An LLM's output is fundamentally a probability distribution over its token vocabulary at each step. This process can be summarized as –

1. For each next token that the LLM predicts, the model calculates probabilities for every token in its vocabulary (which can be 10s or 100s of thousands of tokens)
2. For example, if predicting the next word after "The cat sat on the \_\_\_", it might output:

```

mat: 0.35

floor: 0.25

chair: 0.20

table: 0.15

[other tokens]: 0.05

```

1. When generating text, the model typically samples from this distribution (rather than always picking the highest probability token) to create more natural and varied outputs. The sampling can be adjusted via temperature and other parameters.
2. This process repeats for each new output token, with the previous tokens (both from the input prompt and previously predicted tokens) providing context that influences the next probability distribution.

Thus, changing any token in the input prompt will change the probability distribution of the output token. So, given a sequence of output tokens, we will have a probability distribution over the entire LLM token vocabulary for each output token.

Perturbation techniques to interpret LLMs use the fact that the output token probability distribution is conditioned on the input tokens and compute the change in the output token probability distribution as the tokens in the input are systematically augmented. A token or a sequence of tokens are removed from the input, and the probability distribution of each token in the output are compared to the corresponding probability distribution of token in the original output, and this change in probability is quantified as an interpretability score using various methods. This results in an interpretability score being assigned to –

* Each token or sequence of tokens in the input which are perturbed
* Each token in the original sequence of output token

Feature ablation is one way of calculating this interpretability score by taking the difference in the log probabilities at each token position of the original output. Similarly, Shapley Values also calculate this interpretability score by considering other factors like efficiency, symmetry and linearity along with the difference in the log probabilities at each token position of the original output.

For example, we group the input tokens based on lines in the input prompt, and only perturb the lines present in the input code that needs to be converted to a target language. Running feature ablation method on the YiCoder 9B model for the Logical operations task description from Rosetta Code to convert code from C to C++, we get the following interpretability scores –

A graph on a white sheet

Description automatically generated

In the figure above, we can see that on the y-axis are the code lines from the original C code to solve the Logical Operations task. On the x-axis, are the tokens from the converted C++ code that was generated by YiCoder 9B previously as a response to our prompt.

Lets take the first line from the C code “void print\_logic(int a, int b) {”, and the first token in output “include”. It has a score of -2.3. This means that the probability of the output token “include” was smaller by a factor of 2.3 when the “void print\_logic(int a, int b) {” was present in the input as compared to when the line was removed from the input. This means that the line “void print\_logic(int a, int b) {” has a **negative** influence on the probability of the output token “include”.

Lets take the second line from input, and the token “\_and” from the output tokens. It has a score of +3.3. This means that the probability of the output token “\_and” was bigger by a factor of 3.3 when the second line was present in the input as compared to when the line was removed from the input. This means that the second line in the input has a **positive** influence on the probability of the output token “\_and”.

The numbers imply the magnitude of effect the input line has on the output token, whereas the sign indicates the type of effect it has – positive values means positive effect, while negative values means negative effect. A positive effect means that the presence of that set of tokens in the input increases the probability of the output token, thus increasing its chance of that token being selected as the next token during sampling stage of LLM output (as described in Step 3 above). A negative influence means that it decreases the probability of the output token, which in turn reduces the chance of that token being selected during sampling stage.

Since the number of tokens generated by the LLM will change as we change the target language for conversion, we cannot compare how the input code lines affect the output token across different target languages. Instead, we summarize the entire interpretability score above by taking the sum of interpretability scores row-wise, essentially assigning an overall interpretability score for each line in the input. These can then be compared with the summarized interpretability for other target languages as well. A positive value in such a summary context means that the input line, on average, increases the probability of output tokens in that target language. A negative value in turn means that the input line decreases the probability of output tokens in that target language.

Consider the image below –

A screenshot of a computer

Description automatically generated

Lets take the first line “void print\_logic(int a, int b) {” as an example. Java has the highest summary interpretability score of 13.42. It means that this input line increases the confidence in Java code tokens by a factor of 13.42 on aggregate. It also means that this input line increases the output token confidence the most for Java, followed by Python, Golang and Javascript. Also, the fourth input line seems to have the most negative impact on the output tokens when target language was Rust.

The same observations are also confirmed by the Shapley Value Sampling scores –

A screenshot of a computer

Description automatically generated

**How is this useful?**

**Debugging the LLM model output**

If the response generated by the LLM was acceptable, which in our case was whether the converted code was syntactically correct, could be compiled successfully and had high similarity with reference code implementations in the target language, then the above scores can help in identifying how sensitive to each line of code in the input language was important for the output code generated by the LLM.

If the code had some errors in the translated code generated by the LLM, then we can drill down to identify how the input lines affect the code generated by the LLM, and then adjust the input lines to get a correct output. Instead of changing the lines in the input code randomly, this can be done in a structured manner. Lets say, for the C to Rust conversion, the generated Rust code didn’t compile successfully. Lets look at the Feature Ablation result again –

A screenshot of a computer

Description automatically generated

We can see that the last 2 lines in input had the most negative impact on the output code generated by the LLM in Rust, whereas the 3rd input line had the most positive impact. To reduce probability of LLM generating the problematic code line in Rust, we can either try to reduce the impact of 3rd input line to reduce the probability of tokens generated by the LLM, thus reducing the chance of generating the incorrect code. Or we can try to increase the influence of 4th input line, which will also have the same effect of decreasing the output probability of code generated by the LLM. For finer details, we can then look at the interpretability scores for each output token in Rust with respect to the input tokens as below –

A graph on a white sheet

Description automatically generated

We can then identify the problematic output tokens on the x-axis, and see which of the input lines have the most positive impact on the probability of those output token. Then these input lines can be adjusted to correct the output tokens.

**LLM Reasoning**

It also provides insight into the reasoning capability of the LLM. As you can see above, irrespective of the target language, the LLM was mainly impacted by the first few lines of the source code. Since our input prompt also contained the task description that the C code was trying to solve, it only needs the first few lines to get entire context of the translation it needs to generate. In fact, for some of the languages, the last 2 lines had a negative impact on the output token probability, like Rust. It can be said that the LLM will be more confident in its code translation if those lines were not present in the input.