TestC: Unit Test Generation using OpenAI Codex

ISY Summer Challenge, Linköping University

Simon Johansson, Victor Löfgren, Alva Ringi, Erik Sköld

2022-08-11

Abstract

The release of OpenAI Codex, an AI system that translates natural language to code, has made a huge pre-trained neural network available to developers in various fields worldwide. In this work, OpenAI Codex is applied in software testing. We have developed *TestC*, a tool that generates unit tests based on a natural language prompt. TestC works by generating both code solutions and test cases for the problem described by the given text prompt and uses the solutions to filter out quality test cases. TestC is compared to two other unit test generation tools, Machinet and Pynguin. TestC showed better branch coverage when tested on HumanEval problems but generated more test cases than the other tools. With the support of an experienced software tester, the applicability of TestC was discussed from an industry perspective. We believe a similar tool to TestC can have benefits in software testing contexts like test-driven development. However, in order to be useful in real projects, it needs further development and be able to handle more complex problems with high enough reliability.

1 Introduction

Software testing is one of the most studied areas in the software quality assurance field [1]. Testing can be both expensive and complex, it often accounts for more than half of the software development costs [2]. An approach to avoid this is to automate testing. In order to do this, machine learning can be applied to various testing-related activities [3].

Machine learning techniques can be used to generate code. The company OpenAI has developed *Codex*, a natural language processing model which can translate natural language to code [4]. By making Codex publicly available through an API, several new tools building on this model have been released in the last year [5].

During the ISY Summer Challenge at Linköping University, the group has explored how OpenAI Codex can be used to automatically generate unit tests in Python. In this report, the development of an Codex-based test generation tool, named *TestC*, is presented. The tool is compared to existing unit test generation tools and discussed from an industrial perspective in an interview with an experienced software tester.

2 Theory

This section presents the concepts of unit testing, automatic test generation and natural language processing. The models used in the development of TestC and existing test generation tools which TestC are compared with are also presented.

2.1 Unit Testing

Unit testing means testing the smallest, testable units, often single methods or classes, of an application in isolation from each other. Methods can be either pure or impure, where pure methods does not change any states of objects, which makes them easier to test compared to impure methods [6]. Unit testing can be either black-box, where the test cases are

generated without knowledge of the implementation, or white-box where the internal code structure is examined and the tests are created to fulfill some goal of coverage (of e.g. paths, statements or branches) [7].

2.2 Evaluation of Test Cases

In most cases, it is impossible to test all possible inputs of the unit under test, and there is a need of a strategy to decide what to test. The challenge is to achieve the wanted level of coverage with as few test cases as possible [7]. There are several methods to evaluate whether a set of test cases is good or not. One method is branch coverage, it takes into account the partition of possible program branches that are visited by a set of test cases [8].

2.3 Automatic Test Generation

For test automation, both test cases, that states what inputs should be tested, and test oracles are needed. An oracle is a mechanism that can judge if the actual output from a test case is in line with the expected output, and hence, if the test case should be passed or not. Automatic generation of test oracles usually compares output from test cases with the system specification. This requires a specification that the test generator can understand, which is discussed in an article named *The Oracle Problem in Software Testing: A Survey* [9]. According to the article, many projects have specifications that are not that formal but rather expressed in natural language text. Natural language processing makes it possible to generate oracles based on requirements written in natural language [10].

2.4 Natural Language Processing

Natural language processing (NLP) is an area of research trying to create computer models that can understand natural languages. Following the resurgence of artificial intelligence research in the early 2010s, the field has made huge progress. An effective way of learning representations of words as a vector of numbers, also called embeddings, was first established [11]. Later, a method for generating sequences of text from other sequences using recurrent neural networks (RNNs), Long short-term memory (LSTM) and word embeddings was introduced [12]. This method was applicable to many areas, for example machine translation, text summarization and sentiment analysis. These models showed great promise but had difficulty in understanding very large contexts. Using the idea of attention together with RNNs, their performance improved. It was later showed that state-of-the-art performance could be achieved by only using attention [13]. These architectures based solely on attention are called transformers. They are a lot easier to train in parallel than conventional RNNs but still require a lot of time and a large sample size since they can contain hundreds of billions of parameters. Both of these areas can be improved by pre-training a model on a large unlabeled corpus of general text and recording the parameters [14]. The already computed parameters can then be used as a starting point for training it on a (much) smaller sample of labeled domain specific text.

2.5 OpenAI Codex

Codex is a model, developed by OpenAI, that translates natural language prompts into code. It was released in July 2021. Codex is based on *GPT-3* (*G*enerative *P*re-trained *T*ransformer *3*), also developed by OpenAI [4]. GPT-3 is a language model that handles natural language text as both input and output. The model consists of a pre-trained neural network with transformer architecture, it has 175 billion learnable parameters [15]. Codex is a version of GPT-3 that is fine-tuned to generate code from natural language prompts and was trained on Python code from publicly available projects on GitHub [4]. OpenAI provides an API[1] that allows companies and developers to build their own applications on top of Codex.

According to OpenAI, Codex performs best in Python but is also proficient in several other programming languages [16]. A study of GitHub Copilot, a tool powered by OpenAI Codex that provides

[1] OpenAI Codex API: `https://openai.com/api/`

code suggestions in development environments, compares the correctness of code suggestions provided in the three languages Python, Java and JavaScript and states that it performs best in Java followed by Python and worst in JavaScript [17].

2.6 HumanEval

HumanEval is a hand-written evaluation set with 164 programming problems introduced by OpenAI, it was used to evaluate the correctness of Codex. The problems needed to be hand-written since the models it is used for is trained on public code on GitHub, where you can find solutions to commonly used programming problems [4]. HumanEval is available on GitHub.

2.7 CodeT

CodeT, *Code* generation with generated *T*ests, is a method that tackles the challenge of selecting the correct or best solution to a programming problem from samples generated by pre-trained language models, such as Codex. The method is developed by researchers at Microsoft. The tool uses a pre-trained language model to automatically generate both test cases and code solutions. The test cases are used to evaluate the solutions. To choose a good solution, CodeT considers a dual execution agreement where solutions that pass many of the test cases and also have many similar solutions (siblings) are scored higher [18].

The results from evaluating CodeT, presented by the authors of the article published on the 21^{st} of July 2022, show significant improvements from previous methods [18]. CodeT was evaluated on five different pre-trained models with MBPP[2] and HumanEval benchmarks. For example, the pass@1[3], on HumanEval was improved to 65.8% with a combination of CodeT and the Codex model `code-davinci-002`, an absolute 20+% improvement over the previous state-of-the-art results [18]. CodeT is publicly available at GitHub.

2.8 Machinet

Machinet is an IntelliJ plugin, developed by *Yvision LLC*, that automatically generates unit tests in Java. By clicking on a button beside a Java method in the IntelliJ editor, Machinet generates unit tests for that method. Since the source code is available during the test generation, it is a form of white-box testing. Machinet is currently in beta and was released in October 2021 [19]. It is powered by OpenAI Codex through their API [5].

2.9 Pynguin

Pynguin is a extendable tool used to generate unit and regression tests for Python that tries maximize to code coverage [20]. It is a search-based tool based on mutation testing and genetic algorithms. This method benefits a lot from inserting type hints in the Python code.

3 Method

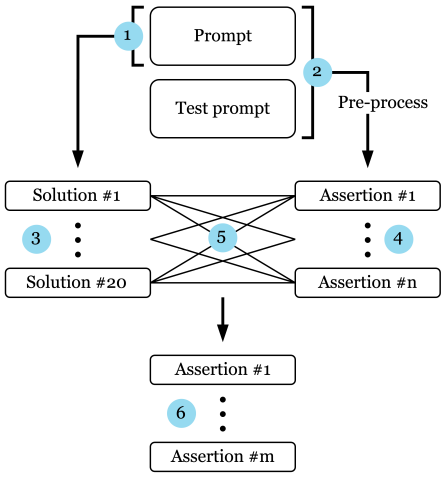
The following section describes how TestC was implemented and how it was compared to other already existing tools.

3.1 Implementation of TestC

We built a tool we call TestC that generates unit tests based on natural language prompts. The study has been limited to test generation for pure methods in Python. TestC is using a similar method to CodeT [18], but with the key difference to extract test cases instead of code generations from the model. Figure 1 shows an overview of TestC.

[2]Mostly Basic Programming Problems is a benchmark introduced by Austin et al (2021)

[3]From the *pass@k* performance metric defined by Chen et al. (2021)



1. Input prompt to OpenAI Codex for generating code solutions.
2. Input prompt to OpenAI Codex for assertion generation.
3. The prompt is executed 20 times to get 20 proposed code solutions.
4. The prompt is executed 15 times and all assertions are extracted, n ≈ 40.
5. All assertions are executed on all solutions and the test result is scored.
6. The best scoring assertions from 5 are selected as the output.

Figure 1: System overview of TestC.

3.1.1 Generate tests and code completions

Figure 2 shows what type of input was given to OpenAI Codex in order to generate both code completions and test case completions. See Figure 3 and 4 for code and test completion examples. Using these prompts, 20 code completions $\boldsymbol{x}$ and 15 test completions $\boldsymbol{y}$ were generated.

The following parameters in addition to the prompt were supplied to Codex:

`model`	`code-davinci-002`
`temperature`	0.8
`max_tokens`	600

Code completion prompt:

```
from typing import List


def has_close_elements(numbers: List[float],
    threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each other than
    given threshold.
    """
```

Additional part appended only for test completion:

```
    pass

# write high quality tests for has_close_elements
assert
```

Figure 2: Example input prompt for code and test completion for Codex.

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False
```

Figure 3: Code completion example.

```
 has_close_elements([1, 2.5, 3], 1.5)
assert not has_close_elements([1, 2, 3], 0.1)

assert has_close_elements([1, 2.5, 3, 4.5], 1.5)
assert not has_close_elements([1, 2, 3, 4], 0.1)

assert has_close_elements([5, -5], 10)
assert has_close_elements([5, -5.5], 10)
assert not has_close_elements([5, -5.6], 10)
assert has_close_elements([5, 5, -5.6], 10)
```

Figure 4: Test completion example.

3.1.2 Extract test cases from test generations

In the previous step, test completions $\boldsymbol{y}$ were generated. From those completions we extracted all syntactically correct assertion statements and removed exact duplicates. This gave us a list $\boldsymbol{z}$ containing assertions, usually around 40 assertions.

3.1.3 Execute the test cases

For all test cases $\boldsymbol{z}$ and for all code completions $\boldsymbol{x}$ each test was evaluated and stored in $\boldsymbol{w}$. The matrix $\boldsymbol{w}$, see Figure 5, contains code completions as rows and test cases as columns, 0 indicates a failed test and 1 a successful test.

$\boldsymbol{w}$
00000000000000000000000000000000
11110111101111010011001010111
00000000000000000000000000000000
11110111101111010011001010111
11110111101111010011001010111
11110111101111010011001010111

Figure 5: Example of $\boldsymbol{w}$ with 6 code generations and 30 test cases.

3.1.4 Scoring the code solutions

Using a very similar method as CodeT, we scored each code generation by first combining all rows in $\boldsymbol{w}$ that are identical, creating a new matrix $\boldsymbol{m}$ that has fewer rows. "The number of identical rows in $\boldsymbol{w}$" $= \boldsymbol{n}$ and "The number of ones in a row" $= \boldsymbol{r}$ are calculated, see Figure 6 as an example. The score is calculated as $\boldsymbol{s} = \boldsymbol{n} \odot \boldsymbol{r}$ and the highest score in $\boldsymbol{s}$ is selected. All successful tests in the corresponding row in $\boldsymbol{m}$ are the test cases that should be returned.

$\boldsymbol{s}$	$\boldsymbol{n}$	$\boldsymbol{r}$	$\boldsymbol{m}$
0	2	0	00000000000000000000000000000000
84	4	21	11110111101111010011001010111

Figure 6: Example of $\boldsymbol{s}$, $\boldsymbol{n}$, $\boldsymbol{r}$ and $\boldsymbol{m}$.

3.2 Comparison with other tools

To evaluate the performance of TestC, a comparison with Machinet and Pynguin was made.

3.2.1 Machinet

The solutions to the HumanEval problem number 0–5, was translated manually from Python to Java so that the two test generators could be tested on the same problems (TestC uses Python and Machinet uses Java). The logical structure and naming of methods and variables from the Python solutions was tried to be preserved as much as possible in order to obtain as fair a comparison as possible. Each solution was written as one Java method. The code was not complemented by any comments or doc-strings.

For each of the Java translations of the HumanEval problems 0–5, Machinet was run once to generate test cases. Machinet was configured to generate at most five test cases per method, which is the maximum possible number.

3.2.2 Pynguin

Pynguin was run once for the HumanEval problems 0–5 to generate unit tests using the default settings.

3.2.3 Analysis of branch coverage

For each of the HumanEval problems 0–5, the branches were identified for both the Python and Java implementation by manually inspecting the code. The functions from the standard libraries in Python and Java as well as from the libraries java.util and java.lang.Math used in the implementations were not further evaluated. When identifying the branches these function calls were handled as unconditional statements. The generated test cases from TestC, Machinet and Pynguin were also manually inspected and mapped to the identified branches. If at least one generated test case covered a branch, that branch was considered covered.

3.3 Interview

An interview was held with a software tester with 20 years of industry experience, five years of which as a programmer and five years as a test leader. Open-ended questions were asked to discuss TestC and gain insight to its usability in a software industry context.

4 Result

The main result of this work is our tool, TestC, which is available at `https://github.com/testc-liu/testc`. It was tested on HumanEval problems 0–5. The test cases that was generated is available in appendix A.

Our Java solutions of problem 0–5 from the HumanEval data set as well as the corresponding test cases that was generated by Machinet are available at the TestC GitHub. The test cases generated by Pynguin are also found there. The generated test data is summarized in appendix A. Figure 7 and 8 presents a summary of the number of generated test cases and the achieved branch coverage. The branch structure for the Python and Java solutions was the same for all evaluated problems.

Problem	TestC	Machinet	Pynguin
HumanEval/0	21	2	8
HumanEval/1	50	4	13
HumanEval/2	29	2	1
HumanEval/3	45	2	5
HumanEval/4	21	3	2
HumanEval/5	34	4	3

Figure 7: Number of test cases generated by TestC, Machinet and Pynguin.

Problem	TestC	Machinet	Pynguin
HumanEval/0	2/2	2/2	2/2
HumanEval/1	3/3	2/3	1/3
HumanEval/2	1/1	1/1	1/1
HumanEval/3	2/2	2/2	2/2
HumanEval/4	1/1	1/1	1/1
HumanEval/5	3/3	3/3	2/3

Figure 8: Proportion of covered branches for the test cases generated by TestC, Machinet and Pynguin.

5 Discussion

The following is a discussion of the usages of TestC, a comparison with other tools and relevant points from the conducted interview.

5.1 TestC

A large part of our work is based on OpenAI Codex and using their API has worked well apart from occasionally reaching rate limits of their private beta free tier service. Overall development of our tool was straight forward and without major issues.

TestC takes a text prompt to generate high-quality tests and a description of the intended behavior of the method to be tested as input. From the test results, see appendix A, it is notable that all generated test cases for all of the problems given, except HumanEval/1 succeeded when run on the canonical solution. This can be explained by looking at the best generated solutions for the problems. We find that for HumanEval/1, the generated code implementation made by TestC was incorrect and therefore the tests were validated against an solution that was incorrect, explaining the poor performance for HumanEval/1. For the other five problems TestC generated a correct solution and the generated tests was therefore also correct. From this it is reasonable to conclude that to be able to generate acceptable test cases using this method, correct code solutions also needs to be generated.

An application of TestC is test-driven development, since TestC does not require an implementation to generate its tests. This could reduce the developer bias when writing test cases. However, you need to put a lot of trust in the tool if you do not want to spend time reviewing the generated test cases, potentially reintroducing the bias and taking more developer time than necessary. TestC relies heavily on the performance of the underlying language model, and we see great potential for development in the field of language models moving forward.

TestC is developed to generate tests in Python. Since the input to OpenAI Codex is a text prompt that accepts common programming languages as well as nat-

ural language, it would be fairly easy to make TestC work for other programming languages as well.

5.2 Comparison with other tools

For HumanEval problem 0–5, TestC achieves 100% branch coverage. Neither Machinet nor Pynguin manages to achieve this. On the other hand, Machinet and Pynguin generates much fewer test cases than TestC. Generating and running too many test cases can be a waste of resources, it is desirable to achieve the wanted level of coverage with as few test cases as possible. To further improve TestC, it may be needed to evaluate the generated test cases and remove tests that cover the same functionality to get closer to the minimum number of test cases needed to cover all branches.

For the problem HumanEval/1, both TestC, Machinet and Pynguin generated test cases with input that contained tokens that are not handled by the methods under test. Neither the text prompt that was given to TestC nor the Java and Python methods given as input to Machinet and Pynguin mention or handle input containing letters. We therefore consider these test cases to test beyond the requirements. Such test cases is not wanted since they may fail correct implementations, which happens in the case with TestC and Machinet. Machinet also generates a test case like this for the problem HumanEval/4 where it inputs an empty array.

In two cases, one for the problem HumanEval/4 and one for HumanEval/5, Machinet generates test cases where the expected output is logically wrong given the input. The correct implementation fails these tests. This did not occur for TestC, for the six tried problems, it never generated a test case with valid input together with an expected output that did not correspond to that input.

TestC acts both as an automatic test generator and an oracle. Pynguin on the other hand only generates input data for unit tests and uses the implementation as an oracle to generate regression tests. It won't find functional bugs in already existing code this way, unless the functions happens to hang on a certain input.

Since TestC uses NLP (through Codex) to construct its tests, they are more uniform than Pynguins tests because it can only guess what arguments to use when calling a function and what the input should look like. Although tests generated by Pynguin exhibit more input variety, most of the inputs fail to conform to the expected format.

5.3 Method Criticism

TestC is a black-box testing tool since it generates the test cases without knowledge of the code implementations while Machinet and Pynguin are white-box testing tools that generate the tests based on the code. This makes it hard to compare them. Branch coverage is a term that mainly makes sense in the context of white-box testing, since the implementation is needed to identify the branches. Evaluating TestC based on code coverage of the canonical solutions in the HumanEval data set can hence be considered misleading since TestC can not take the branch structure into account during the test generation.

There is a lack of benchmarks to evaluate automatic test generators on the market [21]. We chose to use the HumanEval benchmark which is originally designed to evaluate code generation tools. With a benchmark more adapted to the purpose, maybe containing code examples containing bugs and not just correct implementations, a more usage specific evaluation could have been done. It also would have been interesting to evaluate TestC on more complex problems.

5.4 Industrial Perspective

The software tester interviewee both agreed with some of the conclusions drawn during the work and provided new insights. Given that the tool is based on text instead of code, the interviewee also believes that the tool can be helpful in test-driven development. According to the interviewee, a bottleneck today is that testers need to write test code, testers have testing knowledge but not always programming knowledge. If AI can build the tests, the testers can together with the developers produce clear require-

ment texts instead. More time can be spent on defining requirements and the risk of introducing bias is minimized.

The interviewee noted that the tests generated are based on very simple code, the methods tested in industry are rarely so clearly delimited. They were curious to see how the tool handles more complex test cases and how it affects the reliability of the tests. If the tests generated by similar tools are to be used as ground truth, they need to achieve a certain reliability.

As previously mentioned in the discussion, the tool generates a large number of test cases. The interviewee stated that several test cases were redundant and would have been deleted if tested manually. However, the interviewee also noted that with automated testing the number of tests matters less because they do not require any human resources to generate.

A possible further development the interviewee mentioned was to make the tool more flexible by, for example, allowing functions without comments to be used for test generation. It could allow the tool to be used on old software projects without the need for costly manual code refactoring and adding of comments to functions.

6 Conclusion

We have implemented a proof-of-concept tool that uses OpenAI Codex to generate unit tests for individual Python functions. It usually performed better than Machinet and Pynguin when comparing branch coverage, but it also generates more test cases. A tool like this could be used in test-driven development to save development resources but then a high level of reliability needs to be ensured. More development in this area is expected to hugely impact the performance of similar tools in the future.

References

[1] Alessandro Orso and Gregg Rothermel. "Software Testing: A Research Travelogue (2000–2014)". In: FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 117–132. ISBN: 9781450328654. DOI: 10.1145/2593882.2593885. URL: https://doi.org/10.1145/2593882.2593885.

[2] Dianxiang Xu, Weifeng Xu, Michael Kent, Lijo Thomas, and Linzhang Wang. "An automated test generation technique for software quality assurance". In: *IEEE transactions on reliability* 64.1 (2014), pp. 247–268.

[3] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego R. C. Dias, and Marcelo P. Guimarães. "Machine Learning Applied to Software Testing: A Systematic Mapping Study". In: *IEEE Transactions on Reliability* 68.3 (2019), pp. 1189–1212. DOI: 10.1109/TR.2019.2892517.

[4] 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. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[5] OpenAI. *Powering Next Generation Applications with OpenAI Codex*. Ed. by OpenAI. URL: https://openai.com/blog/codex-apps/ (visited on 08/03/2022).

[6] Runze Yu, Youzhe Zhang, and Jifeng Xuan. "MetPurity: A Learning-Based Tool of Pure Method Identification for Automatic Test Generation". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, pp. 1326–1330.

[7] Bui Thi Mai Anh. "Enhanced Genetic Algorithm for Automatic Generation of Unit and Integration Test Suite". In: *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)*. 2020, pp. 1–6. DOI: 10.1109/RIVF48685.2020.9140778.

[8] Ilia Zlatkin and Grigory Fedyukovich. "Maximizing branch coverage with constrained horn clauses". In: *International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems. Springer. 2022, pp. 254–272.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785.

[10] Manish Motwani and Yuriy Brun. "Automatically Generating Precise Oracles from Structured Natural Language Specifications". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 188–199. DOI: 10.1109/ICSE.2019.00035.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27 (2014).

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[14] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. "Improving language understanding by generative pre-training". In: (2018).

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[16] Greg Brockman Wojciech Zaremba and OpenAI. *OpenAI Codex*. Ed. by OpenAI. URL: https://openai.com/blog/openai-codex/ (visited on 08/03/2022).

[17] Nhan Nguyen and Sarah Nadi. "An Empirical Evaluation of GitHub Copilot's Code Suggestions". In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE. 2022, pp. 1–5.

[18] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. *CodeT: Code Generation with Generated Tests*. 2022. DOI: 10.48550/ARXIV.2207.10397. URL: https://arxiv.org/abs/2207.10397.

[19] Machinet. *Machinet*. Ed. by Machinet. URL: https://machinet.net/ (visited on 08/03/2022).

[20] Stephan Lukasczyk and Gordon Fraser. "Pynguin: Automated Unit Test Generation for Python". In: *CoRR* abs/2202.05218 (2022). arXiv: 2202.05218.

[21] Afonso Fontes and Gregory Gay. "The Integration of Machine Learning into Automated Test Generation: A Systematic Literature Review". In: *arXiv preprint arXiv:2206.10210* (2022).

Appendix A Test Case Evaluation

Below are tables containing all generated test cases for the HumanEval problems 0–5. The column "Input" presents the data that should be handed as input to the method under test and "Output" is the expected output. The column "Fail" presents whether the test case was failed when run on the canonical solution (that means the Python implementation included in the HumanEval data set for TestC test cases and our corresponding Java translations for Machinet test cases).

HumanEval/0

TestC

Input	Output	Fail
[1, 1], 2	True	
[1, 2, 3, 4, 10], 2	True	
[1, 2, 3, 4, 5, 1, 2], 1	True	
[1, 2, 3, 4, 5, 1, 2], 1.5	True	
[1, 2, 3, 4, 5], 0.5	False	
[1, 2, 3, 4, 5], 1.5	True	
[1, 2.5, 3, 4.5], 1.5	True	
[1, 2.5, 3], 1.5	True	
[1.0, 1.2, 1.1], 0.2	True	
[1.0, 1.2, 1.4], 0.2	True	
[1.0, 2.0, 3.0, 4.0, 5.0], 1.5	True	
[1.0, 2.0, 3.0], 0.1	False	
[1.0], 0.2	False	
[1.1, 1.2], 0.2	True	
[10, 11, 12, 13, 14], 1.5	True	
[1], 1	False	
[42, 42], 1	True	
[5, 5, -5.6], 10	True	
[], 0.2	False	
[], 2	False	
[5, 10, 90], 100	True	

Machinet

Input	Output	Fail
[1, 2, 3, 4, 5], 0.5	False	
[1, 2, 3, 4, 5], 1.5	True	

Pynguin

Input	Output	Fail
[None], None	False	
[], -374.0	False	
[None, None, -724.8, -724.8], -452.527	Error	
[-3875.508308], -3875.508308	False	
[-3875.508308, 363.0, 363.0], 363.0	True	
[-3875.508308], -3255.0	False	
[-2162.0977], -426.29	False	
[-2162.0977, -2162.0977], -2162.0977	False	

HumanEval/1

TestC

Input	Output	Fail
""	[]	
"((((()))"	["((((()))"]	
"(((((((())))"	["(((((((())))"]	Yes
"(((((((()))))()"	["(((((((()))))()"]	Yes
"((((()()))"	["((((()()))"]	
"((()())"	["((()())"]	
"(((()))"	["(((()))"]	
"(()(((()))))())"	["(()(((()))))())"]	
"(()((()()))"	["(()((()()))"]	
"(()(()()))"	["(()(()()))"]	
"(()((()))(()(()))"	["(()((()))(()(()))"]	Yes
"(()()()())"	["(()()()())"]	
"(()())"	["(()())"]	
"(()())(()())"	["(()())(()())"]	Yes
"(())"	["(())"]	
"(())(()())"	["(())(()())"]	Yes
"(())(())"	["(())(())"]	Yes
"(())(())(()())"	["(())(())(()())"]	Yes
"(())(())(())"	["(())(())(())"]	Yes
"(())(())(())(())"	["(())(())(())(())"]	Yes
"(())(())(())(())(())"	["(())(())(())(())(())"]	Yes
"((a)(bc)((de)(f)(gh))) (af) (((a)))"	["((a)(bc)((de)(f)(gh)))", "(af)", "(((a)))"]	Yes
"((a)(bc)((de)(f)(gh))) (af)"	["((a)(bc)((de)(f)(gh)))", "(af)"]	Yes
"((a)(bc)((de)(f)(gh)))"	["((a)(bc)((de)(f)(gh)))"]	Yes
"((a))"	["((a))"]	Yes
"() (())"	["()","(())"]	
"() () ()"	["()","()","()"]	
"() ()"	["()","()"]	
"()"	["()"]	
"()()"	["()()"]	Yes
"())())"	["())())"]	Yes
"(a) (d,e,f)"	["(a)","(d,e,f)"]	Yes
"(a)"	["(a)"]	Yes
"(a,b,c) (d,e,f) (g,h,i) (j,k,l) (m,n,o) (p,q,r)"	["(a,b,c)","(d,e,f)","(g,h,i)","(j,k,l)","(m,n,o)","(p,q,r)"]	Yes
"(a,b,c) (d,e,f) (g,h,i) (j,k,l) (m,n,o)"	["(a,b,c)","(d,e,f)","(g,h,i)","(j,k,l)","(m,n,o)"]	Yes
"(a,b,c) (d,e,f) (g,h,i) (j,k,l)"	["(a,b,c)","(d,e,f)","(g,h,i)","(j,k,l)"]	Yes
"(a,b,c) (d,e,f) (g,h,i)"	["(a,b,c)","(d,e,f)","(g,h,i)"]	Yes
"(a,b,c) (d,e,f)"	["(a,b,c)","(d,e,f)"]	Yes
"(a,b,c)"	["(a,b,c)"]	Yes
"a(b(c))"	["a(b(c))"]	Yes
"(()()((()))())"	["(()()((()))())"]	
"(()()(()()(((((())))()(((()))"	["(()()(()()(((((())))()(((()))"]	Yes
"(()())"	["(()())"]	
"(())"	["(())"]	
"()"	["()"]	
"(abc)"	["(abc)"]	Yes
"(bar)"	["(bar)"]	Yes
"(foo(bar))"	["(foo(bar))"]	Yes

Machinet

Input	Output	Fail
"((((()))(((()))))"	["((()))", "((()))", "((()))"]	
"((()))(((())"	["((()))", "((()))"]	
"(1)"	["(1)"]	Yes
""	[]	

Pynguin

Input	Output	Fail
"pj>a$\x0bE&wniVrsk``X3"	*	
"_RG|.Bb!\"A+O(\^\}t"	*	
"x/10r'G`9!"	*	
"x/10r'G`9!"	*	
"8 "	[]	
"x/10r'G`9!"	*	
"W\n"	*	
"xfR>n),51>ny^Yh>3\\"	[]	
"W\n"	*	
"N27()0s^#\x0c8sDPk>[Q"	["()"]	
"N27()0s^#\x0c8sDPk>[Q"	["()"]	
"N27()0s^#\x0c8sDPk>[Q"	["()"]	
None	Error	

HumanEval/2

TestC

Input	Output	Fail
-5	0	
-5.0	0.0	
.5	0.5	
0	0	
0.0	0.0	
0.001	0.001	
0.1	0.1	
0.28	0.28	
0.32	0.32	
0.345	0.345	
0.49	0.49	
0.50	0.5	
0.999	0.999	
1	0	
1.0	0.0	
1.5	0.5	
1.54	.54	
1.99	.99	
1.99	0.99	
1.9999	0.9999	
10.0	0	
10.0	0.0	
2	0	
2.0	0.0	
28	0.0	
5	0	
5.0	0.0	
5.5	0.5	
7.5	0.5	

Machinet

Input	Output	Fail
1.5	0.5	
5.0	0.0	

Pynguin

Input	Output	Fail
-3868.6005	0.3995 +- 0.01	

HumanEval/3

TestC

Input	Output	Fail
[-1, -1]	True	
[-1, 5, 0, 1, 0, 5, -1]	True	
[-1,-1,-1]	True	
[-1,-2,-3,-4,-5]	True	
[-1,1,2,-1,-1,1,2]	True	
[-1,1,2,3]	True	
[-10, -5, -4]	True	
[-1]	True	
[-2, 3, -7]	True	
[-20, 1, 2, 3]	True	
[0]	False	
[1, -1, -1, -1, -1, -1, -1]	True	
[1, -1, -1, 1, -1, 1, -1]	True	
[1, -1, -1, 1, -1]	True	
[1, -1, -1, 1]	True	
[1, -1, -1]	True	
[1, -1, -3, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, -1, -1]	True	
[1, -1, -3, 1, -1, -1, 1, -1, 1, -1, -3, 1, -1, -1, 1, -1]	True	
[1, -1, -3, 1, -1, -1, 1, -1]	True	
[1, -1, -3]	True	
[1, -1, 2, -3, 5, -10, 10, 10, -20]	True	
[1, -1, 3]	False	
[1, -1]	False	
[1, 1, 1, -3, 4]	False	
[1, 1, 2, 1, -1]	False	
[1, 2, 3, -7]	True	
[1, 2, 3, 4, 5, 10]	False	
[1, 2, 3, 4, 5]	False	
[1,-1,1,-1,-1]	True	
[1,-1,1]	False	
[1,1,1]	False	
[1,2,3,4,5]	False	
[1,2,3]	False	
[10, -20, 30, -40, 50, 60, -70, 80, 90, -100, 110]	True	
[10, 0, 0, -1, 1]	False	
[100, -100, -100, 100, -100]	True	
[100, -100, 100]	False	
[100]	False	
[1]	False	
[4, -7, -1]	True	
[4, 5, -7, 1]	False	
[4, 5, 7]	False	
[5, 0, 1, 0, 5, -1]	False	
[]	False	
[]	False	

Machinet

Input	Output	Fail
[1, 2, 3]	False	
[-1, -2, -3]	True	

Pynguin

Input	Output	Fail
[-2183, -726, -726]	True	
[]	*	
[]	*	
[2369, 2369, -1450]	False	
[2369]	False	

HumanEval/4

TestC

Input	Output	Fail
[-10, -10, -11, -11]	0.5	
[0, -1]	0.5	
[0, 0, 0]	0	
[0.0]	0.0	
[0]	0	
[1, -1, 1, -1]	1.0	
[1, 1, 1, 1, 1]	0	
[1, 1, 1]	0	
[1, 1, 1]	0.0	
[1, 1]	0	
[1, 2, 3, 4]	1	
[1, 2]	0.5	
[1, 3]	1	
[1.0, 1.0, 1.0]	0.0	
[1.0, 2.0, 3.0]	0.6666666666666666	
[1]	0	
[1]	0.0	
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	0.0	
[2, 2, 2]	0	
[3, 3, 3]	0	
[50]	0	

Machinet

Input	Output	Fail
[1.0, 2.0, 3.0]	1.0	Yes
[1]	0	
[]	0	Yes

Pynguin

Input	Output	Fail
[-249.18306, -249.18306]	0 +- 0.01	Yes
[]	Error	

HumanEval/5

TestC

Input	Output	Fail
[-1,-2,-3], -5	[-1, -5, -2, -5, -3]	
[1, 2, 3, 4, 5], 0	[1, 0, 2, 0, 3, 0, 4, 0, 5]	
[1, 2, 3, 4], -1	[1, -1, 2, -1, 3, -1, 4]	
[1, 2, 3, 4], 0	[1, 0, 2, 0, 3, 0, 4]	
[1, 2, 3, 4], 100	[1, 100, 2, 100, 3, 100, 4]	
[1, 2, 3], -1	[1, -1, 2, -1, 3]	
[1, 2, 3], 0	[1, 0, 2, 0, 3]	
[1, 2, 3], 1	[1, 1, 2, 1, 3]	
[1, 2, 3], 4	[1, 4, 2, 4, 3]	
[1, 2], 0	[1, 0, 2]	
[1,2,3,4], -1	[1, -1, 2, -1, 3, -1, 4]	
[1,2,3,4], 0	[1, 0, 2, 0, 3, 0, 4]	
[1,2,3,4], 101	[1, 101, 2, 101, 3, 101, 4]	
[1,2,3,4], 5	[1,5,2,5,3,5,4]	
[1,2,3], 0	[1, 0, 2, 0, 3]	
[1,2,3], 0	[1,0,2,0,3]	
[1,2,3], 10	[1,10,2,10,3]	
[1,2,3], -1	[1,-1,2,-1,3]	
[1,2,3], 0	[1,0,2,0,3]	
[1,2,3], 1	[1,1,2,1,3]	
[1], -1	[1]	
[1], 1	[1]	
[1], 100	[1]	
[1], 5	[1]	
[3,5,7,11,13], 0	[3, 0, 5, 0, 7, 0, 11, 0, 13]	
[9,9,9], 0	[9,0,9,0,9]	
[], -1	[]	
[], 0	[]	
[], 1	[]	
[], 100	[]	
[], 101	[]	
[], 42	[]	
[], 5	[]	
[], 1	[]	

Machinet

Input	Output	Fail
[1, 2, 3], 0	[*, *, *]	Yes
[1, 2], 0	[1, 0, 2]	
[1], 2	[1]	
[], 1	[]	

Pynguin

Input	Output	Fail
[-1536, -1536, -1536], 10	[-1536, 10, -1536, 10, -1536]	
[], -1096	[]	
None, -372	[]	