One-to-many testing for code generation from (just) natural language

Anonymous ACL submission

Abstract

MBPP is a popular dataset for evaluating models on the task of code generation. Despite its popularitym there are three problems with the original MBPP: (1) reliance on providing test cases to generate the right signature, (2) contamination of the exact phrasing being present in training datasets, and (3) poor alignment between instruction and evaluation testcases. To overcome this, we create MBUPP, by adapting the popular MBPP dataset for code generation from natural language to emphasize on the natural language aspect by evaluating generated code on multiple sets of assertions. Additionally, we update the text descriptions to remove ambiguity and instructions that are not evaluated by the assertions, like specific algorithms to use. This adapted dataset resolves the challenges around contamination, ambiguity and testcase alignment. Further, we compare popular open and closed weight models on the original (MBPP) and adapted (MBUPP) datasets.

1 Introduction

Code generation from natural language (NL-to-code) is a popular task to evaluate the capabilities of language models (Abdin et al., 2024; Achiam et al., 2023; Jiang et al., 2024). One of the most popular NL-to-code datasets is the *mostly basic Python programs* (MBPP) dataset (Odena et al., 2021). In this dataset, each problem contains a natural language description, a code solution and three test cases in the form of `assert` statements.

We identify three main problems with MBPP. First, it heavily relies on test cases to identify syntactic properties of the code to generate, as the provided assertions require a specific signature. Second, descriptions sometimes contain instructions that the assertions are not testing for, like asking to sort "*using heap queue*." Third, being a popular dataset distributed on many channels, data contamination is a significant issue (Riddell et al., 2024).

In this paper, we introduce an adapted code generation benchmark, called MBUPP, that allows for the description to be underspecified with respect to syntactic properties of code. Each problem consists of a text description as input to the model, and a set of assertions to validate the output. We generate both the descriptions and assertions from MBPP problems using a combination of LLMs, intuition and validation. Additionally, we provide results of different open and closed weight models on MBPP and MBUPP. We show which assertions are more often picked, indicating data contamination. Further, We release the dataset and the model generations to seed further research in this area.

We make the following contributions.

- MBUPP: An adapted version of MBPP that allows code to be underspecified and uses generalized testcases to account for that.
- An analysis of different models on MBPP and MBUPP that highlights the need for an improved code generation benchmark.

2 Motivating example

As an example, let us look at the problem *"Write a function to find sequences of lowercase letters joined with an underscore using regex"* and the associated assertions (with `f` = `text_match`)

```
assert f('aab_cbbbc') == 'Found a match!'
assert f('aab_Abbbc') == 'Not matched!'
assert f('Aaab_abbbc') == 'Not matched!'
```

Based on just the text description, it is not clear if the user expects a function `str` → `bool` (validation) or `str[]` → `str[]` (filter) or `str` → `str` (extraction). The tests also do not evaluate whether the function actually uses a regular expression or not.

Our adapted benchmark puts all emphasis on the "NL" part of NL-to-code. We assume that a user is not specific about the syntax of the program and does not care about it: they want to obtain any

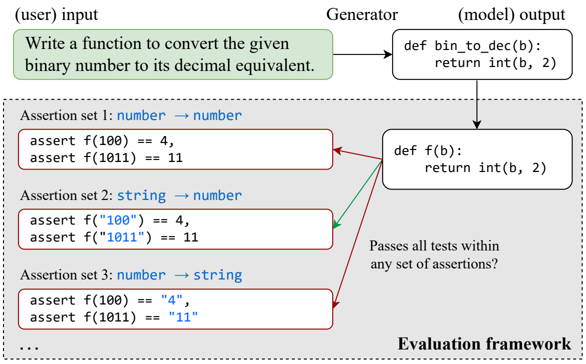

Figure 1: Example of an MBUPP benchmark problem. Given only the description, any code generator returns a function. Instead of providing the signature, which users will not likely do, we match the generated function to the signature of our assertions and then verify if the program satisfies any of the assertion sets.

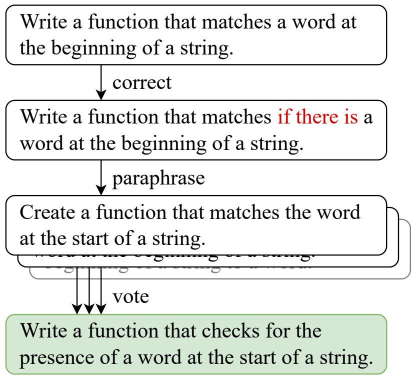

Figure 2: Improving the clarity and diversity of code generation tasks in three steps.

function that does what they describe. The adapted description is *"Write a function to find sequences of lowercase letters joined with an underscore"* with the *"using regex"* part removed. This description is the only input needed by the code generator. We therefore introduce multiple sets of assertions

```
# validator
assert f('aab_cbbbc') == True
...

# filter
assert f(['aab_cbbb',
          'aab_Abbbc']) == ['aab_cbbb']
...

# extractor
assert f('01 aab_cbbbc 23') == 'aab_cbbbc'
...
```

and consider a success if the function generated by the model (with any function name or execution semantics) passes any of the above assertion sets.

3 MBUPP

An example of an evaluation in MBUPP is shown in Figure 2. The only input to the code generator is a text description. This text description is allowed to be underspecified with respect to syntactic properties of the function, like argument order and types (data structures) used to represent the output, and we provide multiple sets of assertions that capture this underspecification. Additionally, if multiple functions are generated to solve the problem, we verify if any of them satisfies the assertions to allow the generator to use helper functions.

We adapt benchmarks in two phases: improving the text descriptions and obtaining sets of assertions to capture ambiguity on syntactic properties.

3.1 Improving descriptions

First, the original description is corrected, removing method specifiers (*"using regex"*) and ambiguity. Next, we use GPT-4 to generate three paraphrased versions using the following strategies.

- Directly paraphrasing the text description.
- Extracting structured information about the problem specification from the description (task, input type, input property, output type, output property, edge cases) in one model generation and generating a textual description from those properties in another generation.
- Similar to the previous extraction, but first instructing the model to individually paraphrase each of the pieces of task information.

Finally, we manually vote to select the best instruction. An example this process is shown in Figure 2.

3.2 Obtaining assertions

We now iteratively update the assertion sets using a combination of intuition and suggestions provided by a code generation model. Starting with the first task, we ask the model to generate multiple completions and verify if they satisfy and of the current assertions. We then inspect all failing programs and select those where the code does the right thing according to the descriptions, but not adhere to the right signature. A new assertion set is added for each mismatch. If we suspect the same mismatch in other programs, like returning a `tuple` instead of a `list`, we automatically find other assertions would be affected by this transformation and verify if they make sense.

Table 1: An overview of common assertion transformations.

Description	Before	After
Ensure list comparisons for sequences. We wrap the function in a `list` call to support any iterable.	`assert f(x) == [1,2,3]`	`assert list(f(x)) == [1,2,3]`
Permutation of arguments.	`assert f(a, b) == a + b` `assert f(m, x, y) == m[x][y]`	`assert f(b, a) == a + b` `assert f(x, y, m) == m[x][y]`
Grouping of arguments.	`assert f(m, x, y) == m[x][y]`	`assert f(m, (x, y)) == m[x][y]`
Removing redundant arguments.	`assert f(a, b) == a + 1`	`assert f(a) == a + 1`
Including selection criteria, like counts and extrema, to allow functions that *show their work*.	`assert f([1,1,2]) == 1`	`assert f([1,1,2]) == (1, 2)`
Dictionaries ↔ list of tuples	`assert f(a) == {1: 2}`	`assert f(a) == [(1, 2)]`
Validator ↔ filter	`assert f(a) == True`	`assert f([a]) == [a]`
Numbers ↔ strings	`assert f(2) == 10` `assert f(10) == 2`	`assert f(2) == "10"` `assert f("10") == "10"`

Table 2: Some examples of one-off assertions updates.

Utterance	Description	Before	After
... splits a string at lowercase letters	Original assertion has error and is ambiguous.	`f("AbCd") == ["bC", "d"]`	`f("AbCd") == ["A", "b", "C", "d"]` `f("AbCd") == ["A", "C"]`
... calculate the 4 most frequent words with their counts.	Allow both lists and strings as input.	`f(["a", "a"]) == [("a", 2)]`	`f("a a") == [("a", 2)]`

Example 1 *Consider the task to "Write a python function to detect non-prime numbers." One of the generated programs is (GPT-4,$n = 25$,$temp = 0.4$)*

```
def is_not_prime(numbers):
    return [num for num in numbers
            if not is_prime(num)]

def is_prime(num):
    # omitted
```

We rename each function to `f` *and verify whether it satisfies the (default) assertion style* `assert f(2) == True` *which fails. Since the description can be interpreted as a filter function, we add*

```
assert f([2]) == [2]
assert f([35]) == []
```

as new assertions. We then look for other problems where the assertions test for `bool` *outputs and add the new assertion if relevant.*

An overview of all common assertion transformations found in MBUPP is shown in Table 1. Some one-off transformations are shown in Table 2.

Figure 4 shows the distribution of frequency of length of updated test sets proposed in benchmark. We observe the concentration of samples with 4, 6, or 8 updated test sets, that prove to provide more possibilities of acceptable code responses. During transformation of test sets, we do a permutation and combination of all the transformations on both input and output arguments . This highlights the reason for significant amount of cases with 16 test sets, that accounts for all possible such cases.

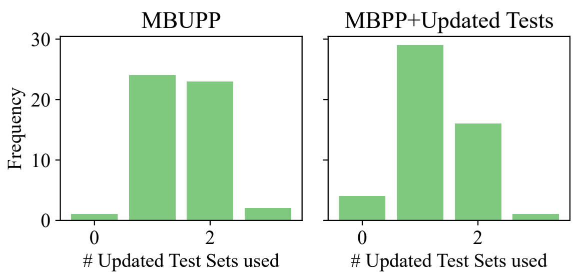

Figure 3: Distributions of unique assertion sets used by `gpt-4-turbo` at $n = 25$ and $t = 0.8$. On the utterances from MBUPP, there is more variety, which hints towards less contamination.

Model	MBPP	+ NL	+ Tests	MBUPP
`gpt-4-turbo`	0.66	0.64	0.90	0.96
`gpt-4o`	0.76	0.68	0.92	0.94
`gpt-35-turbo`	0.68	0.66	0.86	0.88
`phi`	0.58	0.60	0.80	0.84
`mistral`	0.50	0.46	0.66	0.62

Table 3: Evaluation of LLMs on the proposed MBUPP benchmark. We report the fraction of samples with pass@1>0 for $n = 25$ and $t = 0.4$. We find that all models have a higher solve-ability on MBUPP.

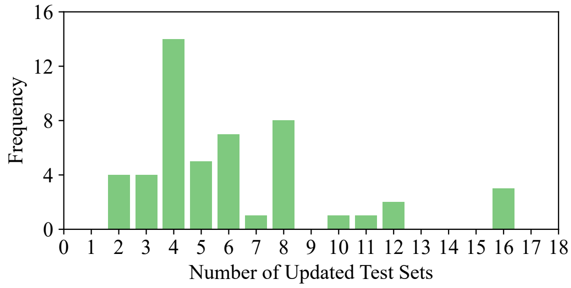

Figure 4: Distribution of number of assertion sets per task in MBUPP. MBUPP on average has 4-5 assertion sets per task showing the ambiguity in utterances.

Temperature	MBPP	+ NL	+ Tests	MBUPP
0.1	0.58	0.56	0.80	0.88
0.2	0.62	0.64	0.84	0.92
0.4	0.66	0.64	0.90	0.96
0.6	0.68	0.70	0.88	0.98
0.8	0.68	0.70	0.92	0.98

Table 4: Effect of temperature on responses with GPT-4-TURBO for $n = 25$ and different temperatures.

4 Results on MBUPP

We describe our evaluation setup, main results, and further analysis on behaviour of different models.

4.1 Evaluation setup

We use a diverse set of open and closed weight models from the `GPT`, `phi` and `mistral` series for evaluation. The input to the models is just the natural language specification alone. During evaluation we test multiple code generations ($n = 25$ and $t = 0.4$) over the updated assertion and measure *solvability* as any of these generations being correct.

4.2 Results

In this section, we discuss the impact each component of MBUPP benchmark on code generation performance.

One-to-many evaluation Table 3 shows the comparison of the number of samples being solved in MBPP versus the proposed MBUPP benchmark. We find that with updating assertion sets, there is a 45% jump in solvability of the benchmark. This is also seen in smaller models like `phi` and `mistral` which tend to have a more diverse response.

Dataset contamination MBPP being a popular and common dataset has made its way into training datasets used in larger models. This contamination in the model training set makes the performance on MBPP an unreliable indicator of model performance. Table 3 shows that with changing the NL phrasing (MBPP + Updated NL) while keeping the semantic consistent, there is a 4.5% drop in solvability, showing that the models remember the phrasing of the descriptions in the original dataset.

Effect of temperature Table 4 shows the task solve-ability for `gpt-4-turbo` with varying generation temperature. We find that performance on MBUPP increases with temperature because with updated assertion sets the generation diversity improves performance. As shown in Table 4, for lower temperature, ($t = 0.1$) the overall increase in success of model on MBUPP over MBPP is as high as $+30\%$. With the higher temperature, ($t = 0.8$) we see performance of system be all time high 98%. Less contamination allows for more diversity, which benefits from the additional assertions.

4.3 Analysis

Qualitatively looking at the generations, we find that on MBUPP, `gpt-4-turbo` failing cases are mainly attributed to logic and knowledge errors. These cases prove the efficiency of updating the test sets, ensuring to capture all possible responses of semantically acceptable code functions.

5 Conclusion

In this paper, we introduce MBUPP, an adaptation of MBPP which addresses three main challenges with the original dataset: (1) ambiguity and underspecification in the descriptions, (2) contamination of the dataset by being present in common training corpora of models, (3) poor alignment of the assertions with the description. We show results of popular open and closed weight models on the original and adapted dataset. Further, we present analysis on different components of MBUPP, diversity and temperature of the generations.

6 Limitation

The adaptation and analysis done in this work are primarily for English language and the same technique needs to be tested for other languages. This work focuses on the scenario when the specific implementation semantics are not relevant for the success of the task. For the case where the utterance needs to be made complete with required specification, we present a case study in the Appendix.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program synthesis with large language models. In *n/a*, page n/a, n/a. N/a.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*.

A Appendix

A.1 Effect of updating test sets

We observe the distribution of various transformations used while updating the test sets of which were used as possible solutions for the NL. Without the case for updated test sets we observe 44% and 56% of samples that could not be captured in corresponding NL of MBPP and MBUPP.

One of the most occurring transformation of ListToTuple signifies the impact we create by incorporating possible cases of variations in input and output type which are syntactically correct and performing the intended task correctly. Other transformations like RemoveArgs, provides variation of inclusive response handling with List size being an input parameter or not, and NumToStr, helps handling samples with binary to decimal conversion and vice-versa where the number can also be considered as string to start with.

A.2 Case Study: MBOPP

In the proposed benchmark we focus entirely on problem solving case for the various LLM, to provide in all possible responses that can be acceptable by the user with the given under-specified NL. As a followup, the NL for the task can be to translate this cleaned benchmark to the state where user mentions all details. Such specification of information would connect to user explicitly about the formatting of the arguments along with checking on the task completion for the desired Python function. We provide this set of benchmark as MBOPP (mostly basic over-specified Python programs).

For this set of benchmark the focus here is for adding more and more information to make the generated code exactly as the desired one, where the user is focused on every possible detail and formatting of the input output responses and the task.

Example for such a transformation is like: "Write a function to find the similar elements from the given two tuple lists." sample within the current MBPP benchmark being translated to "Write a function to find similar elements from two tuple lists and return a tuple.", which mentions the exact output format that the code should respond with and thus pass for all original test cases only. The latter is more specified, where the user is precise about the correct code generated.

One thing to note here is that for evaluation we only consider the generated code that passes all original test cases. The augmented test cases are not considered for evaluation to capture the instruction following of the LLM system, over the evaluation of MBUPP's task completion capability of LLM system solely where those augmented test cases where used. This contains the need for information extraction with bucketing the information present in the specification keeping the test cases in mind, to the various task components, and generate the next set of specifications with explicit mentioning of all information. This ensures mapping back any step of generation of the benchmark for quality assessment, while leveraging the LLMs with a reduced risk of hallucinations and nondeterministic behaviour.