International Journal of Core Engineering & Management
Volume-5, Issue-12, March-2019, ISSN No: 2348-9510

NATURAL LANGUAGE PROCESSING FOR AUTOMATED TEST CASE GENERATION FROM SOFTWARE REQUIREMENT DOCUMENTS

Hariprasad Sivaraman
shiv.hariprasad@gmail.com

Abstract

This paper presents an innovative framework utilizing Natural Language Processing (NLP) for automating test case generation from requirement documents, aiming to address common inefficiencies in manual test case creation. Our approach leverages advanced NLP techniques—such as tokenization, Named Entity Recognition (NER), dependency parsing, and transformer-based contextual embedding's—to interpret requirements accurately and generate standardized test cases. Experimental results from a prototype developed using SpaCy and a fine-tuned BERT model indicate a significant reduction in time and improved consistency in test case creation. This framework offers a scalable solution for enhancing the efficiency and precision of software testing.

Keywords: Natural Language Processing, Automated Test Case Generation, Requirement Analysis, Software Testing Automation, NLP in Software Engineering, Machine Learning

I. INTRODUCTION

Software Testing is a vital phase in delivering reliable and high-quality software. Traditional testing involves manually creating test cases from requirement documents, which is time-consuming process and also causes irregularities. With the rise of agile and continuous integration, there is a strained demand on testing cycles requiring fast, automated testing.

NLP approach for Automated Test Case Generation utilizes Natural Language Understanding (NLU) to convert unstructured requirement text into unstructured natural language-based test case. NLP, aided by the latest deep learning advances, allows systems to understand patterns in language, handle ambiguities, and provide consistent, scalable requirement interpretations. This paper presents a NLP framework to parse requirements, understand contexts, and create structured and executable tests. This not only speeds up testing and limits human error but also provides a more scalable and flexible solution.

II. PROBLEM STATEMENT

Manual generation of test cases from requirements have the following issues:

- **Ambiguity:** Due to the inherent ambiguity in the requirements, it leads to inconsistencies where the same test case may not be applicable across different test suites.
- **Scalability Problems**: Manual processes do not scale easily, which ultimately slows down the delivery and increases the chances of human errors in case of highly complex builds or

deployments.

This pain-point can be alleviated by automating this requirement into a structured test case through the use of NLP that enables the entire process to be more productive while maintaining uniformity in test case interpretation as the software development lifecycle continues.

III. NLP-BASED SOLUTION FRAMEWORK

A. Requirements Analysis and NLP Preprocessing

The first step is to start with pre-parsing the requirements. The unstructured requirement text is converted into a structured format that is suitable for analysis. It is important to retain the context, the inter relationships between the various functionalities, and the subsequent effect of the steps on the test case generation process.

Tokenization: Tokenization is the process of splitting text into words, phrases or sentences to facilitate analysis. Here, every requirement document sentence is broken into multiple tokens or pieces, such as Pathological Tokenization with compound words and multi-words (meaningful together).

- **Example** — In "The user should be able to log in and see there Dashboard", tokenization will convert it to ["The", "user", "should", "be", "able", "to", "log", "in", "and", "see", "there", "Dashboard"]. In advanced tokenizers, particular attention is paid to ensure that multi-word or multi-character entities are kept together — for example if the user has searched for a phrase — 'log in', it is preferable that the individual words —'log', 'in' — not be separated before the next step of analysis.
- **Tools:** Serialization or tokenization functions can be found in tools such as SpaCy and NLTK, providing basic and custom functionality to order how tokens are formed by supporting regularity and exceptional cases like punctuation and special characters in requirements.

B. Part-of-Speech (POS) Tagging— POS tagging is done on each token to get the different grammatical parts, which can distinguish actions, objects, and conditions in a sentence. Knowledge of grammar is necessary for retrieving necessary actions (verbs) and objects (nouns) to generate accurate test cases.

Tagging the verbs in the sentence "The user logs in and accesses the dashboard" as verbs to recognize the actions in the test scenario. Since we identify verbs as actions, and this can be directly transformed to functional requirements in the software we can map these to input.

- **Tools**: SpaCy and Stanford NLP libraries also allow for implementation of POS tagging, and models can also be tuned against domain-level language patterns, enabling customization.

C. Named Entity Recognition (NER): NER is one of the fundamental techniques that helps in identifying the specific elements of requirements such as user roles, expected result and condition. NER does the same so we will not need to recognize such elements manually, thus saving effort we needed to identify them for test cases.

- **For example**, given the sentence "The user should get an error message if login fails" NER marks "user" as an entity, "error message" as the output, "login fails" as the condition. This is where NER comes into play, automatically extracting these components and hence paving the

way to mapping them against the set structured test case template.

- **Advantages:** NER helps the NLP models identify specific entities across a wider variety of requirement documents with different jargon without having any predefined rules.
- **Issues:** It may be necessary to define a custom NER model or labels for some domain-specific entities (especially if there are terms or acronyms unique to the project that are not readily available).

D. Answer Parsing: Another NLP method to reveal word relationships in the same sentence for further analysis — especially multi-condition use cases or complex, nested scenarios.

For instance, in "If payment fails, display an error in the system", dependency parsing links "if" as a conditional keyword, "payment fails" to the condition, and "display an error" to the action.

Dependency parsing may get complicated due to the hierarchical nature of sentences, but there are tools such as SpaCy which have pre-trained parsers that can parse a wide variety of linguistic structures.

IV. TEST CASE GENERATION BY SEMANTIC ANALYSIS

With semantic analysis, the framework can account for context of the sentences in requirement documents which allows for the correct identification of the non-trivial as well as trivial scenarios.

A. Contextual Embedding using Transformers: Contextual embedding, specifically using transformers such as BERT or T5 can help model the semantics of the requirement text. Whereas standard embedding's capture a single representation for each word, transformers provide contextual representations that change with surrounding words.

Process: Domain-specific datasets are used to fine-tune transformer models, such as BERT, allowing them to recognize unique terminology in software and produce concepts that will resemble more complex structure-containing sentences.

Example: Unable to Read Requirement (ex: "The application shall lock the account after three failed attempts to login"), BERT based embedding would understand "lock" in context of security and "failed attempts to log in" in context of triggering condition.

As transformer models see sentences in their entirety (all tokens combined) this forces them to learn a more contextualized representation of the problem, enabling them to learn both sequential dependencies and dependencies/conditions that a rule-based method may not be able to detect.

B. Classification based Intent Recognition: The intent classification determines the ultimate objective or goal that is intended to achieve on each requirement, resulting in increased relevancy and accuracy of test case generation.
Process: Classification models are developed using custom-trained models with labeled data for requirements. It will classify statements in intent categories i.e., login scenarios, error handling, transaction verification, etc.

For instance, for the statement "Generate a confirmation email, if the payment is successful", the model identifies the intent as "post-payment confirmation" and serves inspirational test conditions related to that intent.

C. Templates Matching with Rule-Based Patterns: Template matching uses the structured output from NER & Dependency Parsing where we map the structured output to the already defined test case templates. This guarantees, that when a test case is generated from a sentence, the test cases for all sentences have the same structure making the automation and integration into the testing framework easier.

- E.g. requirement: The user logs in successfully; [Template]: IF THEN; maps log in to Condition and successfully as Action. Since it is generated from a template, all your test cases will have the same structure making it easier to read and maintain for your testing teams.

D. Rule-based post-processing and validation

The next step is rule-based processing in which domain-specific rules are being applied to the result set, which assures that test cases not only conform to project expectations but also the requirements standards.

- **Domain-Specific Rule Refinement:** For certain industries, custom rules are created for regulatory compliance in finance or healthcare applications, among other areas.
- For example, in a banking application, you can have rules that for sensitive data actions (like an account balance check) there must be appropriate access conditions and/or appropriate error messages to be displayed upon failed actions.
- Custom rule set are generated and implemented for the project to filter out scenarios which are unnecessary or irrelevant and these are built in an iterative way and are based on the feedback coming from the testing teams.

E. Simulator-Driven Automated Validation: A set of automated test runs are carried out to validate that the generated test cases accurately reflect expected behavior. For example, simulation of test scenarios in a controlled environment via Selenium or Robot Framework.

- Benefits: Real application behavior vs generated test cases are compared, and discrepancies are highlighted aiding as direct feedback on refining the application of the NLP pipeline.
- E.g. if the requirement states there should be "expected errors", then the validated test cases will ensure that the expected errors do trigger an error message in application environments and cross-verify that requirement documentation is in sync with test execution.

V. IMPLEMENTATION AND CASE STUDY

A. Prototype Setup

This prototype framework utilized SpaCy for pre-processing steps and an already fine-tuned BERT model for semantic analysis. The requirement documents were tested for a fake e-commerce app that had specifications for login, cart, and checkout.

Evaluation and Results

- **Instrumentation:** Computed Precision, recall, and F1-score based on the ability of mismatched test cases to represent requirement intents correctly.

- **Efficiency:** This prototype was able to generate the test cases in 60% less time while achieving a similarity of > 85% with the manual test cases.
- **Key Takeaway**: There were several cases where the model handled easy requirements well, but sometimes failed for spaghetti vending machine type terms of function given only input and output criteria, and where multiple rules were specified, dependency parsing was an area that needed more work.

Challenges

Ambiguous requirements needed further rule-based filtering, while complex, multi-condition requirements needed more profound NLP configurations for consistency.

VI. IMPACT AND FUTURE RESEARCH

Effects on Software Development

The use of NLP-based test case generation reduces dependence on manual activity and consistency of tests increases while speed and efficiency of test development is also enhanced — a boon in agile and CI/CD environments.

Future Directions

- **More NLP Models Improvement**: Focusing transformer models to be more adaptable and accurate through fine-tuning on hidden documents of requirements.
- **Pipeline Conformance to Real-Time Requirement Changes**: Changing the framework to be used with real-time, incremental updates in agile environments to allow for continuous test generation.
- **Language-Agnostic Requirement Support**: Making the framework usable for projects written in different programming languages, solving the international challenge of software testing.

VII. CONCLUSION

Through NLP, this paper highlights that natural language understanding can tailor human-level examples as automated test cases, proving scalable, effective, and precise results to the problems posed by requirements-driven software as in modern era software test automation challenges. This approach simplifies and standardizes test creation through semantic analysis, template-based structuring, and rule-based validation, representing a significant improvement over traditional method.

REFERENCES

1. C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit," in Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, 2014, pp. 55–60. doi: 10.3115/v1/P14-5010.
2. G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural Architectures for Named Entity Recognition," in Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2016, pp. 260–270. doi: 10.18653/v1/N16-1030.

3. S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.
4. J. Pennington, R. Socher, and C. D. Manning, “GloVe: Global Vectors for Word Representation,” in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.
5. T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” in Proceedings of the International Conference on Learning Representations (ICLR), 2013.
6. J. K. Aggarwal and M. S. Ryoo, “Human Activity Analysis: A Review,” ACM Computing Surveys, vol. 43, no. 3, pp. 1–43, 2011. doi: 10.1145/1922649.1922653.
7. J. R. Quinlan, “Induction of Decision Trees,” Machine Learning, vol. 1, no. 1, pp. 81–106, 1986. doi: 10.1007/BF00116251.
8. K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, “Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network,” in Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, 2003, vol. 1, pp. 173–180.
9. R. M. Hierons, M. Harman, H. Singh, A. J. Offutt, P. G. Bishop, and R. Lipton, "Using Formal Specifications to Support Testing," in Software Quality Journal, vol. 19, no. 3, pp. 679-706, 2011. doi: 10.1007/s11219-011-9155-6.
10. Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011. doi: 10.1109/TSE.2010.62.
11. A. Rodriguez, H. Muccini, and H. Astudillo, "Requirements Engineering for Software Product Lines: A Systematic Mapping Study," in Information and Software Technology, vol. 52, no. 4, pp. 561-580, 2010.