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

Erik Mekkes

Supervisors: Arie van Deursen, Maliheh Izadi, Jonathan Katzy

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Erik Mekkes
Final project course: CSE3000 Research Project
Thesis committee: Arie van Deursen, Maliheh Izadi, Jonathan Katzy, Azqa Nadeem

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Large Language Models of code have seen significant jumps in performance recently. However, these jumps tend to accompany a notable and perhaps concerning increase in scale and costs. We contribute an evaluation of prediction performance with respect to model size by assessing the layer-wise progression for language and user-defined elements in code, using a new technique of Tuned Lenses. We show that language-defined elements can be predicted more accurately in earlier layers of the PolyCoder model than user-defined elements and contribute an evaluation of the attention mechanism, which shows patterns that explain such aspects of performance and indicate areas of missed potential. These findings encourage research into the internal prediction performance for other characteristic aspects of code and could lead to the introduction of new methods that make use of these characteristics to improve performance without relying on scaling.

1 Introduction

Interest in Large Language Models (LLM) of code has led to many new models and innovations in the evaluation of LLMs [21] and their ability to learn the structural syntax of code [20]. However, a trend to reach new levels of performance is to rely on scaling up in model size and training data [6]. This trend is accompanied by significantly increased costs and loss of accessibility, and might at some point reach its limits, much like Moore's Law [9]. Like the efforts of semiconductor manufacturers, we should look at other avenues of improving performance to mitigate this trend and apply new evaluation methods to enable this.

Recent work has shown deficiencies in match-based metrics regarding the evaluation of LLM performance on code, which has led to a switch to evaluate functional correctness [14]. The development of CodeBLEU was motivated by such issues with capturing semantic features specific to code [4].

Of specific interest to this paper is the recent introduction of new methods to inspect the internal state of LLMs through lenses [3, 13]. These enable new research into the internals of the now prevalent transformer architecture [17] and inspire us to develop a generally applicable pipeline to evaluate the performance of such LLMs. We use the new Tuned Lens method to look at characteristic aspects of code language, and how they relate to model performance.

We specifically aim to answer the following question: How do user-defined elements compare to language-defined elements with regard to the depth of the first correct prediction?

The Hypothesis is that language-defined elements require fewer layers due to their immutable structure and meaning.

We contribute an application of the Tuned Lens on a large scale of multi-language inputs for the medium-sized version of PolyCoder [21], a GPT2-based transformer model. A Tuned Lens can return an accurate representation of the prediction state at each internal layer of a LLM, allowing us to evaluate the progression of accuracy throughout the model. We present a method to distinguish between token types and apply a first correct layer metric on Lens outputs to relate accuracy to layer depth. Using these results we evaluate specific tokens and correlate this with earlier results and overall prediction accuracy to find that language-defined elements outperform user-defined elements by a significant depth of 2 layers.

Finally, we contribute an evaluation of the patterns formed by the attention heads of the LLM. Recent work indicates patterns of null attention, a non-relevance indicator inside attention heads [19]. We apply this concept of null attention to complete heads to identify patterns of non-contributing heads within layers. Comparing this statistic for both element types we find an increased fraction of heads are idle for both element types in later layers, and infer that earlier layers without idle heads must be contributing a greater degree of accuracy. Our results show a significant increase in non-contributing heads for user-defined elements, indicating that the heads are more limited in their ability to infer the relevance of this token type.

These evaluations reveal areas of interest for future research on model accuracy and computational efficiency, as they identify specific characteristics that LLMs of code perform well on and aspects that need additional resources. Building on these understandings could improve model performance without relying on scaling, to mitigate the trend towards larger and larger models and the subsequent costs and accessibility issues involved.

2 Background and Related Work

As background for our work, we highlight the aspects of modeling tasks and code languages involved in our research question. We also discuss the Transformer Model, Lens developments, and subject model that we base our research on as an introduction to our contributions.

Prediction Task Relevance LLMs can be applied to many task types: analytic tasks such as error detection and correction, descriptive tasks like annotation or summarising, or generative tasks like auto-completion. We focus exclusively on auto-completion performance, also known as code synthesis/generation/inference, where the next element has to be suggested based on the inputs seen so far. The applications of this task are in speeding up code writing tasks and creating cleaner code, provided that the predictions are of sufficient quality and arrive in time.

This subject is relevant and challenging, code completion is widely used by developers [12] and previously required hybrid solutions to meet such latency and validity requirements [2]. This introduces a strong motivation for investigating new aspects that affect how the correctness of predictions relates to the computational resources used.

Significance of User and Language Elements We focus our research question on a specific aspect of programming languages, distinct user and language-defined elements:

Language-defined elements refer to keywords that are reserved by the language, these elements have a very specific immutable meaning within a language, but the vocabulary and exact meaning might vary between languages. This differs from natural language where a word might have different interpretations based on context within a language.

User-defined elements refer to identifiers, words chosen by the writer to refer to elements like functions or variables. Their meaning and composition might relate to the context, but can also vary wildly depending on the whims of the author. These differ from regular language due to the extent to which code documents introduce original identifier names that have never appeared before in the text or even in other projects [1], this also implies the element is not in the learned vocabulary of the model.

As it has been shown that LLMs have the ability to learn the syntax structure of code [20], we hypothesize that knowing these distinctions between user and language elements exist, they should also be present in the internal representations of LLMs and should have different performance characteristics that can be used to our advantage.

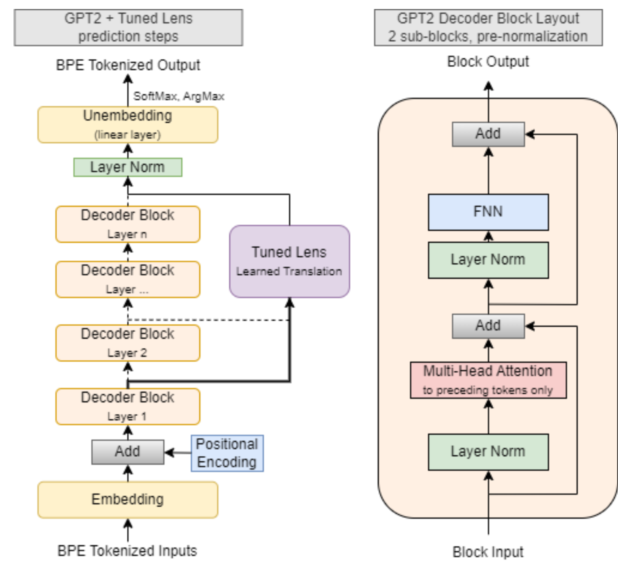

Figure 1: GPT2 Architecture With Tuned Lens translation

2.1 Transformer Model Architecture

As we wish to make a contribution to the current state of the art, we evaluate Models within the Transformer Architecture. We discuss key model design choices, their effect on computational cost, and relation to our research question.

Transformers improve on the Recurrent Neural Network architectures by means of the self-attention mechanism, which allows for parallel processing of the positional meaning of tokens in an input sequence [17], as each internal component operates on a similar dimension of inputs. This parallelization comes at the cost of having a fixed maximum input length. Choosing this internal dimension is a key design choice when creating a model as it is a trade-off between computational cost and the quality of predictions due to the amount of available context.

Matching dimensions ensure matching inputs and outputs, which allows for blocks of operations that can be stacked repeatedly on top of each other. Current models apply two types of block architectures: Encoders and Decoders. We focus exclusively on the Decoder architecture type, as these are most suitable for tasks that only involve inference [10]. The current state-of-the-art decoder-only LLMs available and computationally reasonable for our resources are GPT2 based, for which the architecture can be seen in Figure 1.

The architecture shows the application of multiple such decoder blocks to increase accuracy at the cost of increased computational resources. We refer to these decoder blocks as layers, which introduces the final element of our research question, a comparison based on layer depth that relates accuracy to required computational resources.

2.2 Tuned Lens Inspection

We analyze the state of predictions as they progress through the Layers of a transformer model. To get a meaningful interpretation of this state we make use of the Tuned Lens technique, a recent improvement on Logit Lenses [13].

The Logit Lens is an early exiting technique of applying the learned Unembedding directly on earlier layer outputs to attempt to map them back to the token vocabulary. This technique results in unreliable interpretations due to representational drift at different layers [3]. A trained Transformer Model produces a representation in its final layer that maps back to the input vocabulary when the Unembedding is applied to it, but this is not necessarily the case for earlier layers.

The Tuned Lens improves on this technique with an intermediate translation step. A transformation involving a learned bias and a learned change of basis is trained for each layer of the model, which maps from the output space of that layer to the output space of the final layer of the model. The training minimizes the Kullback–Leibler (KL) divergence between the final layer state and $layer_i_state * learned_change_of_basis + learned_bias$. Once learned, applying this translation to a state from $layer_i$ returns a representation in the final layer, which makes it suitable to apply a final layer norm and the Unembedding to get an improved result. This process is illustrated on the left side of Figure 1.

2.3 The Attention Mechanism

Each layer within the GPT2 Transformer architecture makes use of a multi-head attention sub-block as depicted on the right of Figure 1. For the embedded input tokens, each attention head is a neural network that learns Query, Key, and Value linear projections [11], facilitated by the applied positional encoding [17]. The dot product of Query and Key projections is a measure of similarity between tokens, the output of which is scaled and normalized to produce what are called attention weights. A sequence with n input tokens results in an $n \times n$ matrix of attention weights that signifies the relevance of tokens to the current attention head.

It has been shown that using such attention weights leads specific attention heads to form patterns that relate to struc-

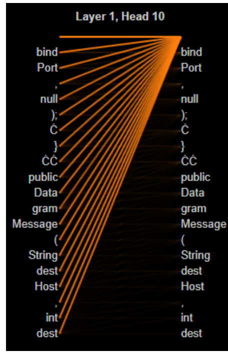

Figure 2: Concept of Null Attention, visualized using BertViz.

tural aspects of language [19]. Multiple head attention allows the model to focus on separate aspects of language simultaneously during each layer and weigh its outputs accordingly.

It has also been shown that a concept of null attention exists [19], which is defined as heads defaulting to placing attention weight on the first token when no tokens in the sequence were found to be relevant by the head. An illustration of an occurrence of null attention is shown using BertViz [18] in Figure 2.

2.4 Subject Model: PolyCoder

The subject of this paper is the medium-sized 400M parameter version of the PolyCoder model [21], further references refer to this PolyCoder-400M variant.

PolyCoder uses a GPT2-based decoder-only architecture, consisting of 24 internal layers with 16 attention heads. Its internal dimension is 1024 positional encodings, which is also the input size it was trained on, so we should provide it with 1024 tokens of context to get ideal predictions during our evaluation.

PolyCoder is trained exclusively on source code, the authors selected up to 25k popular GitHub repositories per language as training datasets. Duplicates were removed and files under 1MB with at least 100 tokens were selected from 12 languages, the languages sampled for training can be seen in Table 1. A GPT2 Byte Pair Encoding [15] tokenizer trained on a subset of the data is used for tokenization. As the model is trained from scratch on source code, any natural language understanding is learned from comments and documentation present in the files.

3 Approach

Our approach to answering if there is a difference in performance for element types is to create predictions for 200k inputs per language with known outputs while using a Tuned Lens to get intermediate results from all layers. We then apply a method to label element types and assess the correctness of outputs across layers. This provides a clear answer to our hypothesis that there is a difference and that language elements require fewer resources, but limited insight as to why.

We follow up by inspecting the focus of the attention heads for specific tokens in these sets. We contribute a method to identify whether these heads are idle or not and use this method to inspect the difference in activity between layers. This offers insight as to why the performance difference exists.

Language	Repositories	Files
C	10,749	3,037,112
C#	9,511	2,514,494
C++	13,726	4,289,506
Go	12,371	1,416,789
Java	15,044	5,120,129
JavaScript	25,144	1,774,174
PHP	9,960	1,714,058
Python	25,446	1,550,208
Ruby	5,826	674,343
Rust	4,991	304,842
Scala	1,497	245,100
TypeScript	12,830	1,441,926
Total	147,095	24,082,681

Table 1: PolyCoder Training corpus [21]

3.1 Data Preparation

Some preparation is required for the data used in prediction and attention experiments, we justify our choices and relate them to previous work.

Model and Dataset selection

We select PolyCoder firstly because it is within the size range we can reasonably evaluate on the compute resources available to us, PolyCoder is on the upper end of this range. The model was part of recent state-of-the-art research within this size class for code inference tasks and is compatible with the Tuned Lens and BertViz inspection methods. Its sole training on code and lack of Natural Language training also presents an additional interest.

As dataset we use a de-duplicated subset of The Stack [8], a dataset of permissively-licensed source code. The selected subset consists of 100k files in the Java, Kotlin, Go, C++, Python and Julia languages. This selection includes The Kotlin and Julia languages to compare performance on code the model was not trained on. As the exact set of files used to train PolyCoder is ambiguous, these unseen languages also serve as a reference point against evaluation on training data.

Pre-Processing

We are interested in the effects of language-specific aspects, as such we focus purely on code inference by stripping comments and docstrings from the inputs using regex-based filters. We then fully tokenize each comment-stripped sample using the tokenizer defined for the model and take a subsection of each tokenized sample to create samples that have similar lengths. As PolyCoder was trained on 1024 token length inputs, we require at least 1024 tokens of left context to generate an optimal prediction, shorter samples are removed. We also limit the number of predictions per sample to 2024 tokens, for 1000 predictions per sample. This is a sufficient length to capture code syntax structure, accounting for white space being a significant fraction of the tokenized inputs. If the sample was longer this 2024 token subsection is randomly selected from the sample in a reproducible manner.

The results of these steps are saved with the original data in a new dataset from which we take samples in future steps. This simplifies matching future results against the original inputs and the subsection method provides a sufficiently spread distribution of code samples across each language, with lengths that are ideal for our predictions.

3.2 Generating Predictions

In order to assess performance we require a large quantity of predictions in each language to ensure that a significant amount of both token types of interest are present. Taking into account the presence of other tokens, we estimate that 200 samples from our pre-processed dataset for 200k predictions per language should be sufficient. We extract layer states and apply the Tuned Lens method to translate these to a meaningful token representation and store the results. We acknowledge the DelftBlue supercomputer, provided by the Delft High Performance Computing Centre [5] for enabling this scale of computation.

3.3 How to tell if a Token is User or Language Defined?

In order to compute statistics per token type, we label the input tokens according to these definitions:

- White-space tokens are excluded from all statistics: $spaces, indents, linebreaks$.
- Tokens listed explicitly as Keywords in the documentation for their respective language are marked as such: for, $else$, $return$, etc.
- Tokens listed as operators in the documentation for each language are marked as such: $+$, $=$, &&, etc.
- Tokens listed as syntax elements in the documentation for each language are marked as such: (, }, (), etc.
- Only the remaining tokens after labelling all the above are considered user-defined.

We consider Keywords, Operators and language-specific Syntax elements as language-defined elements. The exact sources used for these definitions are listed in Appendix B. As syntax and operator tokens occur frequently and consistently and skew the results, we also evaluate the keyword group separately for validation.

We questioned whether this resolves elements that become split by the Tokenizer and found that all language-defined elements are common enough to receive their own token in the vocabulary. If there are white space or syntax elements surrounding these tokens, we can conclude they are not a split section of a user-defined word like $elsewhere$, and can safely assign them as language-defined elements.

This leaves one ambiguous group of elements that we label as user defined elements: names like $print$ from the standard and common libraries of each language. These could also be considered as within the language-defined elements scope. However, we believe these are better defined under user-defined names considering the challenge of defining when a name is common enough.

3.4 How many Layers to achieve a Correct Prediction?

The Tuned Lens gives us a reliable token representation per layer. We match these against the previously stored tokenized inputs in our pre-processed dataset. We label each layer prediction as correct or incorrect and create a new dataset entry where we enter the first correct layer per token defined as:

- Token is not also marked as correct in any lower layer.
- Token is not marked as incorrect in any higher layer.

This metric accounts for uncertain instances where the model still changes its decision later on, before settling on a prediction. Using this metric we can plot the distribution of layer performance per language and element type, draw clear conclusions for our main question, and identify average and outlier instances of interest for a further study of the model's behaviour as these predictions progress through internal layers.

3.5 Specific Token Performance

To further confirm the results of the evaluation of accuracy per layer and explore possible explanations, we begin by looking at the relative performance of specific tokens. For language-defined elements, we select common keywords that are defined for all languages in our dataset, for user-defined elements we select common variable names. To define common we sort the Tokenizer vocabulary and select tokens with a low id, signifying that they are common and distinct enough to receive their own unique token.

- Language Defined Tokens of Interest: $return, else, if, =$
- User Defined Tokens of Interest: $key, value, start, message$

We define variations of the token with a space character (Ġ) prefix to be the same token and check that no further common variations of them exist in the vocabulary. Additionally for the selected user-defined tokens we define the variation starting with a capitalized character to be the same token, this accounts for use in compound names like $NewValue$.

We find all occurrences of these tokens and plot the First Correct Layer performance separately for each of these tokens per language. This will help us select subjects for evaluating the internal attention mechanism. It also serves as a validation of the previous overall Language and User defined element evaluation. If the classification method was good enough we should see a continuation of observed trends in the individual token performances.

3.6 Generating Attention Weights

To explore causes for observed Token Performance we inspect attention heads and their weights to look for patterns that can explain observed performance differences. Attention Matrices are very expensive computationally and storage-wise, for an input size of 1024 each head needs a 1024×1024 matrix of floats. With PolyCoder we have 24 layers and 16 heads per layer for 384 total heads and $\tilde{3}84$ million floats per prediction.

Due to this, we select one token from each of our earlier identified Tokens of Interest groups as subjects to generate attention weights for. We select the language-defined 'else' and user-defined 'start' tokens due to their low variance in layer performance so that we know their observed performance difference is precise. They also show similar numbers of occurrences in our dataset so that we can be confident their observed performance difference is not due to sample size. We generate attention weights for all occurrences of these tokens.

3.7 Where does Attention go?

If we refer back to the concept of identifying null attention as seen in Figure 2, we can use this concept to identify heads that have not found a pattern of tokens in the input sequence that is relevant to them. This implies that the head is effectively idle. We choose to focus on this null attention concept as it provides a measure of the contribution of individual heads.

The challenge is to define when a head can be considered to be idle as there is always some attention weight to every previous token. We need to define a minimum attention weight on the first token to declare the head as *mostly* idle.

Our metric is the sum of attention weight placed on the first token, which is a (0,1] range that corresponds to the clarity of a line if we refer back to Figure 2. The sum of attention weight on the first token is the sum of weight values of every line pointing to it. To get an idea of the distribution of weight on the first token we observed a set of 46k 1024×1024 heads.

We find a significant shift in this distribution from the original BertViz [18] examples with our larger input size. Where a common null attention head in a 20×20 input would have up to 95% of the attention weight on the first token, the maximum we observe across our 46k PolyCoder attention head samples is 15%. The weight is much more evenly spread out in larger sample sizes and tokens with high attention must be identified from a relatively small difference in weight, which increases the difficulty of identifying trends.

We therefore choose the extremes in the distribution as the most reliable statistic. We consider a head to contain significant null attention only if it is within the top 5% of our weight on first token distribution. We classify all heads involved in our Tokens of Interest predictions according to this metric to build a bar graph that shows the fraction of null heads per layer. If a significant fraction of heads in a layer are null heads, we can conclude that the input was not of relevance to those heads and that the layer has contributed relatively little to further the accuracy of the prediction.

4 Experimental Setup and Results

We perform our experiment according to the approach described in section 3 using the HuggingFace transformer libraries [7] to generate the results shown here. Throughout this section we maintain the same coloring for language-defined results (green) and user-defined results (yellow).

Handling Computational Limits In order to facilitate computation on the required scale, we make some critical changes to our approach:

- As we are only using inference functions we enable $.eval()$ mode for models, primarily to reduce the load but also to disable the dropout mechanisms used in training. This increases the quality of predictions.
- We run all model and lens functions with $no_grad()$ to disable gradient computations, this has no effect on prediction quality but reduces GPU memory load.
- We split all inputs and outputs into batches and save these to separate files in between each step, this greatly reduces memory requirements. This also allows us to share intermediate results for review and reuse without the need for significant computational resources.

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

Table 2: Element Quantities Sampled per Language

4.1 Token Type Distribution

As outlined in subsection 3.3 we label input tokens according to our definition of token types, we receive sample sizes of each respective token type per language as shown in Table 2

Note the significantly smaller sample size for tokens that are strictly defined as keywords for each respective language. Inputs contained equal quantities of total tokens per language, variations are due to language characteristics. Java shows a relatively high use of keyword tokens due to being statically typed with common use of access modifiers. Go shows relatively high total counts due to fewer white-space tokens in the samples which are excluded in statistics.

4.2 First Correct Prediction Layer per Type

To show relative performance per element type with respect to layer depth of the model, We assess the correctness of predictions per layer and record the first correct layer as described in subsection 3.4. We plot this distribution as a box plot per Language and Element type in Figure 3.

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), surprising performance on untrained languages, and stronger fluctuation in user-defined element performance.

4.3 First Correct Prediction Layer for Tokens of Interest

We use box plots to show the distribution of the first correct layer across all predictions that include our selected Tokens of Interest. The results for the Java (Figure 4) and Go (Figure 5) languages offer the best results and are of interest as the most and least trained on respectively for PolyCoder. For completeness, the others are included in Figure C. The medians are consistent across languages but the variability is inconsistent, likely due to sample sizes, language characteristics, and token type classification accuracy.

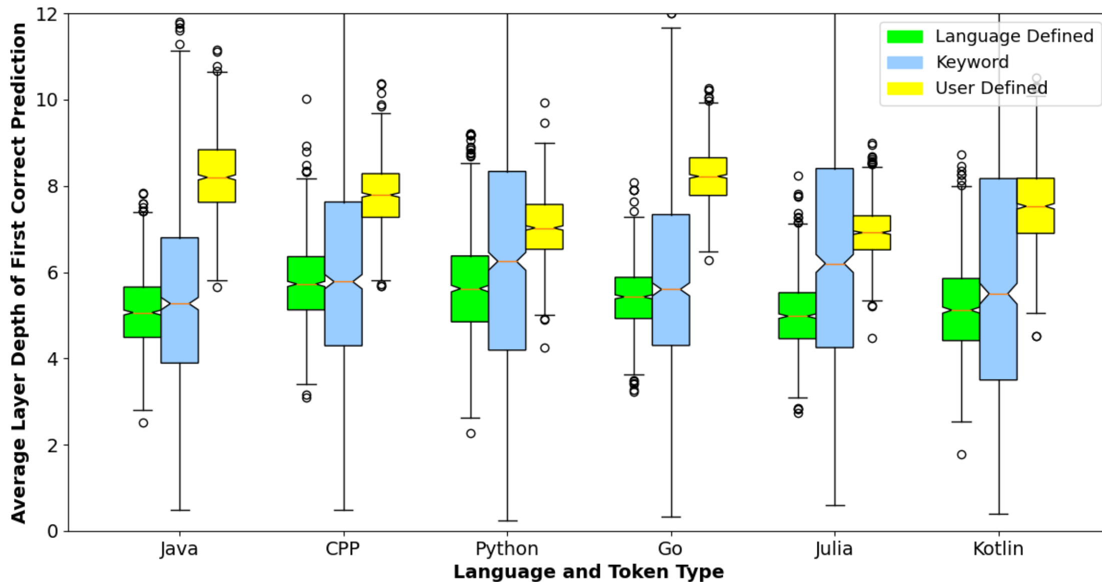

Figure 3: Distribution of First Correct Prediction Layer per Language and Element Type. Lower is better.

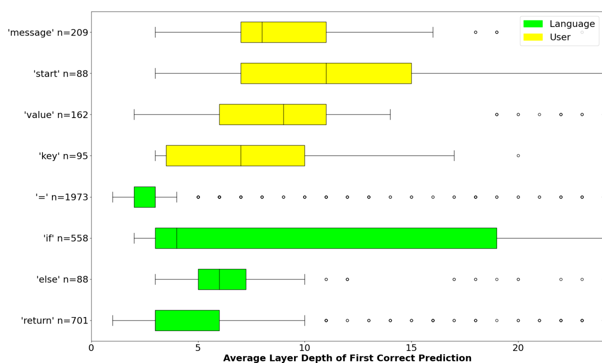

Figure 4: Distribution of First Correct Prediction Layer for Tokens of Interest in Java. Distributions towards the left are better.

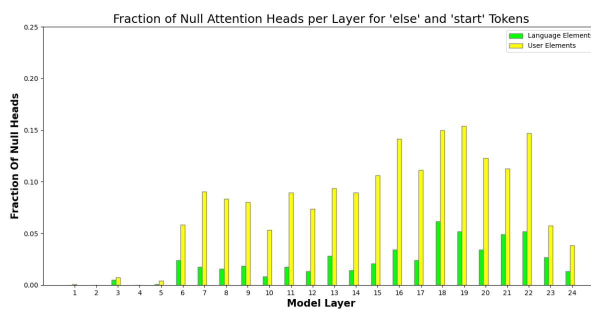

Figure 6: Fraction of Null Attention Heads per Layer in Java for 'else' (language-defined) and 'start' (user-defined) tokens, 960 heads per layer per token. A lower fraction is better.

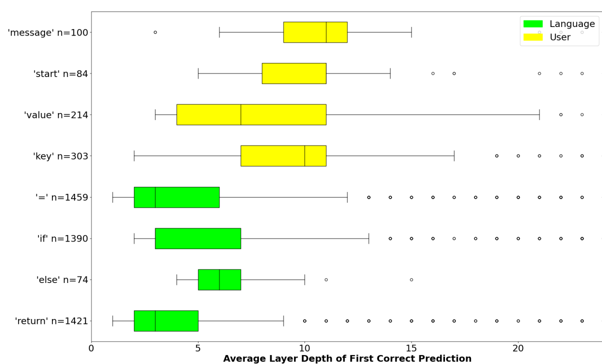

Figure 5: Distribution of First Correct Prediction Layer for Tokens of Interest in Go. Distributions towards the left are better.

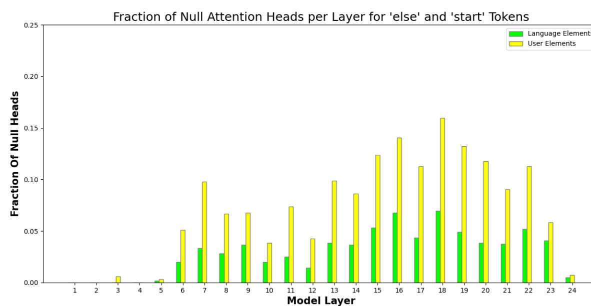

Figure 7: Fraction of Null Attention Heads per Layer in Go for 'else' (language-defined) and 'start' (user-defined) tokens, 960 heads per layer per token. A lower fraction is better.

4.4 Fraction of Null Attention Heads per Layer

We use bar graphs to show the fraction of Null Attention Heads per layer for 'else' (language-defined) and 'start' (user-defined) tokens for the Java (Figure 6) and Go (Figure 7) languages. We used an equal number of samples (n=60) of each token for validity, giving us 960 head samples per layer.

5 Discussion and Limitations

From the results shown for the distribution of First Correct Predictions, we see that language-defined elements are predicted earlier by a consistent and significant 2 layers. This correlates with earlier work indicating greater difficulty for user-defined elements [1].

There is a second observation that both element types are on average predicted correctly in the model's very early layers (6 and 8 respectively). This happens to such a degree that our plot of average distributions excludes the remaining 12 layers of our 24-layer PolyCoder model.

We can infer that early subsections of layers of the model show a strong capability to predict certain token types correctly. Observing the specific tokens of interest shows this is especially true for the common assignment operator '='. There is potential for techniques like the Tuned Lens to skip layers for certain token types and still achieve high accuracy.

From previous work, we saw hints of the transfer of performance between languages [21]. Our results show unexpectedly high layer-wise performance for languages the model is not trained on, despite a higher failed prediction rate [Appendix A]. It is possible that certain language characteristics transfer very well across training sets even if the final overall prediction accuracy is lower, but we are not certain enough to conclude this due to inaccuracies in our methods.

These results also 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 should relate to more occurrences of a first correct prediction in higher layers. 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 this absence evidence that the heads in earlier sections must be of a more general purpose: they see relevance in either type. From the earlier correct layer predictions, we infer that these more general-purpose heads must be significantly better at resolving language-defined elements.

5.1 Limitations

To support our conclusion, we show the limitations of our work that are considered and accounted for.

Data Processing Limitations

In our preparation stage an assumption was made that stripping comments would allow us to more effectively spend our compute resources on predictions of code. We are wary of this decision as it is a deviation from the model's training and will degrade prediction performance. A concern exists that prediction quality for user-defined elements might degrade to a greater degree due to missing context from comments. However, we believe that a greater degree of variability would be observed in prediction depth for user tokens if this was a great influence. Ideally, we would verify this with a rerun of the experiment to strengthen our conclusion.

Lens Limitations

It is important to note that while a Tuned Lens provides an objectively better assignment of meaning to an internal layer state, it is still a model trained to minimize an error and produce a most likely match, and as such is not an absolute truth. However, the authors show significant results in reducing variance, bias, and uninterpretable states, and present significant evidence on the degree to which its outputs are representative of the final layer distribution and corresponding influence of features. As such we feel confident to place a high degree of trust in the validity of its outputs.

Token Type Classification

We have taken great care to be consistent in the application of our white space, keyword, operator, and user-defined labels for each language, but the method is rudimentary and likely has a bias towards some languages. We are confident however that our multiple experiments cross-validate the results.

Classifying Null Heads

As mentioned in subsection 3.7 it is challenging to classify a null head due to the distribution of attention weights for higher input size sequences. This is worsened by our inability to inspect them visually, a rendering method that scales with input size could greatly assist in validating the classifications. Ideally, we would also verify that no other token besides the first one is gaining significant attention. We do believe our choice to classify on extreme first token weight is valid, as our observed ratio of classified null heads corresponds with the ratio of pure null heads we would expect from visual identification for smaller input sequences.

6 Conclusions and Future Work

To contribute to the topic of evaluating model performance we set out to investigate how user-defined elements compare to language-defined elements and posited the hypothesis that a Large Language Model should be able to predict language-defined elements earlier due to their well-defined structure and meaning. From our results, we conclude that such a difference exists, that it is consistent across languages, and is quantifiable as a 2-layer advantage in average layer depth required for correct predictions of language-defined elements over user-defined elements. We therefore confirm our original hypothesis and demonstrate a new capability to evaluate the internal layer performance of specific aspects of code languages using a Tuned Lens. This technique can be further refined by reconsidering the effects of code comments and improving the token classification methods.

For specific high-occurrence language-defined Tokens such as Operators, we observe an even greater difference in performance. We question with what certainty a prediction can be identified as such a token and how early, this could allow for an evaluation of early exiting techniques to relate exit points to effects on accuracy and computational time. We suggest this as a topic of interest for future research.

7 Responsible Research

For our research, we have accounted for several common topics of the code for responsible research: Reproducibility, Integrity, and the ethical consideration of applied Machine Learning.

Reproducibility Reproducibility is a recently increased concern regarding topics in Computer Science [16], especially those involving large Datasets like ours. Not all scales of computation are equally accessible, and not all source code involved in research is shared properly. We have taken great effort to ensure that our results are accessible and reproducible:

- Our Paper is publicly accessible [1], as is our Code [2] and the used Datasets [3] to the best of our ability.
- We also publish our intermediate results of computations along with the above Dataset, to aid reproducibility for those without access to significant compute resources.
- Our Approach section and linked sources are sufficiently detailed to be able to construct a similar method.

Use of Resources Large-scale computation for machine learning is expensive, both financially and ecologically, our research topic itself relates to reducing this effect. Although the topic was new to us and our time limited we have striven to the best of our ability to make our code efficient and to avoid repeated computations. This is also in consideration of our peers who we share our compute resources with.

Integrity of Results Research is competitive and time-constrained, this places pressure on researchers to show positive results and we have taken care to consider our own biases regarding this. We have used reasonable methods to ensure our results are based on a fair and representative distribution of data. Results that conflict with our expectations are shown or used as avenues to identify mistakes, with the mindset that even negative results are still areas of interest.

Ethical Considerations Our research involves no direct application, as such it is easy to hide behind our scientifically driven motive of improving performance. Yet as we contribute to an increased understanding of the performance of LLMs, we should also caution against their dangers, specifically their tendency to maintain wrong or harmful biases present in training data.

[1] https://repository.tudelft.nl/

[2] https://github.com/ErikMekkes/CodeShop

[3] https://huggingface.co/datasets/ErikMekkes/CodeShop

References

[1] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, 2013.

[2] Gareth Ari Aye and Gail E. Kaiser. Sequence model design for code completion in the modern ide, 2020.

[3] Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor Ostrovsky, Lev McKinney, Stella Biderman, and Jacob Steinhardt. Eliciting latent predictions from transformers with the tuned lens, 2023.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, and Jared Kaplan et al. Evaluating large language models trained on code, 2021.

[5] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 1)*. 2022.

[6] Vincent J. Hellendoorn and Anand Ashok Sawant. The growing cost of deep learning for source code. *Commun. ACM*, 65(1):31–33, dec 2021.

[7] Hugging Face. *Hugging Face Hub Docs : Transformers*. Hugging Face, 2023.

[8] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, and Carlos et al. Muñoz Ferrandis. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

[9] Suhas Kumar. Fundamental limits to moore's law. 11 2015.

[10] Thomas Wolf Lewis Tunstall, Leandro von Werra. *Natural Language Processing with Transformers - Building Language Applications with Hugging Face*. O'Reilly Media, 2022.

[11] Thomas Wolf Lewis Tunstall, Leandro von Werra. *Natural Language Processing with Transformers - Building Language Applications with Hugging Face*. O'Reilly Media, 2022.

[12] Alan Blackwell Mariana Mărășoiu, Luke Church. An empirical investigation of code completion usage by professional software developers, 2015.

[13] nostalgebraist. interpreting gpt: the logit lens. https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens, 2020.

[14] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[15] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. ACL, 2016.

[16] V.C. Stodden. Reproducible research: Addressing the need for data and code sharing in computational science. *Computing in Science and Engineering*, 12:8–13, 01 2010.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[18] Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics.

[19] Jesse Vig and Yonatan Belinkov. Analyzing the structure of attention in a transformer language model, 2019.

[20] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. What do they capture? – a structural analysis of pre-trained language models for source code, 2022.

[21] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code. Published as a workshop paper at DL4C @ ICLR 2022, 2022.

A Failed Predictions

A controversial take compared to other performance evaluations: We only compute the fraction of correct final predictions per language and element type as an exploratory statistic and do not consider it further. Our focus lies on evaluating the progression of performance per layer which is difficult to define when progress is still uncertain at the final observed stage. It is unclear if a failed prediction will eventually be correct and after how many additional layers. Whether this can be estimated from the behaviour we see in layer progression is a potential topic for future studies of the attention mechanism.

we show the fraction of correct final predictions made by PolyCoder per language and token type in Figure 8. We match each prediction against the real next token from the sample input, predictions are correct if the token ids match.

B Token Type Definitions

For completeness, we list the exact reference definitions used for token type categories here.

- C++: We use the official cpp reference definitions, currently listed under https://en.cppreference.com/w/cpp/language, specifically the https://en.cppreference.com/w/cpp/keyword and https://en.cppreference.com/w/cpp/language/expressions#Literals sections.
- Go: We use the Go language specification as listed under https://go.dev/ref/spec, specifically the https://go.dev/ref/spec#Keywords, https://go.dev/ref/spec#Operators_and_punctuation and https://go.dev/ref/spec#Types sections.
- Java: we use the Java SE17 specification as listed under https://docs.oracle.com/javase/specs/jls/se17/html/jls-3.html#jls-3.9, specifically the 3.9 Keywords, 3.11 Separators and 3.12 Operators sections
- Julia: We use the Julia language organisation reference manual, specifically the https://docs.julialang.org/en/v1/base/base/#Keywords and https://docs.julialang.org/en/v1/manual/mathematical-operations/ sections
- Kotlin: We use the Kotlin language organisation reference documentation at https://kotlinlang.org/docs/keyword-reference.html, specifically the Hard Keywords and Operators and Special Symbols Sections.
- Python: we make use of The 3.11 language specification as defined under https://docs.python.org/3.11/reference/index.html, specifically the 2.3 Identifiers and keywords and 2.5 Operators sections.

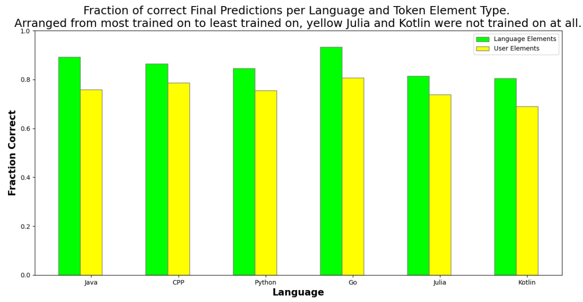

Figure 8: Overall Successful Predictions per Language and Element Type. Higher is better.

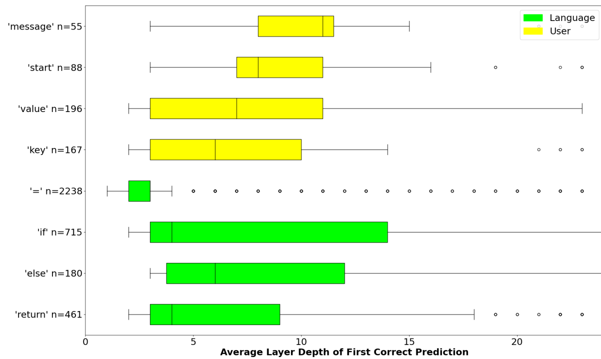

Figure 9: Performance on specific C++ element types.

C Tokens of Interest Performance for remaining Languages

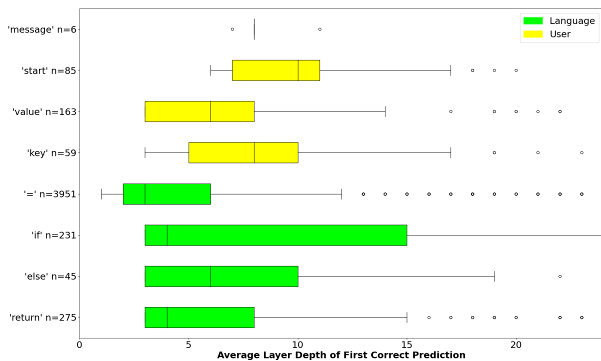

Figure 10: Performance on specific Julia element types.

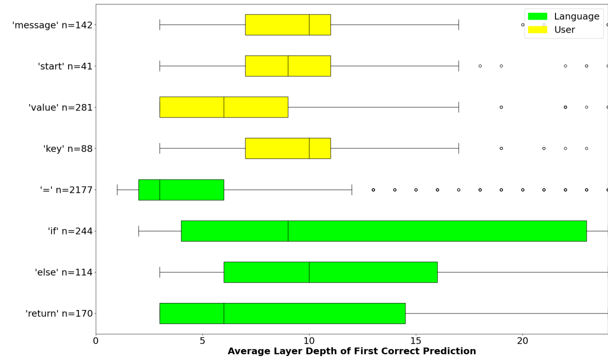

Figure 11: Performance on specific Kotlin element types.

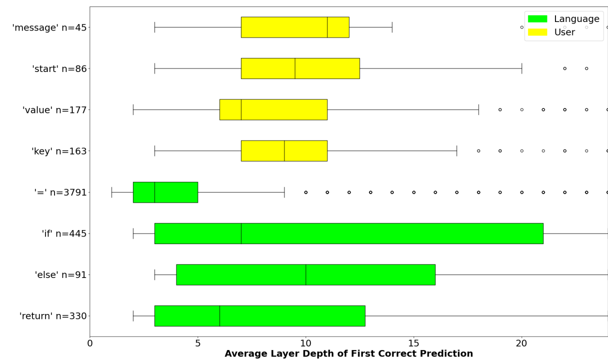

Figure 12: Performance on specific Python element types.