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

Mansi Uniyal
t-muniyal@microsoft.com

Mukul Singh
singhmukul@microsoft.com

Gust Verbruggen
gverbruggen@microsoft.com

Sumit Gulwani
sumitg@microsoft.com

Vu Le
levu@microsoft.com

Abstract

MBPP is a popular dataset for evaluating the task of code generation from natural language. Despite its popularity, there are three problems: (1) it relies on providing test cases to generate the right signature, (2) there is poor alignment between instruction and evaluation test cases, and (3) contamination of the exact phrasing being present in training datasets. We adapt MBPP to emphasize on generating code from just natural language by (1) removing ambiguity about the semantics of the task from the descriptions, and (2) evaluating generated code on multiple sets of assertions to account for ambiguity in the syntax. 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[1], 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 by defining multiple sets of test cases (one-to-many testing) to account for equivalent interpretations.
- 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 are (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 function that does what they describe. The adapted

[1]The evaluation code and MBUPP are available at https://github.com/microsoft/prose-benchmarks/tree/main/MBUPP

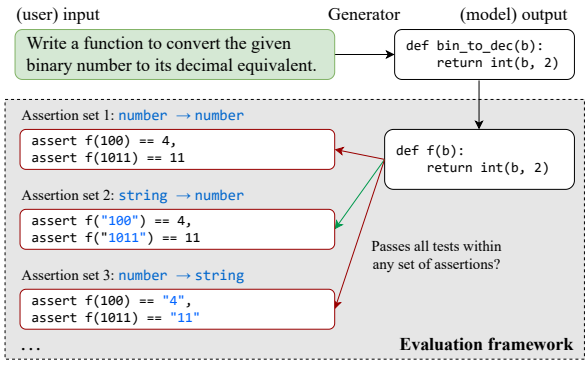

Figure 1: Example of an MBUPP benchmark problem. Given only the description, any code generator returns a function. Instead of providing the signature, which users will not likely do, we match the generated function to the signature of our assertions and then verify if the program satisfies any of the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

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.

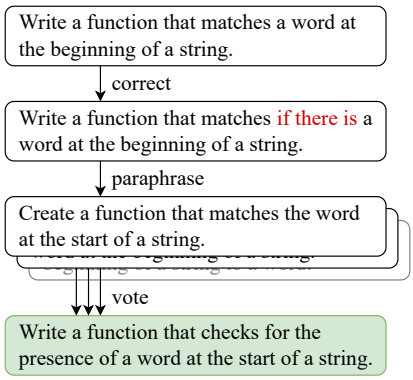

Figure 2: Improving the clarity and diversity of code generation tasks in three steps – **correcting** the original utterance, **paraphrasing** to generate diverse candidates, **voting** to select the most aligned candidate description.

3.1 Improving descriptions

First, the original description is corrected by removing method specifiers (*"using regex"*) and ambiguity. Next, we use `gpt-4-turbo` 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 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 any 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 that would be affected by this transformation and verify if they should also be transformed.

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([9,9,7]) == 9`	`assert f([9,9,7]) == (9, 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') == '2'`

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-turbo`)

```
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 3 shows the distribution of frequency of length of updated test sets proposed in benchmark. We observe problems with up to 16 updated test sets, due to combinations of transformations of both input and output arguments.

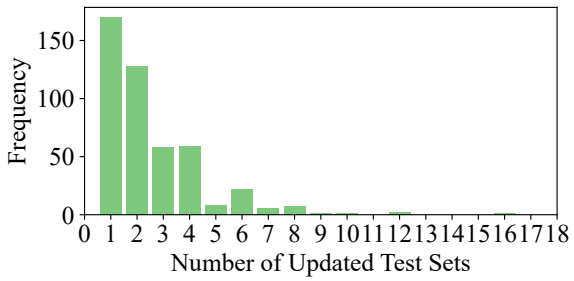

Figure 3: Distribution of number of assertion sets per task in MBUPP, which showcases the ambiguity inherently present in the original utterances.

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

Model	MBPP	+ NL	+ Tests	MBUPP
gpt-4o	62.88	65.88 (4.78↑)	**78.54** (24.91↑)	**88.20** (40.27↑)
gpt-4-turbo	58.15	63.52 (9.23↑)	74.46 (28.04↑)	84.55 (45.39↑)
gpt-35-turbo	**63.09**	**66.31** (5.10↑)	75.11 (19.05↑)	83.91 (32.99↑)
phi	54.72	61.37 (12.16↑)	69.53 (27.06↑)	81.12 (48.24↑)
mistral	41.42	47.00 (**13.47**↑)	54.94 (**32.64**↑)	63.95 (**54.40**↑)

Table 3: Evaluation of different language models on the proposed MBUPP benchmark. We report the percentage of samples with $pass@1 > 0$ for $n = 25$ and $t = 0.4$ with each component. We also show the performance improvement as a percentage over the original (shown in parenthesis). We find that all models have a higher solvability on MBUPP. We use **Bold** and Underline to indicate the best and second best result.

language specification alone. During evaluation we test multiple code generations ($n = 25$ and $t = 0.4$) over the updated assertion sets and measure *solvability* which we define 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 both language and assertion sets, there is an average of 45% jump in solvability of the benchmark. We find that the jump in solvability is much higher for smaller models, like `phi` (~ 48) and `mistral` ($\sim 54\%$), which might be more sensitive to ambiguity in utterances and generate more diverse responses.

Dataset contamination MBPP being a popular and common dataset has made its way into training datasets used in larger models (Matton et al., 2024; Riddell et al., 2024). This contamination in the model training set makes the performance on MBPP an unreliable indicator of model performance. The problem of the current MBPP benchmark NL being already exposed to training in larger `GPT` contributes to the models to perform better.

Table 3 shows that with changing the description (MBPP + Updated NL) while keeping the semantics consistent, there is an average of 9% increase in solvability over multiple models, showing impact of descriptions with intent aligned to target being more useful.

Effect of temperature Table 4 shows the task solvability 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. For lower temperature ($t = 0.1$) the overall increase in success of model on MBUPP is as high as $+52\%$, over results from MBPP. With increasing temperature, the performance differential drops as the model generates more diverse candidates and is more likely to satisfy the original assertion set. This validates the need for a diverse test set allow for equivalent programs. With higher temperature ($t = 0.8$) we see performance of system to be all time high of 87.34%. Less contamination allows for more diversity, which benefits from additional assertions. Figure 4 shows that higher temperatures (slightly) increases the diversity in test sets that are satisfied.

Temperature	MBPP	+ NL	+ Tests	MBUPP
0.1	52.36	59.87	68.24	80.04
0.2	54.51	61.16	71.67	81.97
0.4	58.15	63.52	74.46	84.55
0.6	60.52	66.09	77.47	86.91
0.8	62.02	67.17	78.33	87.34

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

4.3 Analysis

Qualitatively looking at the generations, we find that on MBUPP, the tasks that `gpt-4-turbo` still fails on are mainly attributed to 2 categories: (1) logic and (2) knowledge errors. Both of these point to the efficiency of updating the test sets, ensuring that it captures all possible responses of semantically acceptable code functions.

Logical or semantic errors occur when the model is able to correctly interpret the task but makes a mistake in the implementation of the program. For instance, when asked to *"Write a function to*

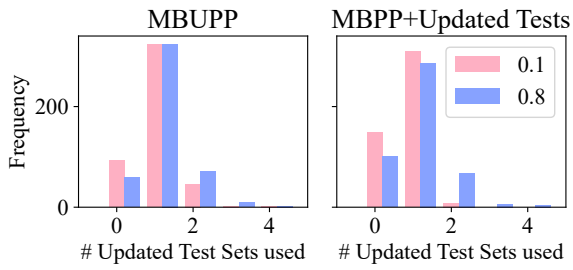

Figure 4: Distributions of unique updated assertion sets used during evaluation by `gpt-4-turbo` at $n = 25$ and $t = 0.4$. On the utterances from MBUPP, there is more variety, reduced failed cases, which hints towards less contamination. Furthermore, this behavior is magnified at higher ($t = 0.8$) temperature.

compute the n^{th} rectangular number", the model makes an arithmetic error in the formula, missing a division by 2, resulting in an incorrect solution, even though the task was interpreted correctly.

Knowledge errors occur when the program generated by the model is not aligned semantically with the user intent. For instance, when asked to *"Write a function to convert snake case string to camel case string."* the model incorrectly generates code that uses the generates mixed case outputs ("pythonProgram" instead of "PythonProgram"). Other examples are asking to check if *"a number is Woodall number"* or to *"find the n^{th} smart number."* The description is complete, but it requires specific knowledge to understand.

5 Related work

Evaluating the coding abilities of large language models is a hot topic. MBPP (Odena et al., 2021) and HumanEval (Chen et al., 2021) were among the first benchmarks and are still the most widely used. They have been translated to different languages (Cassano et al., 2023; Zheng et al., 2023; Peng et al., 2024). Newer benchmarks like NaturalCodeBench (Zhang et al., 2024) and PythonSaga (Yadav et al., 2024) consist of harder programming problems. All of these benchmarks consist of test sets used to evaluate the performance of models on these. EvalPlus (Liu et al., 2024) improves the functional testing of code generation benchmarks by generating more test cases. These benchmarks and evaluations contain a function signature and/or test cases. We focus on code generation from only natural language and adapt the tests to account for any ambiguity.

6 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. We also present analysis on different properties of MBUPP, diversity and temperature of the generations.

7 Limitation

This work focuses on the scenario where the user does not care about syntactic properties of the code, like generating a list of tuples instead of a dictionary to represent counts of items. A version of MBPP with explicit overspecificaion (MBOPP) remains future work, as well as extending MBUPP to multiple languages.

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*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On leakage of code generation evaluation datasets. *Preprint*, arXiv:2407.07565.

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.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8383–8394.

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

Ankit Yadav, Himanshu Beniwal, and Mayank Singh. 2024. Pythonsaga: Redefining the benchmark to evaluate code generating llms. *Preprint*, arXiv:2401.03855.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Yuxiao Dong, and Jie Tang. 2024. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user queries. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 7907–7928.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.