https://doi.org/10.2352/issn.2168-3204.2022.19.1.09

How to generate and import functional test cases into project management software systems using natural language processing.

Ricardo Reyna
Department of Systems Engineering
Colorado State University
Fort Collins, CO USA
rickyrey@colostate.edu

Steven J. Simske
Department of Systems Engineering
Colorado State University
Fort Collins, CO USA
steve.simske@colostate.edu

Abstract

The main purpose of software testing is to identify what the software does and whether it matches its functional expectations. Applying a test plan allows one to prevent problems in early stages, identifying and addressing solutions before a project goes into production. Test cases play an important role during the software testing phase. A test case is a document with comprehensive details and sequences of actions to guide the software tester through the steps that need to be taken and the outputs that are expected.

The proposed system generates test cases based on scraped data that are used to interact with Natural Language Processing (NLP) approaches to generate functional test cases. A project management software (e.g., JIRA) is integrated with the JIRA python library to manage the test cases by the software tester.

KEYWORDS

Software Testing, Natural Language Processing (NLP), Automatic Testing, Test Case, Functional Testing.

1 Introduction

Software development is changing rapidly, and finding new software testing techniques is crucial to keep up. Companies are willing to invest as much as 50% of their software development resources towards testing [1]. The purpose of software testing is to examine all of the components and behaviors of the software system under consideration by applying validation and verification (V/V).

Testing can be performed either manually or automatically [2]. The process of writing the test cases without the assistance of any dedicated software tool is termed manual testing. Unfortunately, manual testing is time consuming and tedious work. However, software developers use test automation approaches to detect problems or defects in early stages of the development of the system. Humans still needed to design, build, and maintain automation scripts, but after built they can be deployed automatically. To fulfill the software quality, manual and automation approaches are required [3]. The motivation for this research is to save time and improve the quality of testing by providing an additional approach to generate test cases for software testing.

The remainder of the paper is organized as follows: In Section II, an overview of NLP and the problem that is addressed in this paper are presented. In Section III, relevant literature is reviewed. In Section IV, the implementation of the NLP algorithm and the python script that is used to generate and upload the test cases to JIRA is provided. In Section V, the results of the NLP approach and the limitations for generating test cases are provided. The concluding remarks are in Section VI.

2 Problem

Test cases are often not given the attention they deserve, which can lead to deleterious economic impacts [4]. Test cases are tools that a software tester needs to map the steps to follow, and a clear definition is essential to ensure that software is competitive, secure, and performs its expected purposes. However, performing software testing is a difficult task because it requires substantial investment of human hours. Therefore, researchers are focused on developing new software testing strategies to save human time investment (except on the front end) while still improving the quality of testing.

One of these areas of research, artificial intelligence (AI), has recently demonstrated that machines have the capability to meet or exceed human performance in specifically-crafted types of testing scenarios. Advances in natural language processing (NLP), a sub domain of AI [5], can be used to bridge the communication gap between computers and humans. Considering these advantages of NLP and automation, the proposed system will use NLP to generate test cases and automate the uploading of the test cases to project management software (e.g., JIRA). Enhancing and automating the creation of test cases and ensuring that the process goes smoothly is a goal of this paper.

NLP can be implemented to assign text to the following categories: speech understanding, automatic translation, question answering, information extraction, and text generation. These types of NLP implementations are implemented in systems used on a daily basis for applications spanning the range from translation software to voice assistants. Researchers are seeing some potential opportunities

for using NLP to aid the software quality of a system. Information extraction can be used as a backbone of generating test cases.

3 Literature Review

NLP can be implemented to generated test cases from functional requirements [6]. The author of [6] proposed a system that would automatically analyze functional requirement and extract important information to generate test cases. They highlight that this approach can minimize non-coverage of pertinent test cases. Their main goal was to reduce the effort and time consumed by testing [6]. One of the concerns with the requirements specifications being written in natural language is that natural language specifications can be ambiguous, incomplete ,and inconsistent [7]. In [7], the authors proposed a system to generate test cases from software requirements in natural language using NLP. They noted that their study can be used to analyze whether the requirements are satisfactory and properly understood. It is crucial to start testing in the early stages of the software development life cycle. This approach helps to reduce the number of defects and eventually the rework cost. In [8], the authors highlighted the importance of using test case prioritization (TCP) techniques to detect defects in early stages with the aid of NLP assistance. Based in their experiments, the risk strategy achieved the best performance across the other approaches [8].

Another proposed using NLP is highlighted in [9]. In this research, the approach was designed to enable the prediction of a test case failure for manual testing that can be implemented as a non-code/specification-based heuristic for testing selection, prioritization, and reduction. The results revealed considerable improvements, demonstrating that a simple linear regression model combined with a history-based feature can accurately predict test case failures. The implementation of NLP improves the accuracy of the traditional history-based predictions [9]. In [10], authors proposed a an overall system that consisting of three layers. The BeautifulSoup library is implemented for web scrapping, machine learning (ML) is used for predicting test cases, and Selenium is used to run the test cases. The results showed that the best classification model to generate test cases for each web element is SVM using tf-idf on top of count vectorizer. This approach [10] has elements that are similar to the system presented in our study. An important distinction is that our system is using a spaCy library for our NLP approach to extract data to generate the test cases.

4 Proposed System

4.1 Generation Approach

In this section, we explain the process of how NLP is used to generate test cases, as per Fig. 1, The steps are as follows: (1) navigating to the desired website. (2) Using Selenium for automatic login. (3) Employing a web scraping library to grab the desired data. (4) The data is exported to a CSV file. (5) With the aid of spaCy, NLP is then used to analyze and process the CSV file to extract and add the required data to generate the test cases. (6) The test cases are processed and reviewed. (7) The JIRA python script is then used to upload the test cases. (8) Finally, the software tester is ready to start testing, as per Fig. 1

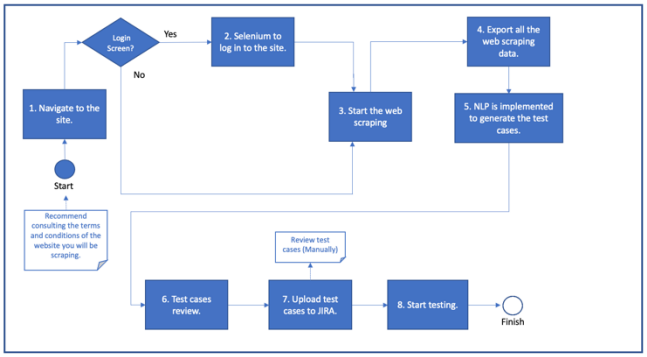

Figure 1. Test Case Generation Approach

Currently, the Python programming language is used due to its simplicity, power, and ability to process linguistic data [11]. Its large collection of useful libraries helps reduce the need for writing code from scratch. Jupyter was used since is a well-supported programming environment, and provides ease of communicating results. For this experiment, the BeautifulSoup v4, Pandas, Selenium, JIRA, and Requests libraries are employed.

The Requests library is used to navigate to the desired site (e.g., https://www.engr.colostate.edu/se/). BeautifulSoup is implemented to extract the data from the Systems Engineering news section and, with the aid of the Pandas library, to move the data to a csv file, as per Fig. 2

```
In [ ]: pip install pandas

In [ ]: pip install beautifulsoup4

In [ ]: pip install requests

In [ ]: import pandas as pd
        from bs4 import BeautifulSoup as bs
        import requests

In [ ]: URL = 'https://www.engr.colostate.edu/se/page/'

In [ ]: SENewsResults = []

In [ ]: for page in range(1, 11):

            req = requests.get(URL + str(page))
            soup = bs(req.text, 'html.parser')

            for titles in soup.findAll(attrs={'elementor-post__title'}):
                tittle = titles.find('a')
                if tittle not in SENewsResults:
                    SENewsResults.append(tittle.text)

In [ ]: df = pd.DataFrame({'SE News': SENewsResults})
        df = df.replace({"^\s*|\s*$":""}, regex=True)
        df.to_csv('/Users/ricardoreyna/Desktop/Jupyter_Project/SENewResultsData.csv', index=False, encoding='utf-8')
```

Figure 2. Python Script To Extract Data

The spaCy library was chosen for advanced NLP in Python. Its aim is to process and understand a large volume of text. Currently it supports tokenization and training for 60+ languages. The use of the spaCy's Ruled-Based Matching engine and components to find the words and phrases is one of interest. The same logic can be employed to generate test cases. This method allows navigation to the tokens within the document (e.g., data) and find relationships, as per Fig. 3

```
In [5]: import spacy
        from spacy.matcher import Matcher
        nlp = spacy.load("en_core_web_sm")

        matcher = Matcher(nlp.vocab)
        pattern = [{"TEXT": "Ph.D."}]
        matcher.add("SENewResultsData",[pattern])

        text = open('/Users/ricardoreyna/Desktop/Jupyter_Project/SENewResultsData.csv').read()
        doc = nlp(text)
        matches = matcher(doc)
        count = 0
        for _ in matches:
            count = count +1
        print("No of times the Ph.D. word is present:",count)

        No of times the Ph.D. word is present: 9
```

Figure 3. The Word "Ph.D." is Present Nine Times.

4.2 Rules – Based spaCy

It would be great to ask a computer to process, read, and understand specific text to generate test cases, such as clicking a button and having it return the correct URL page display or verify image 1 correspond to user 1. Manually repeating or constructing these test cases is costly and time consuming. In fact, with all of the technological advances in this area, it is crucial to focus on optimizing the costs, performance, and quality simultaneously. NLP can help reduce the testing time from hours to minutes. But before we implement the NLP, we need to instruct the computer on the basic concepts of text analysis. Our aim is to build a pipeline that can facilitate the process of making alterations to the data or extracting information that we can employ to generate the test cases. A pipeline is a collection of pipes that manipulates the data for your own purpose. Pipelines in spaCy are powerful and we can take advantage of them.

The first step that spaCy performs when the nlp object is employed is to tokenize the text to create the Doc object. Next, the Doc object is responsible for processing a couple of steps, which is known as the processing pipeline. Most of the trained pipelines in spaCy contain a tagger, a lemmatizer, a parser and an entity recognizer. Each pipeline element returns the processed to the Doc object, then continues with the next element [12], as a per Fig. 4

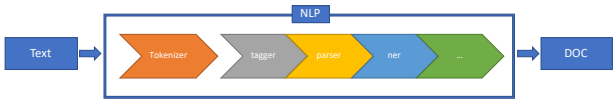

Figure 4. Pipeline process.
Source: Adapted from [12]

spaCy offers a large collection of different types of attribute rulers and matchers pipes that can be implemented for the desired pipeline.

- Tokenizer
- Tagger
- DependencyParser
- EntityRecognizer
- Lemmatizer
- TextCategorizer
- EntityRuler
- DependencyMatcher
- Matcher
- PhraseMatcher

See the spaCy documentation for a full list [12]. The EntityRuler and Matcher are of our interest.

spaCy EntityRuler

spaCy provides different types of ruled-based name-entity recognition (NER), such as EntityRuler. This class allows setting patterns with corresponding labels. To start, it is given instructions to find and label entities. Additionally, a new pipeline to integrate the EntityRuler into our model is needed.

In order to validate the investigation, a basic pipeline in spaCy was constructed. The EntityRuler is added to the pipe. This aids in adding patterns for the desired purpose. The spaCy small model "en_core_web_sm" was employed, and a Doc container to call the nlp object and pass the desired text to it was created. The text used represents four individuals who are in different stages of education. A simple **for** loop is run to extract the entities of each individual and the labels appear as a PERSON (e.g., people, including fictional) and FAC (e.g., building, airports, highways, bridges, etc.). Moreover, custom labels to assign to our four individuals are desired.

To that end, a ruler object was added to the pipe. The add_pipe is implemented to add the "entity_ruler" before the NER. A list of entities and patterns are created (e.g., M.S. individuals are assign to a GRADUATE Student label.) and add_patterns augments the ruler object. Finally, the **for** loop is run again to extract the entities and now each individual appears as desired, as per Fig. 5

Before adding the custom labels.

```
doc = nlp14("Jerry Li Ph.D., Sam White M.S., Ricky Reyna B.S., Pedro Infante AA,")
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Jerry Li Ph.D. PERSON
Sam White M.S. PERSON
Ricky Reyna B.S. FAC
Pedro Infante AA PERSON
```

After adding custom labels.

```
doc = nlp14("Jerry Li Ph.D., Sam White M.S., Ricky Reyna B.S., Pedro Infante AA,")
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Jerry Li Ph.D. PH.D Student
Sam White M.S. GRADUATE Student
Ricky Reyna B.S. UNDERGRAD Student
Pedro Infante AA COLLEGE Student
```

Figure 5. Extracting entities with custom labels.

The same logic could be applied during the creation of functional test cases. With the aid of BeautifulSoup v4 and Pandas libraries input control data, such as checkboxes, radio buttons, buttons, etc., can be collected. Building a pipeline with the EntityRuler factory could help constructing the test cases by assigning the correct labels to the input controls, as per Fig. 6

```
doc = nlp("LOGIN, SIGN UP, SUPPORT, ORDERS, CART, SMALL, MEDIUM, LARGE")
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
LOGIN BUTTON
SIGN UP BUTTON
SUPPORT BUTTON
CART BUTTON
SMALL CHECKBOX
MEDIUM CHECKBOX
LARGE RADIOBUTTON
```

Figure 6. Extracting entities and adding input control labels.

Considering this rule-based approach, the main focus for future research is reviewing different models including a machine learning based approach. The machine learning based approach would help avoid rules that are complicated to implemented. The advantage of the NLP framework in spaCy is that the pattern need not simply take a sequence of characters and look for a match; instead, a sequence of linguistic features that earlier pipes have identified can be employed [13].

spaCy Matcher

The spaCy Matcher is another class that can be implemented to generate test cases. There are some attributes taken by the Matcher.

- ORTH
- TEXT

- LOWER
- LENGTH
- IS_ALPHA
- IS_ASCII
- LIKE_EMAIL
- LIKE_URL

See the spaCy attributes for a full list [12]. For this scenario, the LIKE_URL are of our interest. The process mentioned on the spaCy EntityRuler applies for the spaCy Matcher, but not with the use of the pipeline. The new Jupyter notebook is created to integrated the Macher "from spacy.matcher import Matcher" with the spaCy medium model "en_core_web_md". The main differences between the EntityRuler and the Matcher is in how the data is extracted. The Matcher stores the information within the vocab of the NLP model with a unique identifier or a lexeme [13] [12]. In fact, the Matcher is not going to be stored on the Doc ends.

Going back to the implementation of the Matcher with the small model provided, it is observed that a matcher object to pass the vocab argument is needed. Then, the pattern list come to play. In this scenario, finding all the URL in our text is of our desired. The "LIKE_URL" key is employed in the dictionary to look for tokens that look like a URL. Two arguments are added: the label name and the pattern. The Doc container is employed to call the nlp object to pass the desired text to it. The matcher object is employed to call the Doc container. The first 10 matches are printed. The results display a list of tuples with three indices. The long number is equal to a lexeme. The next two numbers are assigned to a start token and end token. Now, navigation to the nlp vocab to find any of the integers and find what it corresponds to can occur, as per Fig. 7

```
In [21]: import spacy
         from spacy.matcher import Matcher

In [22]: nlp = spacy.load("en_core_web_md")

In [23]: matcher = Matcher(nlp.vocab)
         pattern = [{"LIKE_URL": True}]
         matcher.add("URLS", [pattern])

In [24]: with open ("/Users/ricardoreyna/Desktop/Jupyter_Project/SENewsLinks_05022022.csv","r") as f:
             text =f.read()

In [25]: doc = nlp(text)
         matches = matcher(doc)

In [32]: print (nlp.vocab[matches[0][0]].text)

         URLS

In [27]: print (len(matches))
         #we want to print the first 10
         for match in matches[:10]:
             print (match, doc[match[1]:match[2]])

         100
         (12479508882202238031, 2, 3) https://www.engr.colostate.edu/se/2022/04/29/april-student-of-the-month-peter-lobato/
         (12479508882202238031, 4, 5) https://www.engr.colostate.edu/se/student-exams/
         (12479508882202238031, 6, 7) https://www.engr.colostate.edu/se/2022/04/22/take-the-wheel-or-let-the-car-drive-autonom
         ously-graduate-student-develops-driving-simulations-of-cyberattacks/
         (12479508882202238031, 8, 9) https://www.mdpi.com/2071-1050/14/8/4633/htm
         (12479508882202238031, 10, 11) https://www.engr.colostate.edu/se/student-exams/
         (12479508882202238031, 12, 13) https://source.colostate.edu/bio-cybersecurity-student-challenge-winners-announced/
         (12479508882202238031, 14, 15) https://www.engr.colostate.edu/se/student-exams/
         (12479508882202238031, 16, 17) https://www.engr.colostate.edu/se/2022/03/28/student-qa-danny-call/
         (12479508882202238031, 18, 19) https://www.engr.colostate.edu/se/2022/03/25/jerry-li-ph-d-defense/
         (12479508882202238031, 20, 21) https://www.engr.colostate.edu/se/student-exams/
```

Figure 7. Implementing the LIKE_URL attribute by Matcher

With the advantages of the word vectors, a model can be trained and employ the similarity to match quickly and reliably. By calculating similarity, test cases can be generated. The similarity in spacy tells how close two words are, semantically. This is done by finding similarity between word vectors. A test case to click a button and having it return the correct URL page is readily created. Our results show that the "April Student of the Month: Peter Lobato" Button and the "https://www.engr.colostate.edu/se/2022/04/29/april-student-of-the-month-peter-lobato/" URL have a similarity of 0.912, as per Fig.8

```
In [1]: import spacy
        from spacy.matcher import Matcher

In [2]: nlp = spacy.load("en_core_web_md")

In [3]: matcher = Matcher(nlp.vocab)
        pattern = [{"LIKE_URL": True}]
        matcher.add("URLS", [pattern])

In [4]: with open ("/Users/ricardoreyna/Desktop/Jupyter_Project/URLS_05112022.csv","r") as f:
            data =f.read()
            print(data)
            remove_extra = data.replace('-',' ')
            print(remove_extra)
            text = remove_extra.replace('/',' ')
            print(text)

        https://www.engr.colostate.edu/se/2022/04/29/april-student-of-the-month-peter-lobato/
        https://www.engr.colostate.edu/se/2022/04/29/april student of the month peter lobato/
        https:  www.engr.colostate.edu se 2022 04 29 april student of the month peter lobato

In [5]: with open ("/Users/ricardoreyna/Desktop/Jupyter_Project/Tittles_05112022.csv","r") as f:
            text2 =f.read()

In [6]: doc = nlp(text)
        matches = matcher(doc)

In [7]: doc2 = nlp(text2)

In [8]: for match in matches:
            print (match, doc[match[1]:match[2]])

        (12479508882202238031, 3, 4) www.engr.colostate.edu

In [10]: URL = doc [8:]
         Tittle = doc2

         print(Tittle, "<->", URL, Tittle.similarity(URL))

         April Student of the Month: Peter Lobato <-> april student of the month peter lobato 0.9120883832996505
```

Figure 8. Applying the word similarity using spaCy.

These results show that similarities between buttons and URLs can be identified, and test cases generated from the results.

4.3 Upload Process

The results from the EntityRuler and Matcher can be used to generate test cases. With the aid of the Matcher, the similarity of a button to a URL was identified and test case generated. A simple python script to upload the test cases to a JIRA project can now be created. The JIRA python library can aid in the process [14] , as per Fig. 9

```
from helper import JiraHelper

user_name = ""
api_token = ""
server = ""
jira = JiraHelper(user_name, api_token, server)

# Test Data for Creating Issue
test_data = {
    "project": "project_url",
    "summary": "Match tittles with the correct URL",
    "description": "Verify that the " + Tittle +" button is assign to "+ URL"
    "issuetype": {"name": "Test"}
}

# Creating Test in Jira
jira.create_issue(test_data)

# Test Data for Updating Issue Fields
updated_test_data = {
    "summary": "test_summary",
    "description": "test_description",
}

# Updating Issue Fields in Jira
jira.update_issue_fields("issue_key", updated_test_data)

# Deleting Issue in Jira
jira.delete_issue("issue_key")
```

Figure 9. Connecting to JIRA to upload the test cases.
Source: Adapted from [13]

At this point, the software tester can start reviewing the test cases by navigating to the JIRA project.

5 Results

AI and ML, has improve the ability to test complex software systems With the aid of ML, one can train a machine to recognize any images and label them to generate test cases [15]. A test case classification methodology build on k-means clustering can improve regression testing [16]. NLP has proven that can be an important tool that can aid software testers during their testing [6] [7] [8] [9] [10]. The approach that is provided here can be found in section 4. This

research proves that with the aid of NLP and the benefits of using spaCy and other powerful python libraries, one can generate functional test cases. One goal of this work is sharing a different road to generate test cases and realizing this approach as a basis for future, more in-depth, work.

6 Conclusion

Software testing is a crucial part of the Software Development Life Cycle (SDLC). A novel approach to automating the process of generating functional test cases using NLP is provided. This approach can significantly reduce the time during the creation of test cases manually by a human. The technique was evaluated using two rule-based approaches. The EntityRuler and Matcher was employed to generate test cases. BeautifulSoup v4 and Pandas were implemented to collect data. In the context of building a pipeline using spaCy, collecting different types of data to have enough entities and patterns for the EntityRuler and Matcher is recommended. Word vectors can contribute to construct the test cases. Python script is executed to upload the test cases to JIRA. Future research will focus on reviewing all the results of the EntityRuler and Matcher.

References

[1] M. J. Harrold, "Testing: A roadmap," *Proc. Conf. Futur. Softw. Eng. ICSE 2000*, no. July 2000, pp. 61–72, 2000, doi: 10.1145/336512.336532.

[2] R. M. Sharma, "Quantitative Analysis of Automation and Manual Testing," *Int. J. Eng. Innov. Technol.*, vol. 4, no. 1, pp. 252–257, 2014.

[3] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation," p. 85, 2006, doi: 10.1145/1138929.1138946.

[4] P. D. Gregory Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," *Natl. Inst. Stand. Technol.*, p. 309, 2002.

[5] J. Hirschberg and C. D. Manning, "Advances in natural language processing," *American Association for the Advancement of Science*, p. 7, 2015.

[6] A. Ansari, M. B. Shagufta, A. Sadaf Fatima, and S. Tehreem, "Constructing Test cases using Natural Language Processing," *Proc. 3rd IEEE Int. Conf. Adv. Electr. Electron. Information, Commun. Bio-Informatics, AEEICB 2017*, pp. 95–99, 2017, doi: 10.1109/AEEICB.2017.7972390.

[7] R. P. Verma and M. R. Beg, "Generation of test cases from software requirements using natural language processing," *Int. Conf. Emerg. Trends Eng. Technol. ICETET*, pp. 140–147, 2013, doi: 10.1109/ICETET.2013.45.

[8] Y. Yang, X. Huang, X. Hao, Z. Liu, and Z. Chen, "An Industrial Study of Natural Language Processing Based Test Case Prioritization," *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2017*, pp. 548–549, 2017, doi: 10.1109/ICST.2017.66.

[9] H. Hemmati and F. Sharifi, "Investigating NLP-Based Approaches for Predicting Manual Test Case Failure," *Proc. - 2018 IEEE 11th Int. Conf. Softw. Testing, Verif. Validation, ICST 2018*, pp. 309–319, 2018, doi: 10.1109/ICST.2018.00038.

[10] N. Paul and R. Tommy, "Platform Using Machine Learning and Selenium," *2018 Int. Conf. Inven. Res. Comput. Appl.*, no. Icirca, pp. 851–856, 2018.

[11] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, First Edit. Sebastopol: O'Reilly Media, Inc., 2009.

[12] M. Honnibal, "spaCy," 2015. https://spacy.io/.

[13] W. Mattingly, "Introduction To Spacy 3," 2021. http://spacy.pythonhumanities.com/intro.html.

[14] S. Suwarnarajah, "How can we automate Jira using python?," 2020. https://github.com/sshajeeth/Jira-Automation (accessed Mar. 21, 2020).

[15] J. Arbon, "AI for Software Testing," *PNSQC Proc.*, pp. 1–19, 2017.

[16] Y. Pang, X. Xue, and A. S. Namin, "Identifying effective test cases through K-means clustering for enhancing regression testing," *Proc. - 2013 12th Int. Conf. Mach. Learn. Appl. ICMLA 2013*, vol. 2, pp. 78–83, 2013, doi: 10.1109/ICMLA.2013.109.

Acknowledgements

The author gratefully acknowledges Dun & Bradstreet for their support during his PhD studies. Furthermore, I would like to thank my colleague Vincil Bishop for his insightful input and good discussion about this work and future collaborations.

Author Biography

Ricardo Reyna received the MSE degree in software engineering from the Pennsylvania State University, in 2017. He attended the Johns Hopkins Whiting School of Engineering, in 2018. He recently completed his second year of PhD studies at the Colorado State University under the supervision of Professor Steve Simske. He is currently working as a Senior QA Engineer at Dun & Bradstreet. His research focuses on software testing, computer vision, AI, and ML.