ’	11.2	9.9
’point’	**13.5**	12.9	’for’	4.9	4.4
’1’	5.3	5.7	’log’	9.0	9.7
’els’	5.3	4.0	’var’	7.5	7.4
Average	8.9	8.8	**Average**	8.2	7.3

Table 7: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in Go. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

B.5 Kotlin

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
'get'	6.1	5.3	'}'	**3.6**	**3.2**
'or'	7.4	7.8	':'	6.2	6.3
'ment'	7.3	9.6	'?'	6.2	7.1
'als'	10.1	8.9	'['	10.4	9.3
'ript'	**12.1**	**14.7**	'@'	8.6	9.5
'<'	-	-	'!!'	5.7	4.4
'able'	9.8	3.8	'<'	-	-
'n'	10.1	11.1	'is'	6.6	4.3
'son'	7.8	9.4	'java'	7.2	5.8
'ert'	7.0	7.3	'to'	8.0	8.1
Average	8.6	8.7	**Average**	6.9	6.4

Table 8: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in Kotlin. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

B.6 Julia

Top-10 Best Tokens	Mean (%)	SD	Top-10 Worst Tokens	Mean (%)	SD
'1'	-	-	'end'	8.6	10.2
'0'	10.8	12.3	'1'	-	-
'2'	-	-	'2'	-	-
's'	**11.0**	10.0	'::'	9.7	11.8
'mbol'	6.6	7.5	'!'	9.2	**14.0**
'ices'	6.5	6.4	':'	6.3	7.7
'['	-	-	'['	-	-
'n'	**6.3**	**5.6**	'{'	7.6	8.9
']'	-	-	'function'	10.4	9.6
'estamp'	9.9	12.2	']'	-	-
Average	8.5	9.0	**Average**	8.6	10.4

Table 9: Mean (%) and Standard Deviation (SD) of null attention ratios across layers 6-12, for the top-10 best and worst predicted tokens in Julia. Tokens are categorized into user-defined elements (transparent), language-defined elements (yellow), common code structures (blue), and punctuation (orange).

References

1. Radford, A., Narasimhan, K., Salimans, T. & Sutskever, I. Improving language understandinsg by generative pre-training (2018).
2. Feng, Z. *et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages* in (Jan. 2020), 1536–1547.
3. Wan, Y. *et al. What Do They Capture? – A Structural Analysis of Pre-Trained Language Models for Source Code* 2022. arXiv: 2202.06840 [cs.SE].
4. Chen, N. *et al. CAT-probing: A Metric-based Approach to Interpret How Pre-trained Models for Programming Language Attend Code Structure* 2022. arXiv: 2210.04633 [cs.SE].
5. Chen, M. *et al. Evaluating Large Language Models Trained on Code* 2021. arXiv: 2107.03374 [cs.LG].
6. Guo, D. *et al. GraphCodeBERT: Pre-training Code Representations with Data Flow* 2021. arXiv: 2009.08366 [cs.SE].
7. Lu, S. *et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation* 2021. arXiv: 2102.04664 [cs.SE].
8. Wang, X. *et al. Compilable Neural Code Generation with Compiler Feedback* 2022. arXiv: 2203.05132 [cs.CL].
9. Svyatkovskiy, A., Zhao, Y., Fu, S. & Sundaresan, N. Pythia: AI-assisted Code Completion System. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019).
10. Svyatkovskiy, A., Deng, S. K., Fu, S. & Sundaresan, N. IntelliCode compose: Code generation using Transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020).
11. Vaswani, A. *et al. Attention Is All You Need* 2017. arXiv: 1706.03762 [cs.CL].
12. Peters, M. E. *et al.* Deep contextualized word representations. *CoRR.* arXiv: 1802.05365. http://arxiv.org/abs/1802.05365 (2018).
13. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* 2019. arXiv: 1810.04805 [cs.CL].
14. Radford, A. *et al.* Language models are unsupervised multitask learners. *OpenAI blog,* 9 (2019).
15. Alon, U., Brody, S., Levy, O. & Yahav, E. *code2seq: Generating Sequences from Structured Representations of Code* 2019. arXiv: 1808.01400 [cs.LG].
16. Xie, Y., Lin, J., Dong, H., Zhang, L. & Wu, Z. *A Survey of Deep Code Search* 2023. arXiv: 2305.05959 [cs.SE].
17. Raychev, V., Vechev, M. & Yahav, E. *Code completion with statistical language models* in *Acm Sigplan Notices* (2014), 419–428.
18. Liu, Y. *et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach* 2019. arXiv: 1907.11692 [cs.CL].
19. Kanade, A., Maniatis, P., Balakrishnan, G. & Shi, K. *Learning and Evaluating Contextual Embedding of Source Code* 2020. arXiv: 2001.00059 [cs.SE].
20. Feng, Z. *et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages* 2020. arXiv: 2002.08155 [cs.CL].
21. Izadi, M., Gismondi, R. & Gousios, G. *CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences* in *Proceedings of the 44th International Conference on Software Engineering* (Association for Computing Machinery, Pittsburgh, Pennsylvania, 2022), 401–412. ISBN: 9781450392211. https://doi.org/10.1145/3510003.3510172.
22. Guo, D. *et al. UniXcoder: Unified Cross-Modal Pre-training for Code Representation* 2022. arXiv: 2203.03850 [cs.CL].
23. Vig, J. & Belinkov, Y. *Analyzing the Structure of Attention in a Transformer Language Model* 2019. arXiv: 1906.04284 [cs.CL].
24. Bostrom, K. & Durrett, G. *Byte Pair Encoding is Suboptimal for Language Model Pretraining* 2020. arXiv: 2004.03720 [cs.CL].
25. Belinkov, Y. & Glass, J. *Analysis Methods in Neural Language Processing: A Survey* 2019. arXiv: 1812.08951 [cs.CL].
26. Bahdanau, D., Cho, K. & Bengio, Y. *Neural Machine Translation by Jointly Learning to Align and Translate* 2016. arXiv: 1409.0473 [cs.CL].
27. Xiong, R. *et al. On Layer Normalization in the Transformer Architecture* 2020. arXiv: 2002.04745 [cs.LG].
28. Husain, H., Wu, H.-H., Gazit, T., Allamanis, M. & Brockschmidt, M. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search* 2020. arXiv: 1909.09436 [cs.LG].
29. Belrose, N. *et al. Eliciting Latent Predictions from Transformers with the Tuned Lens* 2023. arXiv: 2303.08112 [cs.LG].
30. Jain, S. & Wallace, B. C. *Attention is not Explanation* 2019. arXiv: 1902.10186 [cs.CL].
31. Vig, J. *A Multiscale Visualization of Attention in the Transformer Model* in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (Association for Computational Linguistics, Florence, Italy, July 2019), 37–42. https://www.aclweb.org/anthology/P19-3007.
32. Kocetkov, D. *et al.* The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).
33. Park, Y., Kim, K. & Jang, D.-k. *The batch size can affect inference results* 2023. https://openreview.net/forum?id=9MDjKb9lGi.
34. Halawi, D., Denain, J.-S. & Steinhardt, J. *Overthinking the Truth: Understanding how Language Models process False Demonstrations* 2023. https://openreview.net/forum?id=em4xg1Gvxa.

35. Karampatsis, R., Babii, H., Robbes, R., Sutton, C. & Janes, A. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. *CoRR*. arXiv: 2003.07914. https://arxiv.org/abs/2003.07914 (2020).

36. Kim, S., Zhao, J., Tian, Y. & Chandra, S. *Code Prediction by Feeding Trees to Transformers* 2021. arXiv: 2003.13848 [cs.SE].

37. Li, J., Wang, Y., Lyu, M. R. & King, I. *Code Completion with Neural Attention and Pointer Networks* in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (International Joint Conferences on Artificial Intelligence Organization, July 2018). https://doi.org/10.24963%2Fijcai.2018%2F578.