Erik Mekkes
Mekkes.Erik@gmail.com
e.j.mekkes@student.tudelft.nl

Evaluating Large Language Model Performance on User and Language Defined Elements in Code.

Supervisors: Jonathan Katzy, Maliheh Izadi   Responsible Professor: Arie van Deursen   Examiner: Azqa Nadeem

Code on GitHub:

Datasets on HuggingFace:

Interest in Large Language Models (LLM) of code has led to many new models and innovations in the evaluation of LLMs[1] and their ability to learn the structural syntax of code[2]. However, a trend to reach new levels of performance is to rely on scaling up in model size and training data[3]. This trend is accompanied by significantly increased costs and loss of accessibility, and might at some point reach its limits.

We should look at other avenues of improving performance to mitigate this trend, as such we apply recently developed techniques to inspect the internal state of LLMs and contribute a method to evaluate specific aspects of Programming Languages that relates performance to their dimension size.

Research Question: How do user-defined elements compare to language-defined elements with regard to the depth of the first correct prediction?

Hypothesis:
Language-defined elements require significantly fewer layers due to their well-defined structure and meaning.

Background: Transformer Dimensions

Transformers improve on Recurrent Neural Network architectures with the self-attention mechanism, adding parallel processing of the positional meaning of tokens in an input[4]. This results in a fixed maximum Input Sequence Length.

Fixed dimensions in transformers ensure matching inputs and outputs that allow blocks of operations to be repeated. Current models apply layered Encoder and Decoder block architectures. We focus on the Decoder type as these are best for tasks that involve next-token predictions. This is a relevant topic to us due to its application in Code-Completion.

The number of layers and maximum input length dimensions are key design choices for a Model as they are trade-offs between computational cost and the quality of predictions: We are interested in how much each layer and its contents contributes to accuracy.

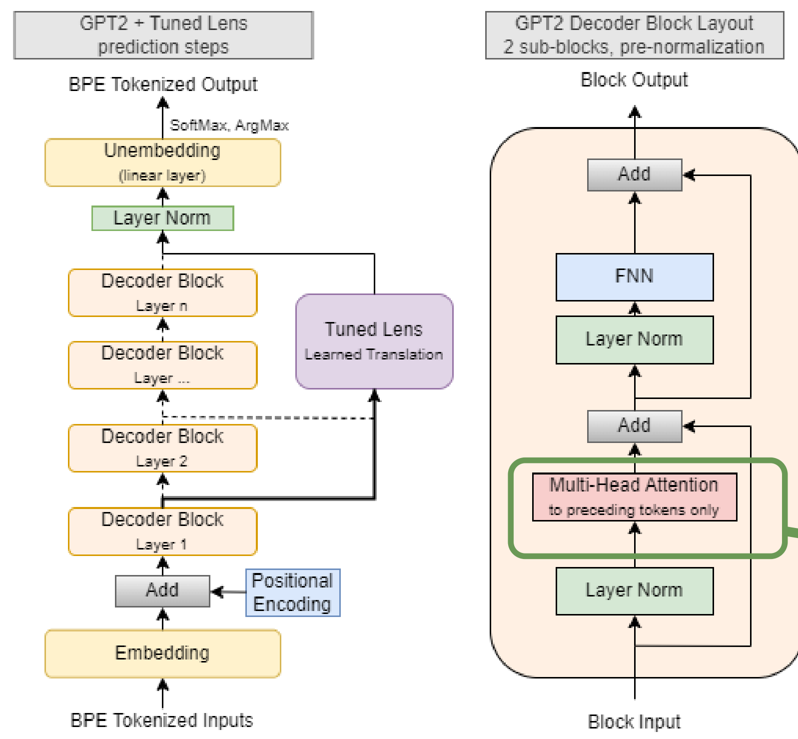

Figure 1: GPT2 Architecture with Tuned Lens Translation

Recent Development: Tuned Lens Translation

Internal layer states do not directly translate to outputs that we can evaluate due to shifts in representation between layers, the model is only required to return a meaningful representation in the last layer.

A Tuned Lens[6] is an early-exiting technique with a learned translation. This translation is trained on minimizing the Kullback–Leibler (KL) divergence between the final layer output state and for any layer x:

$$state_layer_x * learned_change_of_basis + learned_bias$$

Applying this learned translation returns a representation that allows accurate inspection of what the model is predicting in internal layers. This is illustrated on the left side of Figure 1 below.

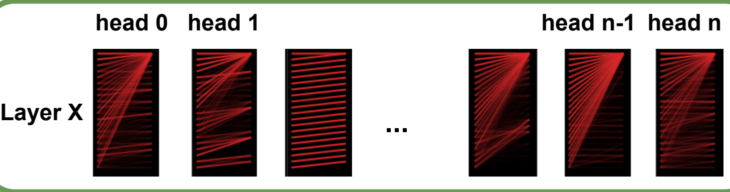

Figure 2: Multi-Head Attention in a Decoder Subblock using BertViz

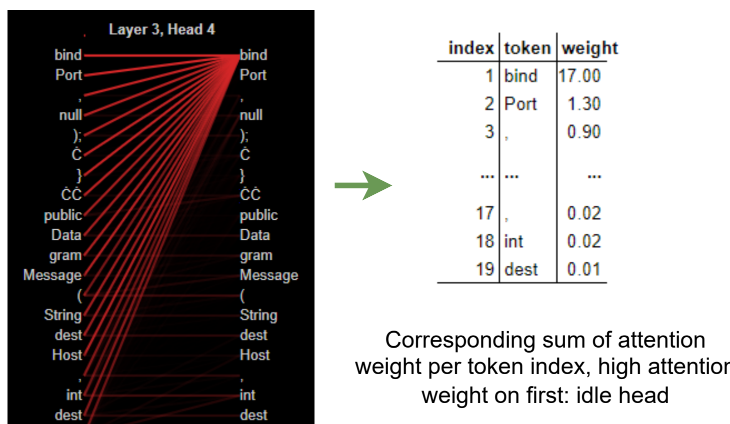

Figure 3: Null Attention in a PolyCoder Attention Head using BertViz

Recent Discovery: (Null) Attention Patterns in Attention Heads

Each layer within the GPT2 Transformer architecture makes use of a multi-head attention sub-block as depicted on the right of Figure 1. This is another development facilitated by adding positional encoding.

For a sequence with n input tokens, each head constructs an n x n matrix of attention weights that signifies the relevance of each token to the current attention head.

It has been shown that using such attention weights leads specific attention heads to form patterns that relate to structural aspects of language[7]. Multiple head attention allows the model to focus on separate aspects of code language simultaneously during each layer.

It has also been shown that a concept of null attention exists, which is defined as heads defaulting to placing attention weight on the first token when no tokens in the sequence are relevant to a head. An example of this occurring is shown to the left in Figure 3.

Selected Model, Evaluation Dataset, and Pre-Processing Steps to Prepare Samples

We select the 400M parameter PolyCoder [1] version as to evaluate, a GPT2-based, decoder-only transformer with 24 layers and 16 attention heads.

PolyCoder's internal dimensions support up to 1024 tokens, which is also the sequence length it is trained on. We use 1024 length sequences as input for ideal predictions, making 1000 predictions per sample.

PolyCoder Training : 24M files

Language	Repositories	Files
Java	15,044	5,120,129
C++	13,726	4,289,506
Python	25,446	1,550,208
Go	12,371	1,416,789
Julia	-	-
Kotlin	-	-

Remaining 12M files across: C, C#, JavaScript, PHP, TypeScript, Ruby, Rust, Scala

We wish to evaluate performance outside it's training set as well !

The Stack - Deduplicated[5]

Scala | php | many more...

Python | C++ | Julia | Java | Go | Kotlin

Our evaluation dataset: subset of 100k random files from these 6 code languages.

Sample File
```
"""
Docstring
"""
var_1 = 1
# Comment
if var_1: exit()
...
```
→ Remove Comments and Docstrings →

Stripped Sample
```
var_1 = 1
if var_1: exit()
if var_2:
  ...
if var_n:
  for x in items
  ...
```
→ BPE Tokenize →

Tokenized Sample
```
828,65,19,260,352,
285,925,65,19,28,4322,336,
285,925,65,20,28,
...
285,925,65,80,28,
209,446,754,353,5384,
209,209,285,828,65,19,260,
...
```
→ Select only files >= 2024 tokens / Take Random Subsection of 2024 tokens →

PreProcessed Sample
```
...
285,925,65,80,28,
209,446,754,353,5384,
209,209,285,828,65,19,260,
...
```
Always guaranteed to have 2024 token length

Definition of Metrics

Prediction Accuracy per layer : Average Depth of First Correct Prediction

To compare performance for token types across layers we are particularly interested in how early a model is able to predict correctly, as this relates to size and computational cost.

The earliest correct prediction alone is not a great indicator, the model is likely to switch between choices while the probability is low. We define Depth of First Correct Prediction as the first layer in which a token was predicted correctly without it changing in later layers as shown below

Input Sequence:	...	for	x	in	items	...
Token Type:	...	L	U	L	U	...
Layer 1:	...	0	0	in	0	...
Layer 2:	...	for	0	0	items	...
Layer 3:	...	for	x	in	0	...
Layer 4:	...	for	x	in	0	...
Layer 5:	...	for	x	in	items	...
Layer 6:	...	for	x	in	items	...

Figure 3: Earliest success without later loss of confidence.

	...	for	x	in	items	average
User 1st correct:	...	-	3	-	5	4
Lang 1st correct:	...	2	-	3	-	2.5

Figure 4: Depth of First Correct Prediction

Defining and labelling User- and Language-Defined Elements

User and Language Defined Elements: Labelling Token Types

Language-defined elements refer to Keywords, Operators and Separators that are reserved by the language, these elements have a very specific immutable meaning within a language. We consistently show these as green in the results.

User-defined elements refer to identifiers, words chosen by the writer to refer to elements like functions or variables. These have no pre-defined form or function and ore often new to the model. We consistently show these as yellow in results.

We hypothesized that if these distinctions between user and language elements exist, they should also be present in the internal representation of LLMs and should have different performance characteristics that we can observe to learn about model behavior and possibly use to our advantage.

Language	Lang. Defined	Keyword	User Defined
Java	42.357	9.317	62.171
CPP	42.263	6.985	82.026
Python	32.039	5.646	81.003
Go	57.001	6.579	96.392
Julia	43.369	4.762	91.094
Kotlin	32.953	4.468	56.212

Element Type Distribution, 200k Tokens per Language

We label language-defined elements according to the official specification of each language, remaining elements are labelled as user-defined..

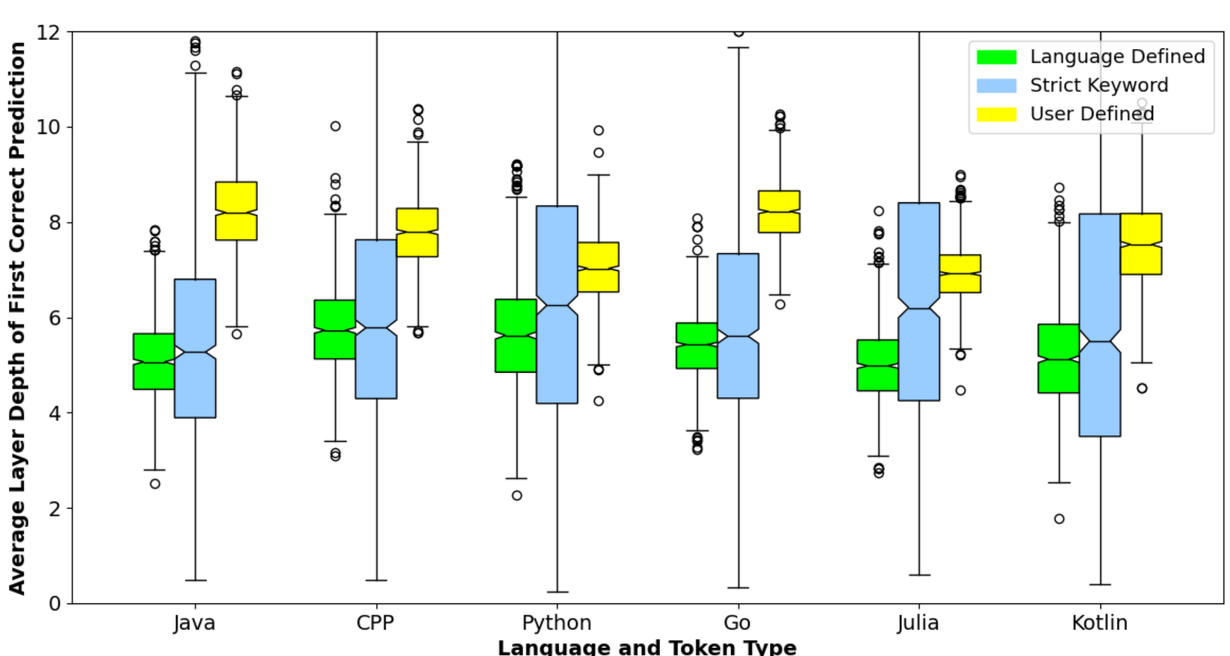

Figure 5: Distribution of Average Depth of First Correct Predictions Across Language and Element Types

Results of Evaluating Prediction Performance

We compute the Depth of Average First Completion for 200k Token predictions per language and plot the distribution as a box plot per Language and Element type in Figure 5. As Operator and Separator tokens occur frequently and consistently and skew the results, we also plot the Keyword group separately.

We immediately see a clear and well-defined 2-layer advantage for language-defined elements. We note an increased variability in Strict Keyword Elements (blue, fewer occurrences in samples) and strong fluctuation in user-defined element performance.

To look for explanations, we look at the relative performance of specific tokens of interest.. For language-defined elements, we select common keywords defined for all languages, for user-defined elements we select common variable names.

We plot the performance of 8 Tokens of Interest in Go and Java below in Figure 6 and 7. The results for individual tokens confirm our element group results. We find similar medians in all languages but with inconsistency due to token sample sizes, language characteristics, and token classification accuracy.

We select the 'else' and 'start' tokens for further attention mechanism analysis, they have similar sample occurrences and low variability which gives us a fair comparison of their attention behavior.

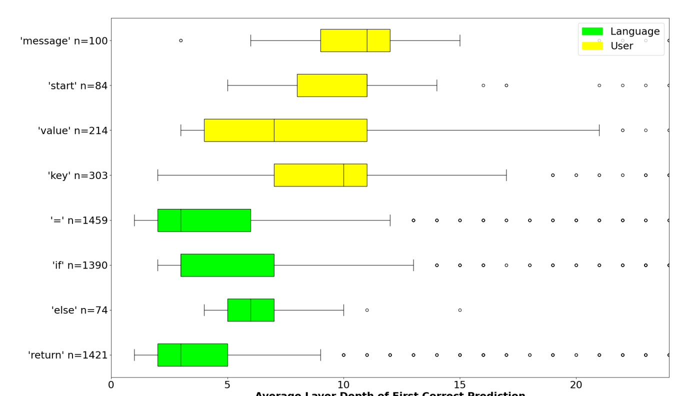

Figure 6: Prediction Performance for Tokens of Interest in Go

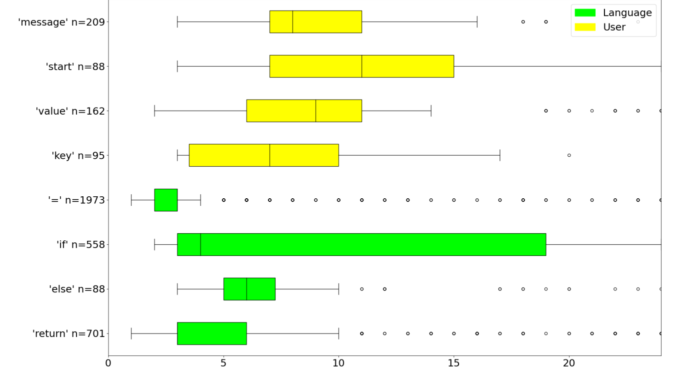

Figure 7: Prediction Performance for Tokens of Interest in Java

Results of Evaluating the Focus of the Attention Mechanism

We use summed weight on the first token to define when a head is idle. This distribution of weight is challenging for higher input sequence sizes, we therefore only classify a top 5% first token summed weight as a null head. We show the fraction of Null Attention Heads per layer for 'else' (language-defined) and 'start' (user-defined) tokens for Java (Figure 8) and Go (Figure 9). We used an equal number of samples (n=60) of each.

The results show that a greater fraction of heads (+7\%, 2-3 fold increase) is unable to contribute to later layers of user-defined element predictions. A loss of contributing heads in higher layers relates to less early predictions. However, this is only a partial explanation as neither element sees null attention in the earlier layers where we identified a high degree of accuracy progression.

We consider the absence of earlier layer null heads as evidence that the earlier layer heads have a more general purpose: they see relevance in either type. From the performance difference, we infer that these more general-purpose heads must be significantly better at resolving language-defined elements.

We see a very clear 2 layer performance advantage for Language-Defined elements over User-Defined elements. This difference is consistent across languages, even outside the training set.

This technique of observation can be further refined by checking performance with comments in code and by improving the token classification methods.

For specific high-occurrence language-defined Tokens such as Operators, we observe an even greater difference in performance. If such a token type can be identified early, this can allow early exiting techniques to gain performance without loss of significant accuracy.

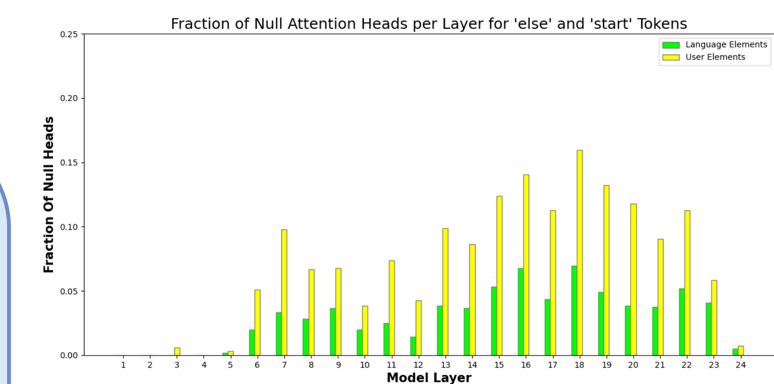

Figure 8: Fraction of Null Attention heads in Go per Layer

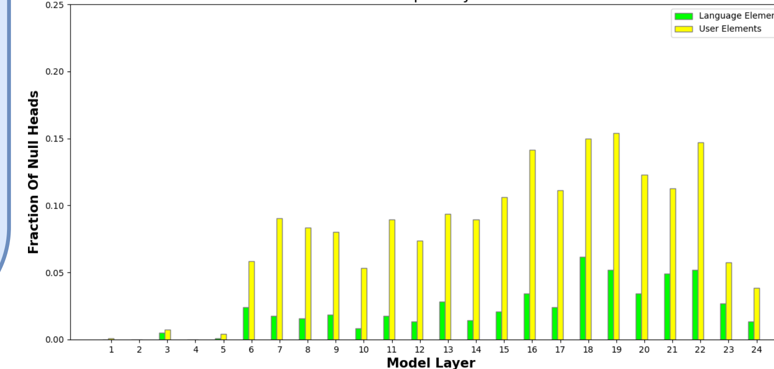

Figure 9: Fraction of Null Attention heads in Java per Layer

Faculty of EEMCS - BSc Computer Science and Engineering

Sources:
[1] Xu, 2022, A Systematic Evaluation of Large Language Models of Code
[2] Wan, 2022, What Do They Capture? -- A Structural Analysis of Pre-Trained Language Models for Source Code
[3] Hellendoorn, 2021, The growing cost of deep learning for source code
[4] Vaswani, 2017, Attention Is All You Need
[5] Kocetkov, 2022, The Stack: 3 TB of permissively licensed source code
[6] Belrose, 2023, Eliciting latent predictions from transformers with the tuned lens
[7] Vig, 2019, Analyzing the Structure of Attention in a Transformer Language Model