Representation and Generation of Machine Learning Test Functions

Souha Ben Hassine and **Steven R. Wilson**
School of Engineering and Computer Science
Oakland University, Rochester, MI, USA
`{souhabenhassine,stevenwilson}@oakland.edu`

Abstract

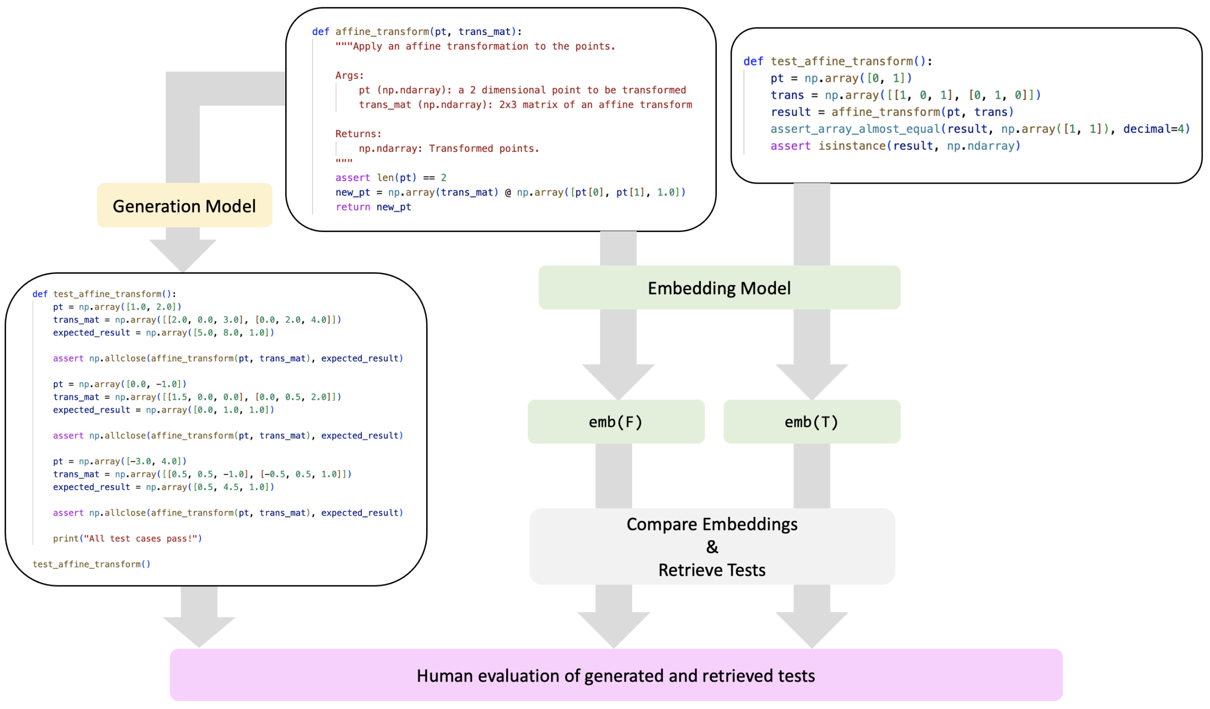
Writing tests for machine learning (ML) code is a crucial step towards ensuring the correctness and reliability of ML software. At the same time, Large Language Models (LLMs) have been adopted at a rapid pace for various code generation tasks, making it a natural choice for many developers who need to write ML tests. However, the implications of using these models, and how the LLM-generated tests differ from human-written ones, are relatively unexplored. In this work, we examine the use of LLMs to extract representations of ML source code and tests in order to understand the semantic relationships between human-written test functions and LLM-generated ones, and annotate a set of LLM-generated tests for several important qualities including usefulness, documentation, and correctness. We find that programmers prefer LLM-generated tests to those selected using retrieval-based methods, and in some cases, to those written by other humans.

1 Introduction

As AI and ML become more and more integrated into everyday processes, ensuring the quality and reliability of these ML models is mandatory, and a critical part of ensuring ML models' performance in production is having good, representative test cases. Traditionally, these tests have been written by engineers and programmers, a process that, while valuable, can be time-consuming and requires extensive experience and expertise in ML methodology. Recognizing the challenges posed by the intricacies of ML code, particularly the distinct nature of ML testing involving both pre-training and post-training tests, our research takes a deliberate focus on this specific domain. This choice serves to constrain the scope of our investigation and allows us to address the unique complexities associated with ML testing, which often deviates from conventional software testing.

One possible way to aid programmers is to retrieve existing functions that have been previously implemented, similarly to what has been done for test case selection within a test suite (Romano et al., 2018). For the purpose of writing tests, relevant test cases could be retrieved from other projects that are written to test functions that are semantically similar to the programmers' target functions. These retrieved functions might serve as references for programmers to consider when developing their own tests. However, with the recent advent of powerful code-generating LLMs such as Codex (Chen et al., 2021) and LLaMA (Touvron et al., 2023), those seeking to develop ML test cases are now able to prompt the model given the source function and instructions required to produce the appropriate test case. This has the potential to revolutionize the way that ML tests are developed, and it is therefore important to analyze how AI-generated tests compare to those written by humans and how developers may consider using these methods.

In this work, we make initial steps toward comparing the ML test functions that are generated by LLMs with those generated by human programmers to better anticipate the consequences of a growing number of ML test functions being generated automatically by LLMs. Using a set of approximately 10,000 pairs of ML functions and their tests, we use code embedding methods to explore the semantic relationships between functions and their tests. We then experiment with semantic retrieval-based approaches to find relevant ML tests given an input test function, and finally, we compare several models' ability to generate useful ML test functions and evaluate them using expert human annotations. An overview of the process that we used is presented in Figure 1. Focusing on the specific domain of ML allowed us to make the focal methods more comparable and facilitated



```
def affine_transform(pt, trans_mat):
    """Apply an affine transformation to the points.

    Args:
        pt (np.ndarray): a 2 dimensional point to be transformed
        trans_mat (np.ndarray): 2x3 matrix of an affine transform

    Returns:
        np.ndarray: Transformed points.
    """
    assert len(pt) == 2
    new_pt = np.array(trans_mat) @ np.array([pt[0], pt[1], 1.0])
    return new_pt
```

```
def test_affine_transform():
    pt = np.array([0, 1])
    trans = np.array([[1, 0, 1], [0, 1, 0]])
    result = affine_transform(pt, trans)
    assert_array_almost_equal(result, np.array([1, 1]), decimal=4)
    assert isinstance(result, np.ndarray)
```

```
def test_affine_transform():
    pt = np.array([1.0, 2.0])
    trans_mat = np.array([[2.0, 0.0, 3.0], [0.0, 2.0, 4.0]])
    expected_result = np.array([5.0, 8.0, 1.0])

    assert np.allclose(affine_transform(pt, trans_mat), expected_result)

    pt = np.array([0.0, -1.0])
    trans_mat = np.array([[1.5, 0.0, 0.0], [0.0, 0.5, 2.0]])
    expected_result = np.array([0.0, 1.0, 1.0])

    assert np.allclose(affine_transform(pt, trans_mat), expected_result)

    pt = np.array([-3.0, 4.0])
    trans_mat = np.array([[0.5, 0.5, -1.0], [-0.5, 0.5, 1.0]])
    expected_result = np.array([0.5, 4.5, 1.0])

    assert np.allclose(affine_transform(pt, trans_mat), expected_result)

    print("All test cases pass!")

test_affine_transform()
```

Figure 1: Overview of the process followed in this paper.

the recruitment of participants familiar with ML algorithms. Our core contributions include: (1) exploration of semantic representations for ML functions and their tests; (2) experiments with retrieval of ML tests; and (3) a human evaluation of LLM-generated test cases. We identified that there is a non-arbitrary relationship between the embeddings of ML functions and their test cases, but neural-network-based retrieval approaches were not able to leverage these representations effectively. However, our results show that programmers prefer LLM-generated tests to those selected using retrieval-based methods, and in some cases, to those written by other humans.

2 Related Work

2.1 Code Representation Learning and Embedding Models

The focus on learning distributed representations of code forms the groundwork of our research. We draw from Alon et al. (2019)'s work on code2vec which demonstrated the potential of learning code embeddings using neural networks. As transformer-based approaches become more popular, CodeBERT (Feng et al., 2020) used bidirectional encoder layers and the authors of the model introduced a large-scale dataset and providing insights about the learning of code semantics. GraphCodeBERT extended that work via the integration of graph-based structural and lexical information to improve the representation of the code (Guo et al., 2020). Later, the CODET (Chen et al., 2022) model tackled the challenge of code generation while also generating unit tests for Java code, demonstrating the potential of multi-faceted code generation for test cases.

It's also important to mention the effort on benchmarking datasets like CodeSearchNet (Husain et al., 2019) and CodeXGLUE (Lu et al., 2021), which offer robust platforms for training and evaluating numerous models in this area. However, none of the previous evaluation datasets checked for the quality of ML function and test matching, that is, evaluating which approaches are best able to retrieve a test case given an input function (rather than a search query written in natural language). Also, the aforementioned methods are not full language models that can generate code for any language. Some only work on one language and are not necessarily applicable to the ML domain.

2.2 Applications and Evaluation of Large Language Models on Source Code

Substantial research has been invested in revealing the power of LLMs in dealing with code-related tasks, from code summarization to test generation and beyond. Supported by billions of trainable parameters and extensive publicly available source code, models like StarCoder (Li et al., 2023) and LLaMA (Touvron et al., 2023) are carving a new path. These models have shown promising results in code generation, thanks to the vast resources at their disposal.

Previous work (Schäfer et al., 2023) shows how LLMs can be used to generate unit tests for Javascript code. Integral to the understanding and broader adoption of these models is the systematic evaluation of their performance. This aspect has been explored by Xu et al. (2022), who showed that Codex displayed superior performance compared to other models tested on the HumanEval (Chen et al., 2021) benchmark.

Additionally, a previous study (Liu et al., 2023) underscores the effectiveness of the HumanEval benchmark in identifying substantial instances of incorrect code generated by LLMs that had previously gone unnoticed.

These works offer valuable insights into the effectiveness of these emerging models, highlighting their capabilities in understanding syntax, pattern recognition, and automation, while also bringing to light their limitations, such as their lack of true understanding, difficulty with complex logic, and challenges with generalizability and interpretability when interacting with code. However, previous applications haven't focused on the unique properties of ML tests (Riccio et al., 2020), this paper aims to bridge that gap and delve into these distinctive features.

3 Data

3.1 Data Collection

We collected a dataset of 56,889 test files extracted from 986 different GitHub ML projects written in Python using the GitHub API [1]. The projects were selected if they use at least one of the Python ML libraries, such as Scikit-Learn, TensorFlow, Theano, Caffe, Keras, or PyTorch. All of these projects were created between January 1, 2007, and September 22, 2022, with three or more contributors. These projects encompassed a wide range of ML code, including personal ML projects and well-known ML libraries or frameworks such as Hummingbird, fvcore, and Sentence Transformers. The dataset contains a fair number of ML tests, making it a valuable resource for analyzing ML test functions, and exploring their characteristics. However, it lacked explicit mappings between individual functions and their corresponding tests, which is a requirement if we seek to analyze the relationships between these types of functions.

[1]https://docs.github.com/en/rest

3.2 Data Preprocessing

In order to link ML functions and their corresponding tests, we applied several heuristics to automate the extraction process:

1. Assume that each test function name begins with 'test', 'Test', or '_test'.

2. Assume that if a test function calls only one of the functions defined within the project, it is testing that specific function.

3. Ignore single-character function names to help remove noisier and less clear examples.

While these rules may filter out some valid test cases, we selected them in order to aim for a high precision in terms of returning a quality set of pairs between *focal methods* and *tests*. In this work, we refer to an ML function undergoing testing as a "focal method", and its corresponding ML test case a "test". We also removed some pairs (approximately 150) that contained accents, emojis, or symbols like progress bars, which made them more difficult to process. After applying the heuristics defined above, we were left with 10,324 (focal_method, test) pairs. Around 5% of the focal methods have multiple tests, while the tests themselves are unique to the project and no test is considered to be testing multiple methods.

Certain types of pairs could not be collected, e.g., when a test is testing the behavior of a predefined model or functions that are not defined within the project. To evaluate this process, we selected a random sample of 100 (focal_method, test) pairings and manually labeled whether each pairing was correct, meaning that the test does test the function it was associated with, and found that the pairing method was 95% accurate.

4 Building Representations for Test Cases and Retrieval Task

4.1 Building Embeddings and Investigating Pairing Relationships

To focus on the relationship between the focal methods and their associated tests, we created embeddings for each focal method using models trained on both code and natural language data. These models included CodeBERT (Feng et al., 2020), text-embedding-ada-002,[2] and LLaMA-1 (Touvron et al., 2023) with 7 billion parameters.

An essential aspect of our exploration involved understanding the semantic relationships between pairs of focal methods' and associated tests' vector representations. Each of the models we used produced embeddings with different shapes (CodeBERT: 768, LLaMA-1: 4096, text-embedding-ada-002: 1536), but for the purpose of visualization, we used Principal component analysis (PCA) to reduce their dimensions to (2).

We visualized these pairings using an arrow plot where each focal method embedding is connected to its corresponding test embedding to inspect potential relationships between them. Figure 2 shows the arrow plots of some sampled pairings (in order to more easily see the results) using all different models.

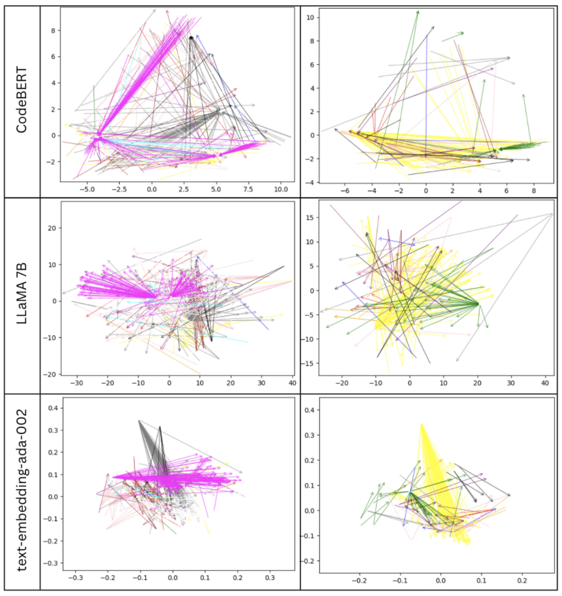


Figure 2: Arrow plots using PCA for 2-D projection of paired focal method embedding and test embedding, sampled across different models. Arrows of the same color represent pairings from the same GitHub project.

We observed that, in general, CodeBERT embeddings showed no clear pattern, with many of the arrows seemingly pointing in the same location. This suggests that CodeBERT assigned the same embeddings to different tests. This difficulty in producing unique and meaningful representations for test cases may be explained by the fact that when training the CodeBERT model, **"function names with substring 'test' are removed"** (Feng et al., 2020). On the other hand, pairs of embeddings generated by LLaMA-1 and text-embedding-ada-002 appeared to display non-arbitrary directions, suggesting that there may be informative patterns to capture that merit further exploration.

Model	$K = 1$	$K = 5$	$K = 10$
CodeBERT	0.35%	0.54%	0.68%
LLaMA-1 7B	7.31%	14.41%	18.38%
ada-002	**31.78%**	**54.45%**	**62.53%**

Table 1: Proximity-based test embedding retrieval results (top-K accuracy). Best results in **bold**.

To confirm our visual findings, we ran a permutation test with the text-embedding-ada-002 embeddings. The test statistic used in our case was the mean cosine similarity between corresponding vectors in the set of tests and the set of focal methods, and the number of permutations was set to 10,000. In each permutation, each test was assigned a random focal method to be paired with, and the mean cosine similarity was computed between all pairs. Our results showed that: $p\_value \approx 0.0$, indicating that the mean cosine similarity between the actual pairs was extremely unlikely to have occurred by chance, and there is some significant relationship between the pairs. Therefore, it may be possible to develop a retrieval model that leverages this relationship in order to find relevant test cases given an input focal method.

4.2 Retrieval Tasks and Neural Network Exploration

4.2.1 Retrieval Tasks: Proximity as a Hypothesis

Based on the results of our permutation test, we next sought to explore whether the closest test embedding to a focal method embedding was its corresponding test embedding. To test this, we used KNN with cosine as a distance metric, to find the closest K tests embeddings to each focal method embedding and see if one of them is indeed its corresponding test embedding. We then performed a comparative analysis using top-K accuracy for $K \in \{1, 5, 10\}$. Our investigation included the evaluation of the performance of CodeBERT, LLaMA-1 7B, and Text-embedding-ada-002 models. The results are shown in the table 1.

Results indicated that the OpenAI Text-embedding-ada-002 model stood out with the high-

[2] https://openai.com/blog/new-and-improved-embedding-model

Model	$K = 1$	$K = 5$	$K = 10$
CodeBERT	6.26%	7.16%	8.09%
LLaMA-1 7B	15.91%	36.65%	46.25%
Text-embedding-ada-002	**20.99%**	**47.9%**	**57.88%**

Table 2: Results of our NN with the different embedding models using top-K accuracy. Best results in **bold**.

est accuracy for each value of K, showcasing its ability to capture effectively the code semantics. In contrast, LLaMA-1's performance was comparatively weaker, while CodeBERT yielded the lowest accuracy.

4.2.2 Neural Network Exploration

The results we obtained motivated us to explore more and see if we could train an NN to approximate the test embeddings given the focal method embeddings. We constructed an NN using TensorFlow's Keras [3] API. We used a sequential NN architecture with five fully connected layers and ReLU activation functions. We used 80% of the data for training, while the remaining 20% was used for testing, and Mean Squared Error (MSE) Loss was used. To evaluate the performance of the NN, we used KNN with cosine metric to find the N closest tests embeddings to the predicted vector given the focal method embedding. We then checked if the corresponding focal method embedding of the test embedding is among those K nearest neighbors and calculated the top-K accuracy scores, and the results are presented in Table 2.

Comparing the two tables 2 and 1, we observed that the NN-based approach had lower accuracy scores than the proximity-based approach for the text-embedding-ada-002 model. However, for the LLaMA-1 7B and CodeBERT models, the accuracy scores improved with the NN-based approach. Despite the accuracy improvements for CodeBERT and LLaMA-1 7B with the NN-based approach, all three models maintained the same ranking based on their accuracy rates.

5 Test Cases Generation Task

5.1 Assessing GPT-3.5-Generated Test Cases in Comparison with Human-Generated Tests

Given the popularity of LLMs for code generation, especially GPT-3.5, we chose to investigate how well these types of models, can generate test cases for ML code. We generated cases for all of our ML functions by invoking GPT-3.5 with the prompt: *"Generate the test function in Python for this code: <focal_method_definition> Give me the code only, with no explanation, but keep the comments."* We chose a simple prompt because we wanted to avoid biasing GPT-3.5 too much (Shapira et al., 2023). However, testing out multiple prompts is a promising direction for future work. We maintained a temperature value of 1 during generation, so for the same function, GPT-3.5 generated different test cases covering different aspects.

We intentionally did not include any information about the project from which we retrieved the ML function in the prompt. Consequently, GPT-3.5 may or may not have seen the project before, as it likely was trained on GitHub projects dating before **September 2021**. Nevertheless, we did confirm that GPT-3.5 was not exactly reproducing the human-written test cases. To ensure consistency across our dataset, we performed preprocessing to retain only the tests' definitions, excluding any explanations that came before or after it, just like we did with the human-generated tests.

Initial analysis measuring the average lines of code and comments in the test functions, as reported in table 3, unveiled that GPT-3.5 tends to create longer (in terms of number of lines) test cases with fewer comments than humans. Additionally, both GPT-3.5 and humans occasionally omitted the function call within their test cases. Notably, 4.6% of GPT-3.5 tests and 3.28% of human tests lacked the call for the focal method. This can be explained by the diverse scenarios of unanticipated GPT-3.5 test case generation outcomes such as when the test case consisted of a `pass` statement only, when the generated code was not a test function, or when GPT-3.5 replicated the code of the focal method when tasked with generating a test case.

5.2 Embeddings Comparison and Statistical Analysis

For further investigation, we used the model text-embedding-ada-002, since it performed the best with our retrieval task, to generate embeddings for the GPT-3.5-generated test cases as well.

Using **PCA** dimensionality reduction technique, we performed visualization to detect if there are some differences between human-generated test embeddings and GPT-3.5-generated test embeddings that are potentially visible. We created scatter plots of the reduced embeddings, as shown in Figure 3, that showcased a general overlap of the two

[3]https://keras.io/

	Average Lines of Code (Including Comments)	**Average Lines of Comments**	**Comment Percentage**	**Percentage of Test Cases Calling the Tested Function**
Human	10.24	0.15	1.46%	96,72%
GPT-3.5	18.57	0.08	0.43%	95.35%

Table 3: Comparison of test cases characteristics: Human vs. GPT-3.5 generated tests

test groups suggesting that there was not a large overall difference between them. The same figure 3 revealed two noticeable clusters in the human-generated tests. Upon examination, we found that the second cluster of human-generated tests consistently included the presence of `@pytest.fixture` decorators before the tests. This condition is sufficient but not necessary to indicate the use of the Pytest framework. Conversely, GPT-3.5 did not use these fixtures as much. Therefore, the clusters in human tests may be attributed to the presence or absence of these fixtures or the choice of different testing frameworks in general. Either way, this clarification highlights the need for a more in-depth investigation.

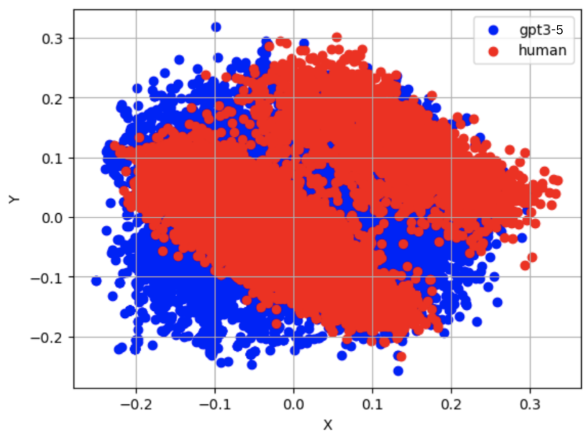


Figure 3: Scatter Plot of Reduced Embeddings using PCA.

To quantitatively confirm our findings, we ran a **t-test**, to determine if there is a significant difference between the means of the embeddings of tests generated by Humans and the tests generated by GPT-3.5. The computed t-statistic values were very close to zero, indicating a minimal variance in means between the Human and GPT-3.5 test embeddings. Consequently, the p-values were nearly 1, far exceeding our significance level of $\alpha = 0.05$. Consequently, we fail to reject the null hypothesis (There is no difference between the means of our two samples). The outcomes of our t-test suggest that statistically speaking, the means of the Human and GPT-3 test embeddings do not display a significant statistical difference. This outcome does not imply that they are identical (as there may be divergences in other parameters like standard deviation, minimum, maximum, etc.). However, it does signify that, from a statistical perspective, we lack evidence to affirm their difference. With that being said, GPT-3.5 tests seem to be very similar to human tests, according to what can be measured using embeddings, which might not represent every facet of the tests. As visualization did not help much capture the differences between both test groups, we conducted a survey to understand which test cases developers and data scientists found more helpful for ML test case generation.

6 Survey Methodology and Results Analysis

6.1 Survey Methodology

6.1.1 Survey Setup

We created four different variations of the survey with the possibility for one person to respond to more than one. Each variation of the survey had 5 ML functions extracted from 5 different GitHub projects, each with 5 accompanying test cases. So overall, there were 20 different ML functions from 20 different GitHub Projects and a total of 100 test cases.

Upon the emergence of newer LLMs such as GPT-4 and LLaMA-2, and recognizing their potential in test case generation for ML code, we aimed to explore their capabilities as well. To manage costs associated with API calls, we opted not to generate test cases for *all* of our ML functions using GPT-4. Due to the smaller sample size required for the survey, we managed to use both GPT-4 and LLaMA-2 (with 70 billion parameters) in order to compare these other large models with GPT-3.5. The 5 accompanying test cases for each ML function were the human-generated test for that function, the GPT-3.5-generated test, the retrieved test, the LLaMA-2-generated test (70B), and the GPT-4-generated test.

Both GPT-4 and LLaMA-2 (70B) tests were generated by invoking the same prompt used to generate tests using GPT-3.5. To provide the retrieved test, we followed the method that we described in section 4.2, only this time, when seeking the closest test embedding to the focal method embedding from all human-generated test cases, we purposely *excluded the test cases originating from the same project as the focal method embedding*. By doing so, we simulated an environment wherein our system had not encountered the project before.

The process of selecting the ML functions used in the survey involved a random selection from functions that had a comment section that clarified the function's objective so that it was easier for survey takers to understand the code. Furthermore, we made sure that we were certain that the associated human test was correctly paired, eliminating cases that could be considered as noise.

Moreover, participants were not provided with links to the associated GitHub projects. This decision was made to ensure fairness, as both the participants and AI assistants may or may not have had prior exposure to these projects. However, since all functions had comments, participants were able to read about the intended purpose of the function.

6.1.2 Survey Structure and Instructions

Our survey starts with inquiries about participants' backgrounds, asking for their experience in ML and software testing, prior usage of AI tools for generating test cases, and more. Afterward, participants were presented with a hypothetical scenario wherein they were tasked with writing a test case for an ML function, and five distinct AI assistants provided example test cases to help them write it. Participants were then requested to evaluate each option based on helpfulness, correctness, and readability. The test cases were labeled as test_A, test_B, test_C, test_D, and test_E. For instance, test_A represented the test generated by humans, while test_B, test_C, test_D, and test_E corresponded to GPT-3.5, retrieved, LLaMA-2 (70B), and GPT-4 generated tests, respectively. Participants did not know the true identity of any of the systems. To eliminate any potential biases, we applied shuffling of system labels across the various survey versions. At the survey's conclusion, participants were asked to indicate their preferred system.

6.1.3 Survey Participant Groups

Our survey enlisted participants from diverse groups including researchers, students, ML engineers, and software developers. To prevent any potential bias, individuals within the same group responded to distinct survey variations. This approach ensured that each survey variant collected responses from a range of groups, avoiding biased results. The participants completed the survey on a voluntary basis and were recruited from the social networks and university groups of the authors' universities in both the United States and North Africa.

6.2 Results Analysis

6.2.1 Distribution of Participants

Our survey was completed by 17 participants from diverse backgrounds. With each survey containing 5 test cases, a cumulative **425** evaluations of test cases was reported. The results revealed that the largest group of participants was students at 41.2%, followed by researchers and software developers at 23.5%, and ML engineers who constituted 11.2% of the participants. Over 64% of our participants had at least 1 year of experience in ML, and over 47% of them had at least 1 year of experience in Software Testing. This overall experience makes them adequate for the evaluation of ML test cases. Surprisingly, the majority of the participants have never used an AI tool to generate test cases before. The few who did mentioned that they have used ChatGPT or Testsigma[4]. The features of a good machine learning test case, as mentioned by participants, are presented in Figure 4 along with the corresponding number of mentions by participants.

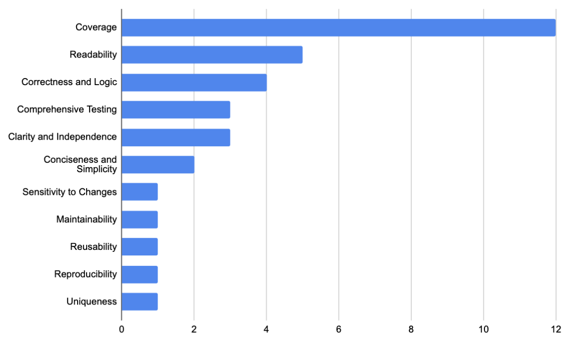


Figure 4: Key Features of a Good Machine Learning Test Case: Participants' Answers.

6.2.2 Survey Findings: Participant Evaluation of Different Test Cases

Throughout our survey, we asked participants to evaluate each test case individually on a scale of 1

[4]https://testsigma.com/

	Human	**Retrieval**	**GPT-3.5**	**LLaMA-2**	**GPT-4**
Correctness Score	3.62	2.86	3.17	3.87	**3.93**
Readability Score	3.25	2.88	3.09	**4.17**	4.12
Documentation: Average Lines of Comments	0	0	0	4.6	**5.1**
Helpfulness Score	1.84	1.44	1.82	2.39	**2.6**
System Preference Distribution	11.8%	0%	11.8%	35.3%	**41.2%**
Rank Chosen by people	3	4	3	2	**1**

Table 4: Survey Results: Evaluation Scores and Rankings for Different AI Assistants. The best results are in bold. For correctness, readability, documentation, helpfulness, and preference distribution scores, the highest is best. For the rank chosen, the lowest is the best.

to 5, considering two criteria: **Correctness**[5] (taking into consideration the testing logic) and **Readability**. We opted for these two criteria because they are crucial for assessing test cases, and they can be assessed by simply reviewing the test case and relying on participants' expertise without having to run the code, which would have been difficult in an online survey setting; we leave verifying the ability to execute the code as future work.

Additionally, we asked our participants to imagine that they needed to write a test case for the target function, and then to rank each 5 test cases associated with the same ML project based on their **helpfulness** as a reference or starting point for writing a test case for the provided ML function.

The averages of participants' scores for each criterion were calculated and summarized in Table 4. We used the Mean Reciprocal Rank (MRR) metric to calculate the helpfulness score using the different ranks associated by people for this criterion.

$$\text{MRR} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\text{rank}_i}$$

where n is the number of the ranked elements and rank_i is the rank assigned for the element i.

Despite having some criteria that led to strong correlations, the reported results reveal that GPT-4 achieved the highest scores in Correctness, Documentation, and Helpfulness. On the other hand, LLaMA-2 (70B) [6] had the highest score in Readability. These two models exhibited similar scores, leading to a tight competition for the top-ranking position. However, LLaMA-2 is an open[7] model, the fact that it performs almost as well as GPT-4 in this task may have a larger positive impact overall since anyone can benefit from it.

Also with very closely matched scores, we find human-generated tests and GPT-3.5-generated tests. Even though human-generated tests slightly outperformed the GPT-3.5 model in terms of Correctness, Readability, and Helpfulness ratings, their scores are still very close. This might confirm the idea first presented in Section 5.2: GPT-3.5 and human tests are similar, with a small but noticeable difference (as suggested by their different scores) that is not captured by embedding similarity.

At last, retrieved tests attained the lowest scores, resulting in a fifth-place ranking. This suggests that participants found all generative models to appear more helpful than the actual test functions that had been written to test similar ML functions.

6.2.3 Participant Insights: System Preference

As a final question in our survey, we inquired about participants' preferred system overall. Our results revealed that the majority of our participants at 41,2% preferred GPT-4-generated tests, followed by 35,3% opting for LLaMA-2-generated tests, while the rest split up between human-generated and GPT-3.5-generated tests, with no preference for retrieved tests.

Individuals with over one year of experience in ML and software testing preferred tests generated by humans and LLaMA-2 (70B) more often than others. This suggests that there may be something lacking in tests generated by GPT-4, which is only apparent to those with more experience. While this trend is interesting, it should be taken with caution due to the limited sample size. To confirm this pattern, additional data is required, making it a potential area for future work.

In summary, the GPT-4 and LLaMA-2 (70B) models excel in generating apparently correct, readable, and helpful tests. Given that a majority of participants indicated that they haven't used AI tools for test generation previously, this suggests they

[5]Note that "correctness" in this case measures *perceived* correctness based on human observation.

[6]Recently, Code LLaMA was introduced, but it was after the conclusion of our survey. Future work could explore this and determine if it (or other newer models) would be preferred even above GPT-4 and LLaMA-2.

[7]Although the pre-training code and data are not fully open, the parameters of the model are available via a license that is fairly unrestrictive for research purposes.

might benefit from using them for such tasks.

7 Conclusion

In this work, we employed state-of-the-art NLP techniques to generate effective representations for ML source and test code. We developed a heuristic method to build a good-quality dataset of ML function-to-test mappings, forming the basis for generating these representations. We have studied these representations through visualization by leveraging a couple of dimensionality reduction methods, and we have successfully captured some patterns, that we later confirmed. Our findings revealed an interesting insight: the CodeBERT model struggled to capture test case semantics compared to other recent GPT embeddings. We also explored the practicality of these representations for retrieving an ML test case given an ML method. Surprisingly, even state-of-the-art NLP models faced challenges in this task. We also assessed the performance of LLMs in automatically generating test cases, which revealed that some of these models outperformed human-generated tests in terms of helpfulness.

8 Limitations

It's important to acknowledge the potential weaknesses in our original dataset. Firstly, it is important to acknowledge that the quality of the collected tests may vary, as not all developers write equally comprehensive or effective tests. This variability in test quality introduces a degree of uncertainty in the dataset. Additionally, the dataset consists of projects of varying sizes. As a result, some projects are larger than others, providing a bigger pool of tests for extraction. This discrepancy in project sizes could potentially impact the representation and diversity of the dataset. Furthermore, it is worth noting that a subset of tests in the dataset may be minimal, such as those with the content **def test(): pass**. These minimal tests lack substantial functionality and may not contribute significantly to the overall depth of the dataset.

It is also essential to acknowledge the limitations inherent in our dataset's size, which does not cover a variety of languages and was selected to increase the precision of paired functions and tests rather than to maximize coverage. Lastly, it is important to acknowledge that while the dataset primarily focuses on ML tests, it is challenging to definitively determine if all tests exclusively pertain to ML functionalities rather than general software testing. Due to the inherent complexity and interplay between ML and software testing, there may be instances where tests encompass aspects beyond pure ML functionalities.

Also, for our retrieval task, and while the proximity-based approach yielded promising results, the NN-based approach might still have room for improvement potentially through refining the neural network architecture or optimization techniques. Further, a retrieval augmented generation (RAG) approach might be useful in order to gain the benefits from both the retrieval and generation-based approaches.

Recognizing the limitations inherent in our survey findings is also important. To begin, participants didn't have the opportunity to execute the provided code within the survey and didn't have access to the whole repository, compelling them to rely on their intuition and expertise only for evaluating the various systems.

Moreover, it is crucial to acknowledge that the survey exclusively measures the perceived correctness of the tests. Actual execution of the tests to determine their functional accuracy could provide a more robust evaluation.

Additionally, while the survey's participant count is relatively modest, it remains representative. However, it's worth noting that outcomes might exhibit variation with a larger sample size. Despite those limitations, the results remain interesting and undeniably pave the way for future research perspectives.

9 Ethical Considerations

Using LLMs to generate ML test cases presents some ethical concerns that demand careful consideration. Firstly, there is the risk of unintentional leakage of sensitive information from the training data into the generated test cases, potentially compromising privacy and confidentiality. Moreover, the lack of transparency in LLMs makes it challenging to understand how these test cases are formulated, raising concerns about accountability and the potential for bias amplification. Over-reliance on the automation capabilities of LLMs in the testing process may lead to the displacement of human testers, impacting job security and employment opportunities. Additionally, there is a risk of intellectual property violation when generated test cases closely resemble proprietary data or test scenarios. Another concern involves the potential for erroneous test cases. LLM-generated tests may contain inaccuracies, ambiguities, or flaws that, if not rigorously reviewed and validated, could lead to unreliable ML models that fail to perform as expected. We urge ML test case developers to use LLMs with caution and scrutiny, even though the generated tests appear to be promising. Verifying the generated tests remains an important step in the software development process.

References

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25:5193–5254.

Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. 2018. Spiritus: A simple information retrieval regression test selection approach. *Information and Software Technology*, 99:62–80.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*.

Natalie Shapira, Mosh Levy, Seyed Hossein Alavi, Xuhui Zhou, Yejin Choi, Yoav Goldberg, Maarten Sap, and Vered Shwartz. 2023. Clever hans or neural theory of mind? stress testing social reasoning in large language models. *arXiv preprint arXiv:2305.14763*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.