



DOMAIN ADAPTATION FOR DEEP UNIT TEST CASE GENERATION

PRESENTED BY

Yazeed Adnan Sahawneh
Mayar Sameer Thiab

Ahmad Ibrahim Athamneh
Gassasn AlNassar

-
- 01 Introduction
 - 02 Softwar
 - 03 Black Box
 - 04 White Box
 - 05 Software Testing
 - 06 Domain Adaptation
 - 07 AI in Software Testing
 - 08 Text case generation
 - 09 methods that we use
 - NLP
 - Code T5
 - Fine-Tuning
 - 10 Discussion
 - 11 conclusion
 - 12 Results
 - 13 References
 - 14 code

Introduction

This project deals with the automation of unit test cases through artificial intelligence, addressing the difficulties and challenges of conventional methods in software testing. The key goals here include better coverage of tests, less manual effort, more accuracy, and scalability. It will aim to achieve higher efficiency and accuracy in software testing by using a Transformer-based code model, CodeT5, fine-tuned under this work. Ultimately, this project contributes to the development of higher-quality software.

Software Development

In the planning phase, meticulous planning and execution were done for various critical phases to prepare software development for AI-driven automated unit test case generation. Initially, the project objectives and definition of the scope and deliverables were comprehensively understood and then followed the stakeholder analysis stage, where the input and requirements from the stakeholders were gathered to be responsive to their needs.

In the design and architecture phase, a huge emphasis was put on developing a robust and modular framework whereby a scalable and modular system architecture was designed, as shown below: data collection and preprocessing, model training and fine-tuning, generation of test cases, and finally, evaluation and validation. We took great care to design every single component so that it carries out its function appropriately: from the collection of dirtied-up, real-world data to cleaning, and then automatically generating and evaluating test cases.

The project adopted the technology stack: Python for coding, PyTorch or TensorFlow for model implementation, the Transformers library for using pre-trained models, and Scikit-learn for model preprocessing and evaluation metrics. Experimentation was mainly done using tools like Jupyter Notebooks, version control was achieved through Git and environment consistency with Docker. With careful planning, detailed design, and the assimilation of AI techniques, this project aimed to effectively automate the generation of unit test cases to contribute to enhanced software testing practices.

Black box and White box

Black box testing treats the software as a closed box; testing is conducted to evaluate the product's functionality without knowledge of the internal structure and/or implementation of the code. The software is tested only through its external interfaces, inputs, and outputs. This approach tries to simulate the user's actions, checking the response of the software against the requirement specified. In simpler terms, about the functional aspects, test cases are designed by the testers based on the inputs that are expected and the corresponding outputs of the system, i.e., how the user will interact with the system, the response of the system, and error handling. Software validation from an end-user perspective concerning usability, functional features, and performance is well tackled using black box testing.

White box testing, also known as structural testing or glass box testing, is a kind of software testing where the internal working of software is being examined. The source code, design documents, and architectural diagrams of the software will be accessible to the testers so that the primary rationale, algorithms, and data structures on which the software functions will be known. White-box testing targets the correctness of the internal implementation of the code, which includes code coverage, logic errors, and programming flaws. The testers design the test cases out of their understanding of the internal structure of the code, which highlights the specific code paths, branches, and conditions to be covered. This methodology is helpful in finding mistakes that cannot be identified only by the external test, like boundary cases, edge conditions, and integration problems.

Software testing

In this chapter, we enter the vast arena of software testing—its various types, the quality factor emphasized, and the methodologies used to make software robust and reliable. There are five basic software testing strategies:

- unit testing,
- integration testing,
- system testing,
- acceptance testing,
- regression testing.

Each of these types plays its dedicated role: from inspection of the functionality of particular units and components to the whole scenario with regard to the business needs and users' needs their software solution would serve. Quality in software testing is fundamental: it may relate to the reliability or performance processes, as well as the satisfaction of users. Key areas to ensure high quality in testing are test coverage, automated testing, test case quality, and performance metrics. Methodologies used are Test-Driven Development (TDD), Behavior-Driven Development (BDD), Continuous Integration / Continuous Deployment (CI/CD), and AI-Driven Testing. The application of AI techniques in testing processes offers various advantages, including test coverage, reduced human effort, enhancement in test quality and scalability, and encouragement to perform software testing practices in a cost- and time-effective way.

Domain Adaptation

It is a machine-learning, artificial-intelligence-based technique developed to transfer knowledge obtained from one domain to another. In the software testing context, domain adaptation can be applied to improve automated software testing with different software projects or environments.

The importance of domain adaptation in software testing arises because most software projects or the environment associated with them will have their characteristics for coding styles, architectural designs, and usage scenarios. When developing automated testing systems or models, it's crucial to adapt to these variations effectively so that their accuracy and reliability can be maintained.

In software testing, the domain adaptation process might entail, for example, modifying or fine-tuning the existing testing models or algorithms to better fit the peculiar characteristics of the target domain. This could be in terms of retraining machine learning models on data from a new domain, adjusting parameters/hyperparameters, or inclusion of domain-specific knowledge in the testing process.

Domain adaptation techniques make testing software systems more resilient against the variations between projects or between project environments, which in turn leads to more accurate test results and better overall test efficiency.

This contributes to higher quality and more reliable software development scenarios

How we use of ai in software testing

AI can change software test automation practices by enhancing precision and increasing the reliability of routine operations in testing. AI in testing will lead to the automation of tedious operations, more efficiency level increases, and a decreased chance of human errors. Additionally, AI-driven tools will be able to process data more accurately, detect defects at an early stage of development, and maintain consistency in meeting the testing standards.

Moreover, AI enables the acceleration of the overall development process, supporting rapid cycles of testing while attaining the practices of continuous integration and deployment. Three primary applications of AI in testing include the automatic generation of test cases, helping the processes of automatically creating detailed test suites from method descriptions or requirements, leaving no room for gaps and saving much manual work.

AI-based test optimization helps to find test cases that are redundant or less effective, focusing more on critical areas of application and optimizing resource allocation toward them. Finally, predictive analytics with historical test data foresees potential outcomes that contribute to proactive decision-making and resource allocation in driving better overall software quality.

Test case generation

Test case generation is the most crucial activity of software testing; it describes the process of generating test cases to verify the functionality, performance, and reliability of a software application. Test cases are for different scenarios, inputs, and conditions, which check the software against its specified requirements and ensure the intended behavior of the software. The paper then delves into the test case generation process, techniques, and considerations.

Test case generation can be taken on several approaches based on software complexity, requirements, and available resources. One of the most common approaches to test case generation is manual test case generation.

In general, testers develop the test cases manually from a general understanding of the software and requirements. This gives them control and flexibility but may be time-consuming and not guarantee the coverage of all possible scenarios

Another approach is the generation of test cases automatically, where tools, scripts, or algorithms are used in the generation process. This approach helps in those large or complex software systems where manual test case creation is not possible. Automated test case generation involves various techniques, which include the following:

Random testing: Generating test cases randomly to explore different scenarios and inputs.

Model-based testing: Test cases can be systematically drawn from the models or specifications of software under test.

Mutation testing: Minor modifications, or mutations, are made in existing test cases, and their fallibility to fault detection is tested.

Code-based testing: Tests derived from the analysis of the source code of the software can be selected.

Details of the software and a good understanding of the software requirements, functionalities, and possible test case generation are the key points for successful test case generation. Other things a tester needs to keep in mind are boundary conditions, handling errors, and integration points to achieve completeness in test coverage. Test cases should be clear, concise, and well-documented for efficient execution and easy maintenance.

methods that we use

- **Natural Language Processing (NLP)**

The method descriptions were then processed to extract the corresponding test cases with the help of Natural Language Processing techniques. NLP algorithms encode the text description of the software functions in addition to furthering the extraction of the pertinent information, which guides test case generation. On the other hand, through semantics and context in the descriptions, it generated test cases accurately and relevantly to the specific functionalities of the software in question.

- **Code T5 Model**

The former is based on the Transformer architecture and, hence, the latter, with an exact instrument trained over appropriate large corpora of code data. Thus, the Code T5 model can generate such syntactically and semantically meaningful test cases from method descriptions based on the deep understanding of programming languages.

This made it possible to comprehend and even produce code snippets, which is an essential tool in test case generation automation.

- **Fine-Tuning**

The techniques in fine-tuning the Code T5 model come in as the model is fitted to the needs and specifics of individual software projects. Model fine-tuning is performed on project-specific data to improve the model in both its performance and accuracy in generating relevant test cases. It is the process of tuning hyperparameters, such as learning rates and batch sizes, and iterations of the output to optimize the performance for a given context of the project. Fine-tuning could ensure that the generated test cases were parallel with the project requirements and could adequately validate the functionalities thereof.

Discussion

The incorporation of NLP techniques and the Code T5 model into our methodology for the generation of test cases is in alignment with the progressive step for testing practices. Our methodologies of NLP, which process descriptions of methods for information extraction, helped us eliminate the disparity in the information made available through natural language specifications and implemented in the code. Therefore, this information forms a basis for the automatic generation of the test case.

The T5 model, specialized and retrained on data involving code, should give the ability to generate test cases that are both syntactically correct and semantically correct for thorough coverage and accuracy in testing. Fine tuning techniques have further enhanced the performance of Code T5, allowing it to adapt to the specific requirements and nuisances of individual software projects. High accuracy in the generated test cases was achieved by iterating hyperparameters of models and training the model on data related to the purpose.

This feature made the models adaptive and customizable to bear the semblance to the project functionalities for good checking of the performances and reliability.

Conclusion

In conclusion, the process of test case generation in software testing advanced a considerable step by applying NLP techniques, the Code T5 model, and methodologies for fine-tuning. The AI-driven methodologies have made predominant changes in the testing process—reducing testing processes, reducing repetition and automation for better accuracy, and speeding up the test life cycle. We use NLP for parsing the natural language specifications and the T5 model of Code to generate the code snippets to help achieve efficient and thorough testing, contributing to the software quality and reliability improvement.

Looking into future horizons, continuous research in AI and machine learning holds much promise in advancing practices in software testing. With more sophisticated AI models, much more can be expected in automation and efficiency in generating the test cases, which eventually lead to a faster pace of development and a higher quality of software products. Consequently, mastering this technology and the respective methodology is critical to remaining competitive in a landscape where dynamism and demand in software development rise at an ever-increasing rate

Results

Q1: Can Transformer-based code models generate high-quality (compilable with high code coverage) test cases?

Answer : The results from CodeT5 without DA, show that post-processing can improve the compilability of generated test cases, but their code coverage is still low

Q2: How effective is project-level domain adaptation in terms of improving code coverage of automatically generated test cases using CodeT5?

Answer : Overall, the results suggest that applying project-specific domain adaptation in a smaller language model has 18.62, 20.38, and 16.34% improvements over the one without DA, 18.02, 15.93, 8.72% improvement over the state-of-the-art LLM (GPT-4), and 19.88, 23.40, 18.66% improvement over specialized models for test case generation that exploits task-level DA (A3Test) on line coverage, BLEU, and CodeBLEU, respectively

Q3: Can our proposed test generation approach augment search-based test case generation?

Answer: In general, our approach can increase the coverage and mutation score of the existing state-of-the-art test generation techniques such as EvoSuite and thus is recommended to be used together with such tools.

Table 1: Evaluation metrics scores for CodeT5 without Domain Adaptation (DA) (RQ1). Comparison of Code-T5 with (DA) versus GPT-4 and A3Test is also shown for RQ2. Cov column stands for line coverage, BL column stands for BLEU and CB stands for CodeBLEU score. Bold values denote the best metric score in the row, i.e. comparing each project's w.r.t baselines

Projects	CodeT5 without DA			CodeT5 with DA			GPT-4			A3Test		
	Cov	BL	CB	Cov	BL	CB	Cov	BL	CB	Cov	BL	CB
compress	2.00	11.59	9.15	32.80	40.84	22.37	0.70	18.53	18.73	2.00	11.33	7.42
gson	25.60	18.64	16.64	52.20	42.06	35.12	32.40	26.39	28.19	29.50	16.44	15.61
jksnCore	2.10	16.39	16.98	21.20	28.41	30.06	4.10	18.29	23.32	2.00	13.08	13.32
jksnDB	31.40	18.34	16.78	43.10	36.74	31.70	33.20	22.43	23.87	31.60	15.75	15.58
jsoup	63.10	25.56	22.10	68.00	44.36	44.10	56.80	27.11	25.65	52.80	18.79	18.11
AVG	24.84	18.10	16.33	43.46	38.48	32.67	25.44	22.55	23.95	23.58	15.08	14.01

Table 2: Line Coverage comparison between EvoSuite and model generated tests. The NewCL has covered lines that neither EvoSuite nor the developer-written tests from the training set have covered.


Project	Model CL		EvoSuite CL		New CL		Total Lines
compress	216	58%	87	23%	174	46.70%	372
gson	458	69%	539	82%	31	4.70%	657
jksnCore	399	30%	674	51%	82	6.20%	1307
jksnDB	1357	50%	136	5%	1246	48%	2595
jsoup	192	82%	39	16%	157	66.50%	519
AVG	524.4	58%	295	35%	338	34.42%	1090

Table 3: Mutation and adapted mutation scores for model generated (Model MS and Model AMS) and EvoSuite (Evo MS and Evo AMS) tests. The New MK column shows the number of mutants that were not killed by EvoSuite but were killed by model-generated tests.

Project	Model MS	Evo MS	Model AMS	Evo AMS	New MK	
compress	0.00%	55.90%	0.00%	69.50%	0	0%
gson	13.50%	64.90%	50.00%	100.00%	0	0%
jksnCore	14.80%	87.20%	50.70%	100.00%	0	0%
jksnDB	22.40%	0.00%	54.20%	0.00%	26	22.40%
jsoup	32.00%	0.00%	47.10%	0.00%	8	32%
AVG	16.54%	41.60%	40.40%	53.90%	6.8	11%

References

Code

<pre> 1 [LINE] return this.labelInsets; [LINE] 2 public RectangleInsets getLabelInsets() { [EOL] 3 return this.labelInsets; [EOL] 4 } 5 protected Axis(String label); 6 public boolean isVisible(); 7 public String getLabel(); 8 public void setLabel(String label); 9 public Font getLabelFont(); 10 public void setLabelFont(Font font); 11 public Paint getLabelPaint(); 12 public void setLabelPaint(Paint paint); 13 public RectangleInsets getLabelInsets(); 14 public void setLabelInsets(RectangleInsets insets); 15 public void setLabelInsets(RectangleInsets insets, boolean notify); 16 public String getLabelURL(); 17 boolean DEFAULT_TICK_MARKS_VISIBLE=Optional[true]; 18 Stroke DEFAULT_TICK_MARK_STROKE=Optional[new BasicStroke(1)]; 19 Paint DEFAULT_TICK_MARK_PAINT=Optional[Color.gray]; 20 float DEFAULT_TICK_MARK_OUTSIDE_LENGTH=Optional[2.0f]; 21 boolean visible; 22 String label; 23 Font labelFont; 24 RectangleInsets labelInsets; 25 double labelAngle; 26 String labelToolTip; 27 String labelURL; 28 float minorTickMarkOutsideLength; 29 Paint tickMarkPaint; 30 double fixedDimension; </pre>	 <div style="display: flex; flex-direction: column; align-items: flex-end;"> <div style="margin-bottom: 10px;">Target line</div> <div style="margin-bottom: 10px;">Focal method (FM)</div> <div style="margin-bottom: 10px;">Focal class constructor (C)</div> <div style="margin-bottom: 10px;">Signatures of the other methods in the focal class (M)</div> <div>Public fields of the focal class (F)</div> </div>
--	--