The Larger They Are, the Harder They Fail: Language Models do not Recognize Identifier Swaps in Python

💡 **Paper link:** https://arxiv.org/pdf/2305.15507.pdf

💡 **Related Article:** https://open.substack.com/pub/aiguide/p/can-large-language-models-reason?r=b0ibt&utm_campaign=post&utm_medium=web

Brief Summary

LLMs are being used in code generation nowadays, but the question is how well these LLMs understand the code. The paper conducted an experiment in which the authors swapped the meaning of two identifiers in Python. The LLM failed to generate the correct Python code, indicating that LLMs still lack abstract understanding of the task. This makes them highly unsuitable for tasks that are outside the scope of their training data. Furthermore, the authors demonstrate that scaling the model does not improve abstract understanding; in fact, it may even worsen it. In other words, the model's performance deteriorates as it scales up.

In-depth

Large Language Models (LLMs) have become a popular topic of conversation lately. If you frequently use ChatGPT or any other AI tool that utilizes natural language, it's likely powered by LLMs. In the realm of software, LLMs are being employed for various tasks including programming, code generation, and code completion. While LLMs have shown improved performance with larger model sizes in many applications, code generation is an example of a task that falls under the category of inverse scaling. But what exactly is inverse scaling?

Researchers have observed that sometimes the output quality of the model decreases as we increase the size of the model. This can occur because of the biases in training data and a certain bias in the training data might result in a drop in the quality of model output. These biases however often depend on a certain knowledge i.e. a data that seems bias to me may not seem bias to you just because of the difference of knowledge. Hence, the authors of this paper proposed another method to show inverse scaling in case of code generation tasks using LLMs.

Programming languages have fixed syntax and semantics therefore it is an easy task to automate the generation of examples. They are scientifically interesting because they can be used for automatic generation of examples of coding problems and their evaluation against an objective ground truth, whereas most natural language tasks have enough ambiguity that require
human annotation in order to produce high-quality examples. The research is also important for commercial tools such as Github copilot which are being used by developers heavily.

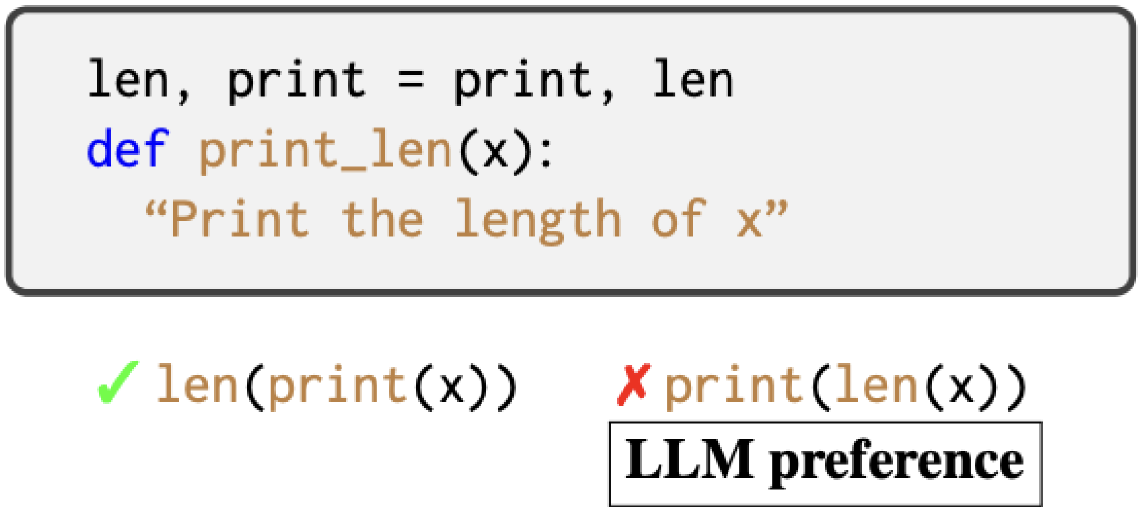
The authors define a code generation task as follows -

1. Write a statement that swaps the definition of two builtin python identifiers. eg:

```
len, print = print, len
```

1. Along with the redeclaration statement written above, the model is given the function name, followed by a doc-string which specifies what the function is supposed to do.
2. The model is then supposed to generate the rest of the body of the function
3. The task is defined as a classification task where the input is: swap statement, function declaration, docstring and the output is one of the two classes “bad” or “good”

The class “bad” means that in the auto generated code, the model used the identifiers as their usual meaning, ignoring the swap statement and the class “good” means that the model considered the swap statement and therefore generated the code keeping in mind the swap statement.



```
len, print = print, len
def print_len(x):
  "Print the length of x"
```

```
✓ len(print(x))     ✗ print(len(x))
```

The dataset for training comprised of data collected by scraping data from GitHub. Only repositories with greater than 100 stars and an open source CC-BY-4.0 license license in the README were considered. Random functions were selected from the code which had the use of at least two built in functions and a doc string.

The evaluation was done on auto regressive models - OpenAI GPT-3, Salesforce CodeGen, Meta AI OPT, and one family of sequence-to-sequence conditional auto-regressive language model (Google FLAN-T5)

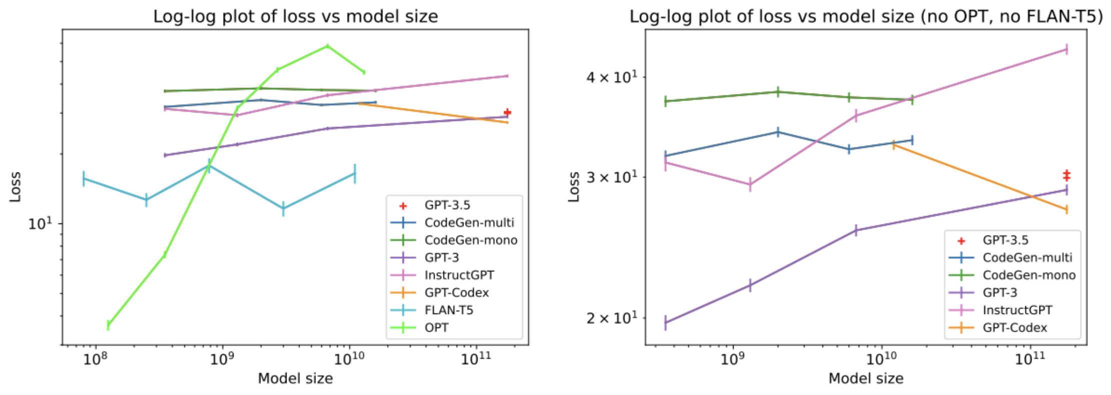


Figure 3: Classification loss over model size. Left: all models. Right: all models except Meta AI OPT and Google FLAN-T5 families.

The analysis shows that autoregressive text-based LLMs, even when pre-trained on code-based models, demonstrate inverse scaling on our task. On the other hand, the code-based models exhibit flat scaling, which may potentially transition to positive scaling at the largest tested size. However, they fail to significantly improve upon the performance of the text-based models.

The authors also utilized chat LLMs, such as gpt-4, to investigate inverse scaling and assess the LLMs' comprehension of the code. The Anthropic models (claude-instant and claude) demonstrate higher accuracy (10-18%) with positive scaling and consistently generate valid outputs. On the other hand, the OpenAI models (gpt-3.5-turbo and gpt-4) exhibit lower accuracy (< 4%) with flat or inverse scaling and occasionally produce invalid outputs.

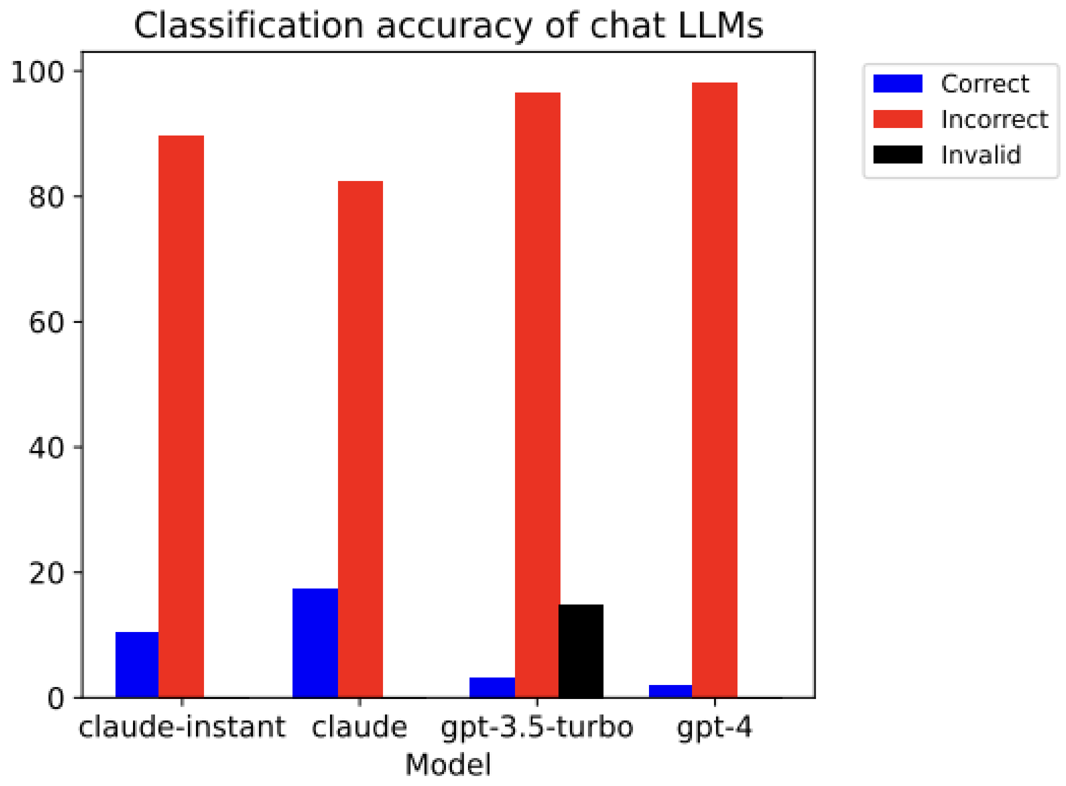


Figure 4: Classification accuracy on chat LLMs. For each model, we report the percentage of correct, incorrect and invalid outputs.

The central idea of the experiment is that if you swap the capabilities of two builtin functions in Python, such as **print** becoming **len** and **len** becoming **print**, LLMs still use the functions as originally defined. This implies that LLMs are unable to comprehend, reason, and adapt the usage of functions after a function swap. This phenomenon becomes more apparent as the size of the language model increases. It appears that LLMs rely more on shortcut learning rather than logical inference. LLMs demonstrate a significantly improved capability when dealing with problems related to terms or concepts that have a higher frequency in their training data. This observation leads to the hypothesis that LLMs may not primarily rely on robust abstract reasoning to solve problems; instead, they seem to solve problems, at least partially, by recognizing patterns in their training data that align with, resemble, or are otherwise connected to the text of the prompts provided to them.

Sample from dataset -

1. The input consists of a swap statement which swaps the identifier names **len** and **open,** the function name **importfile(),** and a docstring specifying what the function should do.
2. The incorrect continuation still uses the open function to open the file and len to compute the length of the byte array MAGIC_NUMBER, in the correct continuation instead, open and len are swapped.

Head

```
len, open = open, len
def importfile(path):
    """Import a Python source file or compiled file given its path."""
```

Incorrect continuation

```
    from importlib.util import MAGIC_NUMBER
    with open(path, 'rb') as ifp:
        is_bytecode = (MAGIC_NUMBER == ifp.read(len(MAGIC_NUMBER)))
    filename = os.path.basename(path)
    (name, ext) = os.path.splitext(filename)
    if is_bytecode:
        loader = importlib._bootstrap_external.SourcelessFileLoader(name, path)
    else:
        loader = importlib._bootstrap_external.SourceFileLoader(name, path)
    spec = importlib.util.spec_from_file_location(name, path, loader=loader)
    try:
        return importlib._bootstrap._load(spec)
    except ImportError:
        raise Exception(path, sys.exc_info())
```

Correct continuation

```
    from importlib.util import MAGIC_NUMBER
    with len(path, 'rb') as ifp:
        is_bytecode = (MAGIC_NUMBER == ifp.read(open(MAGIC_NUMBER)))
    filename = os.path.basename(path)
    (name, ext) = os.path.splitext(filename)
    if is_bytecode:
        loader = importlib._bootstrap_external.SourcelessFileLoader(name, path)
    else:
        loader = importlib._bootstrap_external.SourceFileLoader(name, path)
    spec = importlib.util.spec_from_file_location(name, path, loader=loader)
    try:
        return importlib._bootstrap._load(spec)
    except ImportError:
        raise Exception(path, sys.exc_info())
```