AixBench: A Code Generation Benchmark Dataset

Yiyang Hao[1], Ge Li[2], Yongqiang Liu[1], Xiaowei Miao[1], He Zong[1], Siyuan Jiang[1], Yang Liu[1], and He Wei[1]

[1]aiXcoder , {haoyiyang, liuyongqiang, miaoxiaowei, zonghe, jiangsiyuan, liuyang, weihe}@aixcoder.com
[2]Peking University , lige@pku.edu.cn

July 22, 2022

Abstract

We present a benchmark dataset for evaluating method-level code generation task. The benchmark contains a dataset of 175 samples for automated evaluation and a dataset of 161 samples for manual evaluation. We also present a new metric for automatically evaluating the correctness of the generated code, and a set of criteria to manually evaluating the overall quality of the generated code.

1 Introduction

This is a method-level benchmark for evaluating code generation models, which take natural language as input and code as output, and is primarily used to evaluate the ability of code generation models. AixBench is divided into two datasets (see Table 1):

1. Automated Test Dataset

   Each sample in this part of the dataset contains a functionally independent and well-described natural language function description, the function signature of the function, and a set of unit tests that verify the correctness of this function.

   The main use of this dataset is to automatically evaluate the correctness of the code generated by the model.

2. NL Task Description Dataset

   Each sample in this part of the data set contains a relatively independent functional description. This part of the data is closer to the real method description in the code, and contains some functional descriptions whose details are not very clear.

   Please refer to 4.2.1 for the human evaluation criteria.

Currently, these two datasets only contain Java codes, and the natural language description part contains English and Chinese languages. If you only care about code correctness, you can just use the automated test dataset.

Datasets	Automated Test Dataset	NL Task Description Dataset
Test Set Size	175	161

Table 1: Data statistics of the two datasets of AixBench.

2 Backgrounds and Related Work

Commonly used metrics, such as Exactly Match, BLEU or Perplexity, are not suited for evaluating the correctness of method-level code generation, because when considering the correctness of a program, the difference in many details between two pieces of generated code, like the name of a variable, the order of two data-flow independent instructions, the way loops and branches are written, and sometimes even the algorithm, do not directly decide which program is "correct" or not. In actual software development, people commonly use test cases to ensure a certain function works as intended. Therefore each sample of the Automated Test Dataset contains several hand-crafted automated test cases to ensure the correctness of the generated code.

The closest dataset to our Automated Test Dataset is HumanEval[CTJ+21], released by OpenAI together with Codex model. Their dataset contains 164 hand-written programming problems together with examples and test cases. However we find this insufficient to test our aiXcoder XL code generation model because of three reasons: One, their dataset is purely in Python and our model is fine-tuned on Java; Two, the problems are mostly about pure algorithm and string manipulation, which are only a small subset of real world problems. Three, the prompts in HumanEval contain examples, which is hardly written by human in a "text-code" interaction scenario.

Another dataset for the same purpose is APPS [HBK+21], which also includes description, examples, and test cases. APPS falls short for the same reasons as HumanEval: being in Python only and mostly contains algorithm and programming contests problems instead of challenges developers face in daily work.

An additional reason for us to create yet another automated code generation benchmark is that none of the datasets that we know of contains non-English prompts. And as non-native English speakers, we know well that how well a model adapts to the native language of the user affects users' accessibility by a lot.

PandasEval and NumpyEval[ZCY+22] are similar to HumanEval but more limited to specific Python packages.

Other datasets like CONCODE[IKCZ18] or PY150 do not include test cases at all.

3 Task Description

The goal of a code generation model is to generate a piece of code from a piece of natural language description. To automate the tests, we also add the signature of the desired function to the model. A signature of a function defines how this function should be called, by specifying the return type and the types and names and the order of the parameters. In this paper, we call the piece of natural language description from the input as "task".

4 Datasets

AixBench contains two datasets: Automated Test Dataset for mostly-automated code correctness evaluation and NL Task Description Dataset for manual code quality evaluation.

4.1 Automated Test Dataset

This data is a collection of hand-picked batches of "Method Comments" from open-sourced "Method Comments - Method Implementation" pairs. Our selection criteria are:

1. Comments well describe a function that can be implemented.

2. The functions are relatively independent and do not depend on the understanding of the context of the project and business logic.

3. The functionality is reasonable and could occur in a developer's day-to-day work. rather than programming competition quizzes or coursework.

4. Comments are descriptions of the objective, rather than descriptions of the implementation process.

```
1  {
2      "raw_nl": "Close Reader. If object is null it is ignored",
3      "signature": "public static void close(Reader reader)"
4  }
5  {
6      "raw_nl": "max() that works on three integers",
7      "signature": "public static float max(float a, float b, float c) "
8  }
9  {
10     "raw_nl": "将 Date 类型转为时间字符串，格式为 format",
11     "signature": "public static String date2String(final Date date, final DateFormat format)"
12 }
13 {
14     "raw_nl": "获取类上具有指定注解的接口的名称，如果有多个，则以第一个为准 找不到符合条件的接口则返回 clazz 类的名称",
15     "signature": "public static String getInterfaceName(Class<?> clazz, Class<? extends Annotation> annotation)"
16 }
```

Figure 1: Example data from the Automated Test Dataset. Test cases are implemented separately in source files.

On this basis, we extracted the descriptions in the comments, and then made some supplements, so that:

1. The description contains specific information necessary to implement the function. For example: `Returns whether or no the JDK version is high enough.` There is no clear "high enough" standard. So we added it manually as `Returns whether or no the JDK version is 1.7u40 and above.` This step is needed purely because we want to automate the tests.

2. The irrelevant part of description is deleted. For example we removed the second half of the `max() that works on three integers. Like many of the other max() functions in this class.` from the original data.

Natural language descriptions naturally will contain certain grammatical errors or punctuation or inconsistencies in capitalization. We keep these because we think these perturbations test the model's anti-disturbance ability.

Each sample in the Automated Test Dataset contains a `raw_nl` and a `signature`. `raw_nl` is the natural language description and `signature` is the name and the parameters of the desired function. `signature` exists solely because of the need of automating tests.

4.2 NL Task Description Dataset

This data is a collection of hand-picked batches of "Method Comments" from open-sourced "Method Comments - Method Implementation" pairs. Our selection criteria are:

1. Comments well describe a function that can be implemented.

2. The functions are relatively independent and do not depend on the understanding of the context of the project and business logic.

3. The functionality is reasonable and could occur in a developer's day-to-day work. rather than programming competition quizzes or coursework.

4. We allow a certain degree of ambiguity, such as in `Read the encoded image data from a JPEG image.`, we do not specify how the read data should be handled. During evaluation, as long as the code generated by the model fully implements the functions described in the description, then a full score is awarded for correctness.

4.2.1 Evaluation Criteria

We manually evaluate the code generated by the model in three dimensions.

1. Correctness:

```
 1  {
 2  ····"raw_nl": "return the last day of the date's month of specified string value in format: yyyy-MM"
 3  }
 4  {
 5  ····"raw_nl": "transform a string to a valid classname string"
 6  }
 7  {
 8  ····"raw_nl": "从 http 服务拉取并解析 Properties 文件"
 9  }
10  {
11  ····"raw_nl": "创建简单颜色选择板"
12  }
```

Figure 2: Example data from the NL Task Description Dataset.

    (a) 4 points: The specified function is fully realized.

    (b) 3 points: The main function is realized. However, some details are missing, which does not affect the correctness of the overall logic. A little modification is need to meet all the requirements.

    (c) 2 points: Only the core function is implemented. Most of the requirements are not reflected in the code. More modifications are required to meet the requirements.

    (d) 1 point: The specified function is not implemented at all.

2. Code Quality:

    (a) 3 points: The details are in place. No obviously better code in terms of performance exists. If possible, resources are released accordingly. No obvious code smell.

    (b) 2 points: Some details are not in place. There is code smell of low severity.

    (c) 1 point: There is significantly better solution in terms of performance. Or there is serious code smell.

3. Maintainability:

    (a) 5 points: The method implementation is very standardized, the variable naming is semantically straightforward, the method is not unnecessarily bloated, the readability is good, the code is short, and the code blocks are clearly structured.

    (b) 4 points: The method implementation is relatively standardized, the variable naming is basically semantically straightforward, and the readability is better.

    (c) 3 points: The method implementation meets certain specifications, some variable names are meaningless, and defective code and deprecate methods are used.

    (d) 2 points: The code is written in a confusing way, or does not follow a consistent specification, or there are many meaningless names in variable naming, or there are certain repetitions and redundant codes. Poor readability.

    (e) 1 point: Very confusing, completely illogical, hard-to-read code.

5 Experiment

In addition to pass@1, a special case of pass@k where k=1, introduced in [CTJ+21], we also use the average test cases pass ratio (AvgPassRatio) to evaluate the code generation model. $AvgPassRatio$ can be calculated like this:

$$AvgPassRatio = \frac{1}{n} \sum_{i}^{n} PassRatio_i$$

$$PassRatio_i = \frac{Count_{i,pass}}{Count_{i,total}}$$

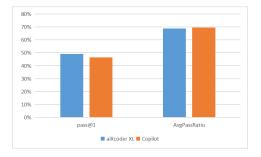| Metrics	aiXcoder XL	Copilot
pass@1	86	81
	(49.14%)	(46.29%)
AvgPassRatio	120.1979	121.7152
	(68.68%)	(69.55%)

Table 2: Automatic Comparison on Correctness over 175 samples

Figure 3: Automatic Comparison on Correctness

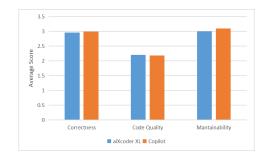| Metrics	aiXcoder XL	Copilot
Correctness	2.9503	2.9875
Code Quality	2.2049	2.1739
Mantainability	2.9937	3.0931

Table 3: Manual Comparison between aiXcoder XL and Copilot on NL Task Description Dataset

Figure 4: Manual Comparison between aiXcoder XL and Copilot on NL Task Description Dataset

where $Count_{i,pass}$ is the number of passed test cases in sample i and $Count_{i,total}$ is the total number of test cases in sample i.

We prefer AvgPassRatio over pass@k because we want to directly measure how helpful the generated code can be for developers. Intuitively, a program that passes 90% of the test cases is already good enough to help a developer implement that task, but this program will fail in pass@k because pass@k requires 100% of test cases passed.

We evaluated aiXcoder XL[aiX] and GitHub Copilot[Git] on our datasets. The results show that both model perform similarly on both automatic tests and manual evaluation.

6 Conclusion

In this paper, we present a benchmark for automatically evaluating correctness and manually evaluating overall quality of the generated code for code generation models. And we also evaluated two released code generation products, aiXcoder XL and GitHub copilot on this benchmark.

References

[aiX]   aiXcoder. aixcoder xl natural language to code demo.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam

McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[Git]   GitHub. Github copilot, your ai pair programmer.

[HBK$^+$21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[IKCZ18]   Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context, 2018.

[ZCY$^+$22] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: Continual pre-training on sketches for library-oriented code generation, 2022.