A Cross-Lingual Evaluation of CodeGen's Performance in Code Completion

Miranda Keeler[1]

Supervisors: Prof. Dr. Arie van Deursen[1]**, Dr. Maliheh Izadi**[1] **and Ir. Jonathan Katzy**[1]

[1]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 26, 2023

Name of the student: Miranda Keeler
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi, Ir. Jonathan Katzy, and Azqa Nadeem

Abstract

We present an investigation into the relationship between the average depth of the first correct prediction and the performance of CodeGen. This was done on a dataset comprised of code files comprised of C++, Go, Java, Julia, Kotlin, and Python. The analysis involved investigating the model's predictions at different layers using a Tuned Lens, which enables examining the intermediate representations. Additionally, attention heads were examined to gain insights into the model's behavior. We found that there is a subset of four layers in which tokens are predicted correctly for the first time. These peaks are evident in CodeGen's performance and come after a small dip, a dip that is present in the last layer. The results shed light on the varying performance of different layers and provide valuable insights into the strengths and weaknesses of CodeGen. These findings contribute to our greater understanding of language model performance in code completion tasks and provide implications for future improvements in this domain.

1 Introduction

PLMs have made significant strides in code completion and program synthesis, enabling developers to write code more efficiently, access common patterns quickly, and problem-solve more effectively, making software engineering more accessible to novices. Notable examples of this are GitHub's Copilot[1], powered by OpenAI's CodeX engine and Salesforce's CodeGen models.[2]

While these models make software engineering more accessible, our knowledge and understanding of their inner workings, performance characteristics, and limitations remains limited. Their ever-expanding architectures and training processes make it challenging to comprehend their attention mechanisms fully. These models may be complex and diverse, but they share the same fundamental foundation—the Transformer architecture.

Learning about a model solely on current performance evaluations is limiting, as they are often limited to a small selection of programming languages such as Java and Python, which can be limiting in evaluation against a broader set of languages. Therefore, we need additional ways to evaluate the performance of PLMs which focus on the internal processes and transformer architecture alongside the final output.

There is research into the behavior of attention heads throughout the layers of a PLM, this has been focused on the patterns found regarding parts of speech or code syntax [1] [2]. Furthermore, general performance metrics focus solely on the final predictions, rather than observing the intermediate steps that were taken to get there[3].

Given the patterns that have been found, we can try to connect these two to evaluate a model's performance based on the layers' predictive patterns which may tell us something about the efficiency of the model [1].

This research investigates the cross-lingual performance of CodeGen. It aims to answer the following research question: How does the average depth of the first correct prediction relate to the performance of CodeGen? We generate a test dataset of CodeGen's hidden states on a multi-lingual corpus of code files from the Stack and analyze the predictions at each, translated by a Tuned Lens [4] [5]. We then observe the attention heads at several points of interest to determine the link between attention and performance.

Our findings suggest a weak correlation between the depth of the first correct completion and CodeGen's performance. Further research is needed to explore how to adjust this new metric to extend its potential and identify and understand the model's strengths and weaknesses more thoroughly through attention.

The main contributions of this study are:

- A multi-lingual analysis of predictions made by CodeGen.
- Patterns found in the performance of CodeGen across its layers.

2 Background and Related Work

2.1 Code Completion

Code comprises various components, including tokens, lines, statements, functions, classes, and more. Code completion can be applied to any of these components, where the goal is to predict the next component of the same type. For this research, we are evaluating code completion on the token level.

2.2 CodeGen

Transformers

Transformers were proposed by Vaswani *et. al.* as the successor to Neural Networks (NN) in the field of language modeling [6]. It is a model architecture that utilizes attention mechanisms to establish global dependencies between input and output, enabling parallel processing of long input sequences and improving accuracy and efficiency [6].

Self-attention is an attention mechanism that computes a sequence representation by relating different positions within a given sequence with an attention function[6]. This function maps a query and a set of key-value pairs to an output, a weighted sum of values based on the compatibility of the query with the corresponding key [6]. Self-attention will be advantageous as we can map out where it is going for clearer insights [7].

We focus on autoregressive decoder transformers, transformers with self-attention layers that enable each position to attend to the current and preceding positions only [6].

Model Details

CodeGen is a set of transformer models with left-to-right causal masking with next-token prediction language modeling as the learning objective[8]. They have a standard transformer decoder architecture with left-to-right causal masking,

[1] https://openai.com/blog/openai-codex

[2] https://github.com/salesforce/CodeGen

Model Summary		
Parameters	Training parameters	350M
	Number of layers	20
	Number of heads	16
	Context length	2048
Training data		
The Pile	Natural language	1159.04 GiB
	Code	95.16 GiB
BigQuery	Java	120.3 GiB
	C++	69.9 GiB
	Python	55.9 GiB
	C	48.9 GiB
	JavaScript	24.7 GiB
	Go	21.4 GiB

Table 1: This is an overview of the CodeGen-350M-multi's parameters and training data.

have rotary position embedding for the positional encoding, and execute self-attention and feed-forward circuits in parallel [8].

For this research, we work with the CodeGen 350M-multi model. The details of this model can be found in Table 1. It was initialized from another CodeGen model that was trained in Natural Language Processing (NLP), on data from the Pile dataset[9], and fine-tuned with programming data from BigQuery [8].

The model's tokenizer was trained with whitespaces and treats them like parts of the tokens, meaning a word will be encoded differently depending on if it starts with a whitespace[8].

In the feed-forward passes, self-attention and the feed-forward circuits are executed in parallel, with each computation happening simultaneously. This is $x_t + 1 = x_t + attn(ln(x_t)) + mlp(ln(x_t))$, where $attn()$ is the self-attention computation, $mlp()$ is the feed-forward computation, and $ln()$ is the layer-norm[8].

2.3 Tuned Lens

The Tuned Lens allows us to peek into the computational process of a model. It comprises one translator for each decoder layer, transforming the hidden states to ensure that their unembedded representations closely match the final layer logits [5]. Combining these with the pre-trained unembedding maps the model's hidden states, which contain the current predictions and attention scores, to a probability distribution over the model's vocabulary [5]. With the Tuned Lens, we can observe the predictions for each of the model's layers. In Figure 1, you can get a better idea of how the Tuned Lens works with CodeGen.

Probing-Based Observation

Through a novel probing technique, Chen

3 Methodology

CodeGen was used as-is to generate next-token predictions on a multi-lingual code dataset.

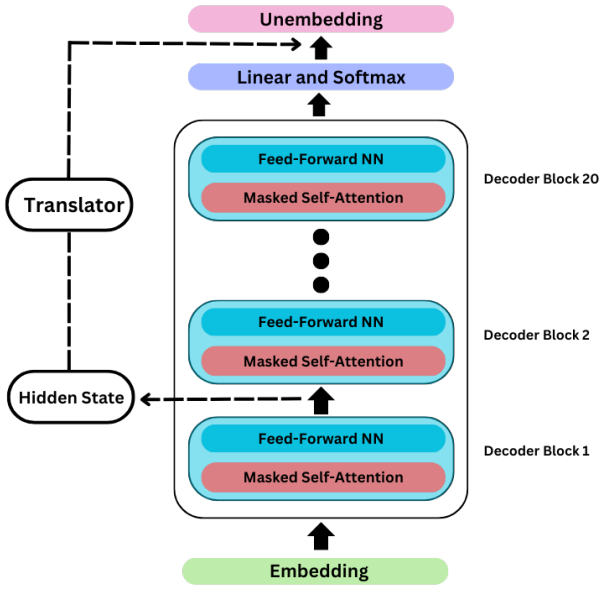

Figure 1: This is how the Tuned Lens interacts with CodeGen. The Tuned Lens takes the hidden state between layers and applies a learn affine translation. This is then converted into logits with the unembedding layer.

3.1 Data Collection

We collected a subset of code files in six languages, C++, Go, Java, Julia, Kotlin, and Python, from the Stack dataset [4]. From this subset, there are two languages that CodeGen was not trained on, Julia and Kotlin [8]. Examining familiar and unfamiliar languages asses CodeGen's generalizability and identifies potential biases or limitations.

3.2 Data Processing

We stripped the files of their comments, removed those that could not provide at least 512 tokens of context, and took a random section from files that were more than 1500 tokens long.[3]

3.3 A Two-Part Exploration

To answer the research questions, we observed what was happening and then examined why. We first looked at CodeGen's predictions across all of its layers with a Tuned Lens, looked at potential interesting samples or findings, and then analyzed the behavior of the attention heads across all parts and at those points of interest.

Phase 1: Tuned Lens Investigation

To determine if there is a relationship between the average depth of the first correct completion and the performance, we generated a multi-lingual dataset of predictions made by CodeGen. The Tuned Lens translated the predictions for each

[3]We determined this value by finding the average number of chars per token across multiple models, 6.5, and taking a random selection with char length $6.5 * 512 + 6.5 *$ number of tokens to predict ≈ 1750 tokens.

layer. With this data we could then find the average depths of the first correct completion and the performance.

1. We compared the true values with the outputs from each layer to determine when a prediction is correct.
2. We then found the token performance per layer and occurrence in the entire dataset.
3. We visualized our findings to assess the potential relationship between average depth and performance and locate potential points of interest to look at in part two
4. We found the correlation between average depth and performance to decide if a relationship between depth and performance could exist.

We excluded a set of standard tokens that appeared over 10,000 times in each dataset to keep the focus on potentially more interesting tokens.

Phase 2: Attention Investigation
In addition to visualizing attention for individual inputs to the model, we also analyzed attention in aggregate over a large corpus.

1. We generated a set of attention matrices from random samples and layers, to ensure an even distribution between sample language, layers, and attention heads.
2. We then found the number of null heads present in each layer, as an aggregate of the random attention heads.
3. We then analyzed at the ratio of null heads and theorized how it could explain the patterns found in phase 1.

4 Experimental Setup

This work aims to provide insight into the relationship between the model's architecture and model performance by answering the question, How does the average depth of the first correct completion relate to the performance of CodeGen? To accomplish this, we split the experiment into two phases, with each phase answering its respective question:

R1: What is the relationship between the average depth of the first correct completion and performance?

R2: Are there distinct, observable patterns in attention that explain the findings from Phase 1?

4.1 Evaluation Metrics

The predictions are compared with the true values and evaluated with several metrics across various scales.

Average Depth
For each token, we found the summation of the layers that it was correctly predicted for the first time / its total number of occurrences. To find the average depth of the first correct completion

Performance
We use accuracy to measure the performance of tokens and layers. For token accuracy, this is c_t/n_t, where c_t is the number of correct predictions of a token t and n_t is the total number of occurrences of the token t. This is c_l/n for layer accuracy, where c_l is the number of correct predictions in the layer l and n is the total number of tokens predicted.

Statistics on CodeGen's Next-Token Predictions		
Language	Accuracy	Freq.
C++	90.23%	116
Go	91.96%	116
Java	92.08%	80
Julia	85.96%	106
Kotlin	85.92%	54
Python	89.45%	86
Combined	89.33%	-

Table 2: An overview of the statistics gathered on CodeGen's next-token completions for each language and a combined dataset.

Null Head Count
Vig and Belinkov have defined null attention as attention directed towards the first token [2]. We have applied their definition to determine if an attention head is a null head. We define null heads as attention heads receiving more than 70% or 80% of the total attention.

4.2 Dataset

There was a wide spread in the frequency of tokens. Several tokens were present over 10,000 times per language in the dataset while others were present once. To reduce any potential noise in the performance metrics and keep the focus on more interesting tokens, we removed tokens that were present in the dataset less than 10 times and more than 10,000 times.

4.3 Environment and Tools

Configuration
To optimize the speed at which the predictions were generated, we utilized a CUDA setup on a GeForce RTX-3080 GPU. The experiments can also be done on a CPU.

Implementation
We batched our inputs for efficiency, combining eight different context windows for eight next-token predictions. Each context window was at least 512 tokens long. If the context windows were not the same shape, we padded the shorter inputs with zeros. This way, we ensured that there was not any right context provided for prediction.

5 Results

An overview of the overall statistics per language can be found in Table 2. The performance of

5.1 Findings from Phase 1

Distribution of First Correct Completions
We found notable spikes across the distribution of the first correct completion of tokens over CodeGen's layers, at layers 1, 5, 8, 13, and 14. These spikes differed across the low-resource languages that CodeGen was not trained on, Julia and Kotlin. This difference can be seen in Figure 2.

It is also visible that the tokens that were correctly predicted for the first time in layers 0 and 4 were more likely to be correct in the final prediction than tokens predicted correctly for the first time in layers 12 - 19. This finding was consistent across all languages and indicates that we should look closely at the attention heads in these layers.

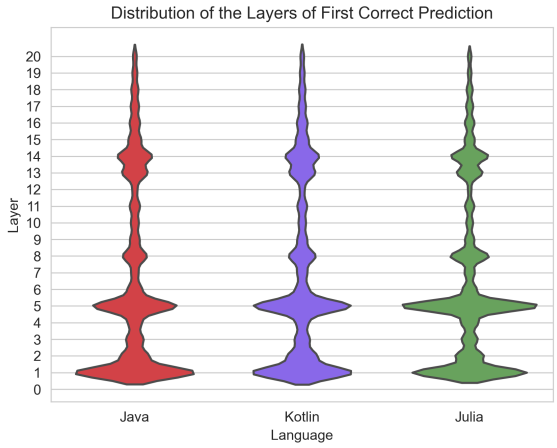

Figure 2: The distribution of the first correct completions for each layer. There are notable differences between Java, Kotlin, and Julia in layers five and eight.

Performance

The performance, measured as accuracy, of the predictions, increased as the layers got deeper. This trend is visible in Figure 3. There were drops in performance in layers seven, twelve, and twenty. These drops in layers seven and twelve happen right before an increase.

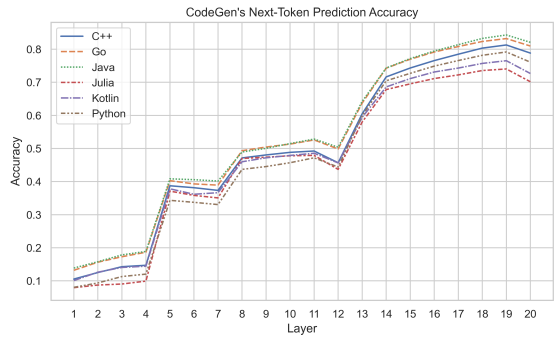

Figure 3: The distribution of the first correct completions for each layer. There are notable differences between Java, Kotlin, and Julia in layers five and eight.

Relationship Between First Correct Completion and Performance

There was a weak correlation between the average depth of the first correct completion and the performance of CodeGen. This is clear in Figures 4 and 5.

General Performance on New Languages

We have found that CodeGen's predictions tend to be more accurate for the languages it was trained on, compared to the languages it was not (Julia and Kotlin).

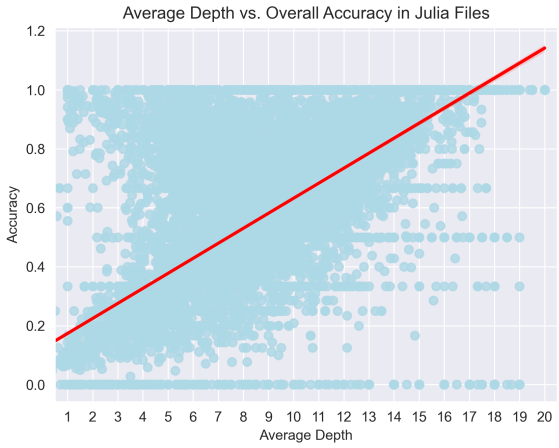

Figure 4: The average depth of the first correct completion and performance on Julia files.

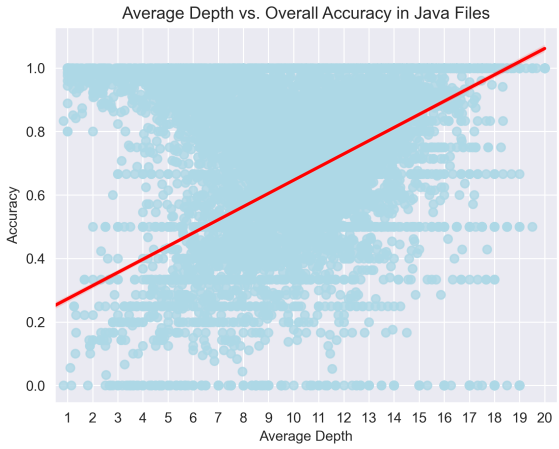

Figure 5: The average depth of the first correct completion and performance on Java files.

5.2 Java

Token patterns in the Layers

There were clear patterns in the predictions of a select set of tokens. For example, the token "s" was always predicted correctly for the first time in layer 0, and the token for "Column" was always correct for the first time in layers 12-14.

Relationship Between Average Depth of the First Correct Prediction and Performance

The languages that CodeGen was not trained on had lower scores for the token predictions than the languages it was. The tokens that it predicted most accurately for these languages were tokens that are more commonly found across the set of languages.

5.3 Findings From Phase 2

Attention Patterns

We took a random subset of attention matrices across randomly selected heads in aggregate. For each language, the patterns were initially similar, however, there were noticeably more null heads present in layers 12 and onwards, with Java having the most.

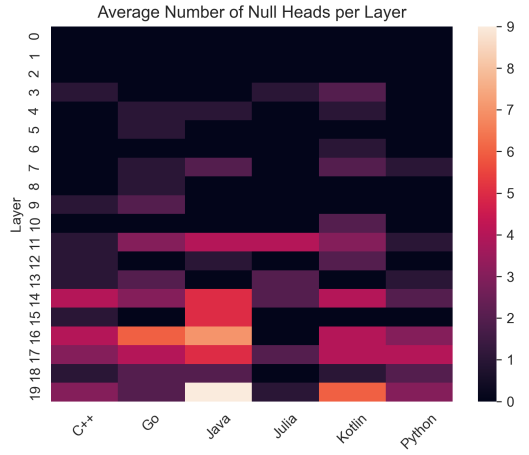

Figure 6: The average number of null heads in each layer for each language when the threshold was set at 70%. The data was sampled from a random set of predictions across 100 files.

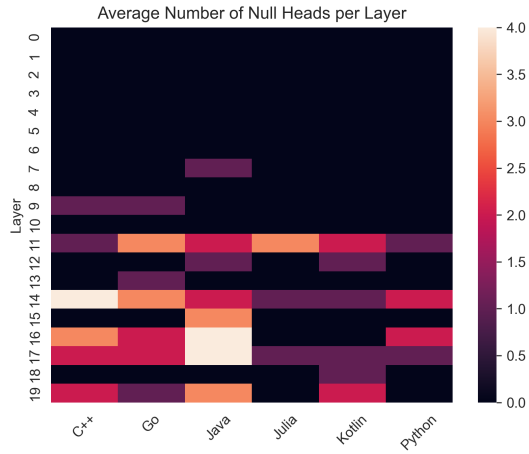

Figure 7: The average number of null heads in each layer for each language when the threshold was set at 80%. The data was sampled from a random set of predictions across 100 files.

6 Discussion

6.1 Limitations

Due to the limited time of the project, we needed to make choices that were less involved than could be deemed necessary. It took longer than initially thought to gather the correct predictions and interpret them. It also took longer than initially thought to gather and process attention, a task that is computationally very expensive. As a result, the conclusions that can be drawn are not very definitive.

In order to really understand the relationship between attention and predictions, we need to have a more robust measurement of attention. Using the average number of null heads per layer did not provide any new information beyond what has already been discovered in previous work. Furthermore, the metric that was chosen, the definition of a null head, missed out on the details regarding the attention to other tokens that ultimately may have had a larger impact than we initially thought.

7 Conclusions and Future Work

The overall performance of CodeGen is quite high, with the accuracy for each language being greater than 85%. The relationship between the average depth of the first correct completion was not very strong, with correlation values around 0.3 and 0.4.

The performance tended to increase so much across the layers, that having a lower or higher average depth did not contribute as strongly. It is interesting to see the spikes in performance correlating with the layers that have the largest amount of tokens predicted correctly for the first time. Also interesting that these happen right after a dip in performance, which could indicate a relationship between the learning process that goes on between the layers. It also seems that CodeGen's performance would be improved if it had one less layer.

While there were notable spikes in the layers that tokens were correctly predicted for the first time, looking at this as an average was not as insightful as hoped. There are still a lot of things that can be explored with this topic.

8 Acknowledgements

I would like to thank Jonathon Katzy for all of his guidance over the course of the project and for providing support whenever asked. I would also like to thank Maliheh Izadi for the guiding feedback on the planning and report-writing process. I would also like to thank Arie van Deursen for his advice regarding the research process. Lastly, I would like to thank my fellow peers. We collaborated a lot over the quarter and they contributed a lot to my understanding of the experimental process.

9 Responsible Research

9.1 Ethical Aspects

Sensitive information

The Stack has indicated that there may be personal or sensitive information within the datasets, as it was taken from public repositories [4]. As a result, this data may persist in our local datasets, even if it is removed from the stack at a later point.

If this information was located in the comments, then it was not given to the model for predictions, as these were removed. There may, however, be personal or sensitive information within the dataset, as it was taken from public GitHub

repositories. If such information is found, this sample is removed from the Stack, but we do not know when this happens. If we used this file within our own dataset, this information would then persist.

9.2 Reproducibility

Our experiments have been designed to be completely reproducible, as we documented all the decisions that we made and we generated and saved all of our processed and generated data to our own datasets.

References

[1] N. Chen, Q. Sun, R. Zhu, X. Li, X. Lu, and M. Gao, "Cat-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure," 2022.

[2] J. Vig and Y. Belinkov, "Analyzing the structure of attention in a transformer language model," 2019.

[3] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[4] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," 2022.

[5] N. Belrose, Z. Furman, L. Smith, D. Halawi, I. Ostrovsky, L. McKinney, S. Biderman, and J. Steinhardt, "Eliciting latent predictions from transformers with the tuned lens," 2023.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[7] Y. Belinkov and J. Glass, "Analysis methods in neural language processing: A survey," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 49–72, 2019. [Online]. Available: https://aclanthology.org/Q19-1004

[8] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.

[9] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," *CoRR*, vol. abs/2101.00027, 2021. [Online]. Available: https://arxiv.org/abs/2101.00027