Predicting Test Results without Execution

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

ABSTRACT

As software systems grow, test suites may become complex, making it challenging to run the tests frequently and locally. Recently, Large Language Models (LLMs) have been adopted in multiple software engineering tasks. It has demonstrated great results in code generation, however, it is not yet clear whether these models understand code execution. Particularly, it is unclear whether LLMs can be used to predict test results, and, potentially, overcome the issues of running real-world tests. To shed some light on this problem, in this paper, we explore the capability of LLMs to predict test results without execution. We evaluate the performance of the state-of-the-art GPT-4 in predicting the execution of 200 test cases of the Python Standard Library. Among these 200 test cases, 100 are passing and 100 are failing ones. Overall, we find that GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. However, the results vary depending on the test complexity: GPT-4 presented better precision and recall when predicting simpler tests (93.2% and 82%) than complex ones (83.3% and 60%). We also find differences among the analyzed test suites, with the precision ranging from 77.8% to 94.7% and recall between 60% and 90%. Our findings suggest that GPT-4 still needs significant progress in predicting test results.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

software testing, large language models, LLMs, GPT-4

ACM Reference Format:
Andre Hora. 2024. Predicting Test Results without Execution. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663794

1 INTRODUCTION

Software testing is a key practice in modern software development. Developers rely on tests to avoid regressions and ensure sustainable software evolution [1, 9, 10, 17]. One important benefit of software testing is to provide fast feedback when evolving the system. For example, if any regression is inserted into the system, by continuously running the tests, developers can quickly detect and fix it [2].

Over time, as software systems grow, test suites may become complex, making it challenging to run the tests frequently and locally, on the developer's machine. For instance, the Pip project requires multiple heavy dependencies to run successively: "*Running pip's entire test suite requires supported version control tools (subversion, bazaar, git, and mercurial) to be installed*".[1] Similarly, the CPython testing documentation states the challenges of properly running its test suite: "*There could be platform-specific code that simply will not execute for you, errors in the output, etc*".[2] The Ray project documentation about tests also highlights its complexity: "*The full suite of tests is too large to run on a single machine*".[3]

To overcome these issues, a common approach is to execute the tests in CI/CD servers or containers, which can handle multiple operating systems, platforms, and dependencies [3, 7, 15]. Due to the benefit of these environments, developers rely on them to run a variety of tests, from simpler ones (*e.g.,* unit tests) to more complex tests (*e.g.,* e2e tests). One drawback of running the tests on such environments is that they may not provide fast feedback, which is fundamental when developing new features or fixing bugs [2]. Another limitation is that such environments are commonly commercial products, thus, developers may be subjected to billing (or usage limits). **In this context, it would be inestimable and unprecedented for software engineering to have the possibility to predict test results without actually executing test suites, bypassing any challenge that may exist during test run.**

Recently, Large Language Models (LLMs) have been adopted in multiple software engineering tasks [4, 6, 11, 13, 16]. Particularly, it has demonstrated great results in code generation [4, 14], however, it is not yet clear whether these models understand code execution [16]. A recent study performed by Microsoft researchers evaluated the capability of LLMs in understanding code execution by exploring code coverage prediction tasks, that is, determining which lines of a method are executed based on a given test case [16]. The study evaluated four state-of-the-art LLMs (OpenAI's GPT-4 and GPT3.5, Google's BARD, and Anthropic's Claude). The results demonstrated that GPT-4 achieved the highest performance, with 24.48% in the best-tested scenario, suggesting that LLMs still have a long way to go in predicting code coverage. However, it is unclear whether LLMs can be used to predict test results, and, potentially, overcome the issues of running real-world tests.

To shed some light on this problem, in this paper, we explore the capability of LLMs to predict test results without execution. We analyze the state-of-the-art GPT-4 because it has the best results

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663794

[1] https://pip.pypa.io/en/latest/development/getting-started/#running-tests
[2] https://devguide.python.org/testing/coverage
[3] https://docs.ray.io/en/master/getting-involved.html#testing

in code coverage prediction [16]. Specifically, we evaluate the performance of GPT-4 in predicting the execution of 200 test cases of the Python Standard Library. Among these 200 test cases, 100 are passing and 100 are failing ones.

To evaluate the prediction, we propose the following research question: *what is the performance of GPT-4 to predict test results?* Overall, we find that GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. The results vary depending on the test complexity: GPT-4 presented better precision and recall when predicting simpler tests (93.2% and 82%) than complex ones (83.3% and 60%). We also find differences among the analyzed test suites, with the precision ranging from 77.8% to 94.7% and recall between 60% and 90%. Our findings suggest that GPT-4 still needs significant progress in predicting test results. However, GPT-4 provided better results for test results prediction than for code coverage prediction [16].

Contributions. The contributions of this are threefold: (1) we provide the first study to analyze test result prediction with LLMs; (2) we discuss actionable implications; and (3) we make our dataset with 200 test cases publicly available for further studies.

2 STUDY DESIGN

2.1 Selecting Test Cases

In this study, we aim to study test cases provided by real-world test suites. For this purpose, we analyze test cases of the Python Standard Library.[4] We selected Python because it is among the most popular programming languages nowadays. The Python Standard Library was selected because its code base and documentation are among the training sources of GPT-4, therefore, its model is "aware" of this library and its inner workings.

We inspected the test suites of five important libraries (ast, calendar, csv, gzip, and string) and selected two test cases per library, totaling 10 unique test cases, as presented in Table 1. During this process, for each library, we took special care to select one simpler and one more complex test to assess the performance of GPT-4 in predicting test results in tests with distinct levels of complexity.

Table 1: Selected test cases.

Library	Test Case	Complexity
ast	`test_AST_objects`	Complex
	`test_positional_only_feature_version`	Simple
calendar	`test_locale_calendar_formatweekday`	Complex
	`test_january`	Simple
csv	`test_read_linenum`	Complex
	`test_write_simple_dict`	Simple
gzip	`test_bad_params`	Complex
	`test_fileobj_mode`	Simple
string	`test_format_keyword_arguments`	Complex
	`test_basic_formatter`	Simple

Next, for each test case, we manually modified it to create 10 passing tests and 10 failing tests, totaling 200 test cases (100 passing and 100 failing ones). To create the passing tests, we modified the inputs or outputs with distinct but valid data. Similarly, to create the failing tests, we modified the inputs or outputs with invalid data. For example, Figure 1a presents the original test case `test_basic_formatter`[5] of the string library. Figure 1b presents a modified passing version and Figure 1c shows a modified failing version. Note that both passing and failing versions contain different inputs/outputs when compared to the original test. The passing version is modified in line 7 with the valid input `"foo{0}{0}-{1}"` and its output `"foobarbar-6"`. The failing one is modified in line 5 with the incorrect output `" foo "`, *i.e.,* with extra blank spaces.

```
1. import string
2. ...
3. def test_basic_formatter(self):
4.     fmt = string.Formatter()
5.     self.assertEqual(fmt.format("foo"), "foo")
6.     self.assertEqual(fmt.format("foo{0}", "bar"), "foobar")
7.     self.assertEqual(fmt.format("foo{1}{0}-{1}", "bar", 6), "foo6bar-6")
8.     self.assertRaises(TypeError, fmt.format)
9.     self.assertRaises(TypeError, string.Formatter.format)
```

(a) Original test case.

```
1. import string
2. ...
3. def test_basic_formatter(self):
4.     fmt = string.Formatter()
5.     self.assertEqual(fmt.format("foo"), "foo")
6.     self.assertEqual(fmt.format("foo{0}", "bar"), "foobar")
7.     self.assertEqual(fmt.format("foo{0}{0}-{1}", "bar", 6), "foobarbar-6")
8.     self.assertRaises(TypeError, fmt.format)
9.     self.assertRaises(TypeError, string.Formatter.format)
```

(b) Passing test version (modified line 7).

```
1. import string
2. ...
3. def test_basic_formatter(self):
4.     fmt = string.Formatter()
5.     self.assertEqual(fmt.format("foo"), " foo ")
6.     self.assertEqual(fmt.format("foo{0}", "bar"), "foobar")
7.     self.assertEqual(fmt.format("foo{1}{0}-{1}", "bar", 6), "foo6bar-6")
8.     self.assertRaises(TypeError, fmt.format)
9.     self.assertRaises(TypeError, string.Formatter.format)
```

(c) Failing test version (modified line 5).

Figure 1: Test `test_basic_formatter` of the string library.

2.2 Creating Prompts and Assessing Answers

We select OpenAI's GPT-4 because it is state-of-the-art in code generation [14] and it has the best results in code coverage prediction [16]. The prompt used to evaluate GPT-4 in the test result prediction was provided in a natural language description of the task (as presented in Figure 2) and was inspired by a related Microsoft research [16]. Notice that we provided the exact Python version (3.10) in which the test result is supposed to be predicted.

We created a prompt for each test case and submitted them to GPT-4. Next, we read the prompt answers to assess the test result prediction. In most of the cases, the results were directly reported by GPT-4 as a passing or failing test. For example, Figure 3a presents an example in which GPT-4 directly reported that the submitted test case will fail, while Figure 3b presents an example in which GPT-4 reported a passing test case.

[4] https://docs.python.org/3/library/index.html

[5] https://github.com/python/cpython/blob/3.10/Lib/test/test_string.py#L34

```
Consider the following test of the Python Standard
Library, version 3.10:

<test_case>

Your job is to figure out whether this test will pass or
fail. If it fails, provide the rationale.
```

Figure 2: Test result prediction prompt.

```
[...]
In summary, the test will fail due to the second case
where the test incorrectly checks that the newly set
attribute `foobar` is not equal to `42`.
```

(a) Predicted failing test.

```
[...]
Given the above analysis, the test should pass without
any failures.
```

(b) Predicted passing test.

Figure 3: Examples of prompt answers.

2.3 Evaluation: Precision, Recall, and Accuracy

Table 2 summarizes the confusion matrix adopted in our prediction task. Notice that when running test cases, we want them to fail when there is some problem in the tests themselves or in the SUT. Therefore, a true positive (TP) represents a correctly predicted failing test case, while a false positive (FP) represents an incorrectly predicted failing test case (that is, a "wrong alert"). A true negative (TN) represents a correctly predicted passing test case, while a false negative (FN) represents an incorrectly predicted passing test case (that is, a "missing alert").

Table 2: Confusion matrix of test result prediction.

		Actual Test Result	
		Pass	Fail
Predicted Test Result	Pass	TN	FN
	Fail	FP	TP

We evaluate the performance of test result prediction tasks by computing precision, recall, and accuracy. We compute the precision of GPT-4 in correctly detecting failing test cases, looking for true positives and false positives; *precision = TP/(TP+FP)*. We compute the recall to verify whether GPT-4 is possibly missing to detect failing tests, i.e., the false negatives; *recall = TP/(TP+FN)*. Lastly, we compute the accuracy, which is the number of correct predictions (that is, true positives and true negatives) divided by the total number of predictions; *accuracy = TP+TN/(TP+TN+FP+FN)*.

To evaluate the prediction, we provide the following research question: *what is the performance of GPT-4 to predict test results?*. We consider (RQ1) all test cases, (RQ2) test case complexity, and (RQ3) test suite? In the first analysis, we explore all 200 test cases. In the second analysis, we divide the tests into two groups according to their complexity: 100 simpler tests and 100 more complex tests. Lastly, we divide the tests into five groups according to their test suite (*i.e.,* ast, calendar, csv, gzip, and string), each group with 40 test cases. Our dataset with the 200 test cases as well as the GPT-4 prompts and their answers is publicly available at: https://github.com/andrehora/predicting-test-results-gpt-4.

3 RESULTS

3.1 RQ1: Prediction for All Test Cases

Table 3 presents the confusion matrix for the 200 test result prediction tasks (100 passing tests and 100 failing tests). Overall, we note a large number of false negatives (29) and a few false positives (9). Regarding the correct predictions, we note 71 true positives and 91 true negatives.

Table 3: Summary of test result prediction.

		Actual Test Result	
		Pass	Fail
Predicted Test Result	Pass	91 (TN)	29 (FN)
	Fail	9 (FP)	71 (TP)

Table 4 details the precision, recall, and accuracy. Overall, when considering all test cases, we find that GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. The overall precision close to 90% indicates that GPT-4 makes a wrong prediction one time for every 10 test runs. Notice that the overall recall is low (71.0%), indicating that GPT-4 frequently misses to detect real failing tests. Finally, the accuracy at 81% shows that GPT-4 provides 4 out of 5 correct predictions.

Table 4: Performance test result prediction.

Tests	Precision	Recall	Accuracy
All	88.8%	71.0%	81.0%
Simple	93.2%	82.0%	88.0%
Complex	83.3%	60.0%	74.0%
ast	94.7%	90.0%	92.5%
csv	93.8%	75.0%	85.0%
gzip	92.3%	60.0%	77.5%
calendar	85.7%	60.0%	75.0%
string	77.8%	70.0%	75.0%

> *Summary RQ1.* Overall, GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. This means that unnoticed failing tests (*i.e.,* false negatives) are more problematic than miss-detected failing tests (*i.e.,* false positives).

3.2 RQ2: Prediction by Test Case Complexity

Table 4 details the results according to the test case complexity. In this analysis, we notice that the results vary depending on the test complexity. Interestingly, GPT-4 presented better performance

when predicting simpler tests (precision: 93.2%, recall: 82%, accuracy: 88%) than complex ones (precision: 83.3%, recall: 60%, accuracy: 74%). This means that the simpler the test case, the easier to predict the test result. However, it is important to note that, even for the simpler tests, the performance is far from the ideal 100% of accuracy.

> *Summary RQ2.* GPT-4 presented better precision, recall, and accuracy when predicting simpler tests than complex ones. However, the results are still far from 100%, meaning that even for simpler tests the results are not satisfactory.

3.3 RQ3: Prediction By Test Suite

We also find differences among the five analyzed test suites (*i.e.,* ast, calendar, csv, gizp, and string). None of the test suites presented 100% accuracy. In this analysis, the precision ranged from 77.8% to 94.7% and recall was between 60% and 90%. The best results happened for the test suite of the ast library, with a precision of 94.7%, recall of 90%, and accuracy of 92.5%. In contrast, the worst result happened for the test suites of the calendar and string libraries, both with an accuracy of 75%. Our findings suggest that GPT-4 may present a large variation in the test prediction results depending on the domain of the analyzed test suites.

> *Summary RQ3.* GPT-4 presented differences among the analyzed test suites, with the precision ranging from 77.8% to 94.7% and recall between 60% and 90%.

4 DISCUSSION AND OBSERVATIONS

Overall, GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. Moreover, GPT-4 presented better precision and recall when predicting simpler tests (93.2% and 82%) than complex ones (83.3% and 60%). We also find differences among the analyzed test suites, with the precision ranging from 77.8% to 94.7% and recall between 60% and 90%. Our findings suggest that GPT-4 still needs significant progress in predicting test results. However, the results are significantly higher than the ones obtained in code coverage prediction [16]. We hope that our results highlight the need for improvement in LLMs regarding the understanding of code execution. Further studies may replicate our test result prediction task for other programming languages and test suites.

To better understand the reasons for the false positives and false negatives, we provide some observations:

Correct analysis but incorrect conclusions. In some cases, we find that GPT-4 provided the correct explanation (rationales) for a passing or failing test, however, the final verdict was incorrect.

Reliance on comments rather than on test code. We also find cases in which GPT-4 seems to be relying on the code comments rather than the test code itself to support the prediction. This is a potential problem because comments may be wrong or outdated [12].

Explanations based "general knowledge" (rather than on code). In some cases, GPT-4 provided explanations based on "general knowledge" to complement the rationales. For example, for the test `test_positional_only_feature_version`, an answer included: "*The feature of positional-only parameters was introduced in Python 3.8. This means that using this feature in versions before 3.8 should raise a 'SyntaxError'*". Despite this being true, this is not necessarily the behavior that is actually implemented in the SUT.

Non-deterministic test results. We have not explored non determinism, but we noticed that in rare cases GPT-4 changed the test result when executing the same prompt again. This can be a potential research direction for further studies to better understand to what extent the test results are consistent over multiple executions.

5 THREATS TO VALIDITY

Data sources. In practice, GPT-4 may rely on multiple sources to classify the test results, such as source code (production and test), documentation, and general knowledge that is available on the web. This may be a source of noise for our study since we would be interested in source code only. However, we recall that the Python Standard Library was selected because its code base and documentation are among the training sources of GPT-4.

Generalization of the results. We analyzed real-world Python test cases and GPT-4 because it is state-of-the-art in code generation and code coverage prediction [14, 16]. Despite these observations, our findings may not be directly generalized to other projects or other programming languages.

6 RELATED WORK

LLMs have been adopted in multiple software engineering tasks [4, 6, 11, 13, 16]. Particularly, it has demonstrated great results in code generation [4, 14], however, it is not yet clear whether these models understand code execution [16]. A recent study performed by Microsoft researchers evaluated the capability of LLMs in understanding code execution by exploring code coverage prediction tasks, that is, determining which lines of a method are executed based on a given test case [16]. The authors evaluated four state-of-the-art LLMs (OpenAI's GPT-4 and GPT3.5, Google's BARD, and Anthropic's Claude). The results presented that GPT-4 achieved the highest performance, with 24.48% in the best-tested scenario. In our study, we explored the capability of GPT-4 to predict test results without execution. We show that GPT-4 still needs progress in predicting test results, however, GPT-4 provided better results for test results prediction than for code coverage prediction [16]. Test result prediction has been previously explored [5, 8], but not in the context of LLMs.

7 CONCLUSION

In this paper, we explored the capability of LLMs in predicting test results. We evaluated the performance of GPT-4 in predicting the execution of 200 test cases (100 passing and 100 failing ones.) of the Python Standard Library. Overall, we found that GPT-4 has a precision of 88.8%, recall of 71%, and accuracy of 81% in the test result prediction. Moreover, GPT-4 presented better precision and recall when predicting simpler tests than complex ones.

As future work, we plan to extend this research to other programming languages and test suites. We also plan to perform a qualitative analysis of the LLMs answers to get more detailed insights about the false positives and negatives.

ACKNOWLEDGMENT

This research is supported by CNPq, CAPES, and FAPEMIG.

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Simon and Schuster.

[2] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[3] CircleCI. January, 2024. https://circleci.com.

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).

[5] Ruizhi Gao, W Eric Wong, Zhenyu Chen, and Yabin Wang. 2017. Effective software fault localization using predicted execution results. *Software Quality Journal* 25 (2017), 131–169.

[6] Roar Elias Georgsen. 2023. *Beyond Code Assistance with GPT-4: Leveraging GitHub Copilot and ChatGPT for Peer Review in VSE Engineering.* Technical Report. EasyChair.

[7] GitHub Actions. January, 2024. https://docs.github.com/en/actions.

[8] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. 2005. Applying classification techniques to remotely-collected program execution data. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 146–155.

[9] Cem Kaner, Sowmya Padmanabhan, and Douglas Hoffman. 2013. *The Domain Testing Workbook.* Context Driven Press.

[10] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns.* Simon and Schuster.

[11] Jenny T Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. 2023. Can GPT-4 Replicate Empirical Software Engineering Research? *arXiv preprint arXiv:2310.01727* (2023).

[12] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship.* Pearson Education.

[13] Mauricio Monteiro, Bruno Castelo Branco, Samuel Silvestre, Guilherme Avelino, and Marco Tulio Valente. 2023. End-to-End Software Construction using ChatGPT: An Experience Report. *arXiv preprint arXiv:2310.14843* (2023).

[14] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[15] Travis-CI. January, 2024. https://www.travis-ci.com/.

[16] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting Code Coverage without Execution. *arXiv preprint arXiv:2307.13383* (2023).

[17] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.

Received 28-JAN-2024; accepted 2024-04-09