DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Test Case Generation from Specifications Using Natural Language Processing

ALZAHRAA SALMAN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Test Case Generation from Specifications Using Natural Language Processing

ALZAHRAA SALMAN

Master in Computer Science, TCSCM
Date: August 10, 2020
Supervisor: Somayeh Aghanavesi
Examiner: Viggo Kann
School of Electrical Engineering and Computer Science (EECS)
Host company: Ericsson AB
Supervisor at host company: Sahar Tahvili
Swedish title: Testfallsgenerering från specifikationer med hjälp av naturlig språkbehandling

Abstract

Software testing plays a fundamental role in software engineering as it ensures the quality of a software system. However, one of the major challenges of software testing is its costs since it is a time and resource-consuming process which according to academia and industry can take up to 50% of the total development cost. Today, one of the most common ways of generating test cases is through manual labor by analyzing specification documents to produce test scripts, which tends to be an expensive and error prone process. Therefore, optimizing software testing by automating the test case generation process can result in time and cost reductions and also lead to better quality of the end product.

Currently, most of the state-of-the-art solutions for automatic test case generation require the usage of formal specifications. Such formal specifications are not always available during the testing process and if available, they require expert knowledge for writing and understanding them. One artifact that is often available in the testing domain is test case specifications written in natural language. In this thesis, an approach for generating integration test cases from natural language test case specifications is designed, applied and, evaluated. Machine learning and natural language processing techniques are used to implement the approach. The proposed approach is conducted and evaluated on an industrial testing project at Ericsson AB in Sweden. Additionally, the approach has been implemented as a tool with a graphical user interface for aiding testers in the process of test case generation.

The approach involves performing natural language processing techniques for parsing and analyzing the test case specifications to generate feature vectors that are later mapped to label vectors containing existing C# test scripts file names. The feature and label vectors are used as input and output, respectively, in a multi-label text classification process. The approach managed to produce test scripts for all test case specifications and obtained a best F1 score of 89% when using LinearSVC as the classifier and performing data augmentation on the training set.

Keywords— *Software testing*, *Test case generation*, *Natural Language Processing*, *Test case specifications*

Sammanfattning

Programvarutestning spelar en grundläggande roll i programvaruutveckling då den säkerställer kvaliteten på ett programvarusystem. En av de största utmaningarna med programvarutestning är dess kostnader eftersom den är en tids- och resurskrävande process som enligt akademin och industrin kan ta upp till 50% av den totala utvecklingskostnaden. Ett av de vanligaste sätten att generera testfall idag är med manuellt arbete genom analys av testfallsspecifikationer, vilket tenderar att vara en dyr och felbenägen process. Därför kan optimering av programvarutestning genom automatisering av testfallsgenereringsprocessen resultera i tids- och kostnadsminimeringar och även leda till bättre kvalitet på slutprodukten.

Nuförtiden kräver de flesta toppmoderna lösningarna för automatisk testfallsgenerering användning av formella specifikationer. Sådana specifikationer är inte alltid tillgängliga under testprocessen och om de är tillgängliga, så krävs det expertkunskap för att skriva och förstå dem. En artefakt som ofta finns i testdomänen är testfallspecifikationer skrivna på naturligt språk. I denna rapport utformas, tillämpas och utvärderas en metod för generering av integrationstestfall från testfallsspecifikationer skrivna på naturligt språk. Maskininlärnings- och naturlig språkbehandlingstekniker används för implementationen av metoden. Den föreslagna metoden genomförs och utvärderas vid ett industriellt testprojekt hos Ericsson AB i Sverige. Dessutom har metoden implementerats som ett verktyg med ett grafiskt användargränssnitt för att hjälpa testare i testfallsgenereringsprocessen.

Metoden fungerar genom att utföra naturlig språkbehandlingstekniker på testfallsspecifikationer för att generera egenskapsvektorer som senare mappas till etikettsvektorer som innehåller befintliga C# testskriptfilnamn. Engenskaps- och etikettsvektorerna används sedan som indata och utdata, respektive, för textklassificeringsprocessen. Metoden lyckades producera testskript för alla testfallsspecifikationer och fick en bästa F1 poäng på 89% när LinearSVC användes för klassificeringen och datautökning var utförd på träningsdatat.

Nyckelord— *Programvarutestning, Testfallsgenerering, Naturlig språkbehandling, Testfallspecifikationer*

Acknowledgements

I would like to begin this thesis by expressing my deepest gratitude to my supervisor at Ericsson, Sahar Tahvili, for her dedication, motivation, and guidance. I really appreciate the valuable help and support that I have received from her throughout the entire duration of this degree project.

I am also grateful to Somayeh Aghanavesi, my supervisor at KTH, for her continuous feedback and advice. Moreover, I would like to thank my examiner Viggo Kann for showing interest in my thesis subject and for taking the time to review my master thesis.

Last but not least, my profound gratitude goes to my loving family, partner and closest friends for being there for me and supporting me, not only during the degree project but throughout my years at KTH. None of this would have been possible without them.

Sincerely,

Alzahraa Salman

Contents

Chapter 1

Introduction

Software testing plays a crucial role in software engineering. It ensures the reliability and quality of a software system which can lead to high quality of the end product and thus the end-users' satisfaction [1]. However, one of the major challenges of software testing is that it is costly and time-consuming [2]. Furthermore, as software products grow in size and complexity, the efforts needed for performing software testing increase. Therefore, over the past years, there has been an increase in popularity in the topic of test optimization. Several strategies, approaches, and methods have been presented with the goal of optimizing the testing process.

Software testing involves designing and creating appropriate test cases that aim to achieve verification and validation of a system. Test case generation is considered to be one of the most difficult stages of the software testing life cycle (STLC) since test case generation consumes between 40% and 70% of the total effort [3]. Today, one of the most common ways of generating test cases is through manual labor. Manual testing involves a group of testers who operate and analyze a set of test case specifications manually [1]. Test case specification documents describe the functions of the software that are to be tested. Although manual software testing is still an important approach for the validation of computer systems [1, 4], it is a labor-intensive and time-consuming process that is prone to human errors [5]. By automating the generation of test cases, manual efforts can be eliminated, which can lead to time and cost reductions [6].

Specification-based test case generation is one way of generating test cases which involves examining the functionality of an application based on its specifications [7]. Specifications are used as one of the primary sources of information for deriving test cases. Test case specifications are most often written in natural languages, such as English, since these specifications need to be easy to specify, use, and understand [3]. The fact that most of the specifications are written in natural language, makes the usage of Natural Language Processing (NLP) techniques an appropriate methodology for generating test cases. Promising results have been obtained when using NLP techniques for identifying desired information from large raw data. These results have lead to increased interest in NLP techniques, especially in the field of automation of software development activities such as test case generation [8].

One of the main applications of NLP in software testing is specification mining which is also often used in the process of test case generation. Several previous studies [3, 9, 10, 11] have addressed the generation of test cases from specifications. However, most of the state-of-the-art proposed solutions require the usage of formal specifications during the process of generating test cases. Such formal specifications are not always available during the testing process and if available, they require expert knowledge for writing and understanding them.

In this thesis, an approach for generating integration test cases from test case specifications written in natural language is designed, applied, and evaluated. Machine learning and NLP techniques are used to implement the approach which involves mapping test case specifications to existing test scripts in C# containing the code needed for executing the test cases. The proposed approach is applied and evaluated on an industrial testing project at Ericsson AB in Sweden. Finally, the approach has been implemented as a tool for aiding testers in the process of test case generation.

1.1 Purpose

The purpose of this degree project is to investigate the possibility of improving the process of software testing by incorporating NLP and machine learning techniques for automatic test case generation. The project is conducted at Ericsson AB at their Global Artificial Intelligence Accelerator (GAIA) group who are interested in applying artificial intelligence and machine learning techniques in different domains. One such domain is software testing where artificial intelligence and machine learning can be used for optimization purposes. Research in the area of software testing is very important and beneficial since software testing plays a fundamental role in the process of ensuring the correctness of large-scale software systems. Therefore, optimization of software testing has been one of the topics that has been thoroughly researched over the years. One way of optimizing the process of software testing is by automating the generation of test cases. Applying machine learning techniques for automation of the test case generation process can help in minimizing the required time and effort for the testers. The results of this study can potentially lead to cost reductions for the company in the long run.

1.2 Research Question

The degree project entails solving a test optimization problem, which is the generation of integration test cases from test case specifications, written in natural language, by using NLP and machine learning techniques. The proposed approach of this study produces suggestions of relevant C# test scripts given unseen test case specifications. The C# test scripts are the files containing the code that should be run to execute the test case that is described by the test case specification. This leads to the following research questions:

- To what extent can test cases be generated by using Natural Language Processing on test case specification written in natural language?
- To what extent can automatic generation of integration test cases, from natural language test case specifications, replace manual generation?

1.3 Scope

There exists different sources of information that can be used for test case generation. Test cases can be generated from source code, binary code, UML models, formal specifications et cetera [12]. In this degree project, the test cases are generated by only using test case specifications that are written in natural language. Moreover, the tool implemented in this project only gives suggestions of existing C# test scripts for integration testing and can not produce unseen test scripts. The tool can be extended to generating test scripts in any programming language and in any testing level. In such case, mappings to test scripts in the specific programming language must be provided.

1.4 Thesis Outline

The report is divided into six chapters. The first chapter contains an introduction of the subject, the problem statement, the purpose, and the scope of the study. Chapter 2 provides the required background information about the topic. Software testing, test case generation, and relevant machine learning and NLP techniques are described in that chapter. In Chapter 3, the methodology of how the proposed approach is carried out is explained and motivated. Chapter 4 consists of the results of this study which are later discussed in Chapter 5. Threats to validity, sustainability aspects and future work are also explored and discussed in Chapter 5. Finally, the thesis is concluded in Chapter 6.

Chapter 2

Background

Chapter 2 provides an overview of the required background information for this project. First, a brief overview of software testing and test optimization is presented, including a description of test case generation. The next section focuses on specifications and how they are used in the testing process. Afterwards, a section about NLP is included, with some of the most common NLP techniques presented and explained. Also, the usage of machine learning techniques with NLP is described. The chapter closes with a section reviewing some relevant previous work in the area of software testing and NLP.

2.1 Test Optimization

According to IEEE international standard (ISO/IEC/IEEE 29119-1) [13], *Software Testing* is the process of analyzing a software item to detect the differences between existing and required conditions (i.e. defects and bugs) and to evaluate the features of the software item. Software testing plays a fundamental role in software engineering. It ensures quality of the system under development which can lead to the end-users' satisfaction and also high quality of the end product [1]. However, a main challenge in the software industry is the costs associated with software testing as it is a time and resource-consuming process which according to academia and industry can take up to 50% of the total development cost [14]. The testing process is often divided into several levels: unit, integration, system and acceptance testing levels. This thesis involves working with test cases at the integration testing level. Integration testing includes combining individual units and testing them as a group with the purpose of exposing faults in the interaction between integrated units [15].

In some practical scenarios, integration testing is considered as the most complex testing level since testing the behavior of combined units can result in a great increase in complexity [1].

Over the last decades, the topic of test optimization has increased in popularity and many researchers have presented strategies, approaches, and methods that could help in optimizing the testing process. There exists several ways to optimize the process of software testing such as test automation, test case prioritization [16, 17, 18], and test suite minimization [19, 20]. Moreover, employing artificial intelligence methods such as machine learning, deep learning, and NLP in the testing domain has received a great deal of attention. It has been shown that artificial intelligence can help in achieving more accurate results and save time in software testing [21]. In recent studies [21], it has been predicted that artificial intelligence-driven testing will be the leading technology of quality assurance work in the near future.

2.1.1 Test Case Generation

Improving the process of generating test cases is one way of optimizing software testing and increasing test effectiveness. A *test case* is defined by IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-199) [22] as:

Definition 2.1.1. *A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.*

Test cases are designed for every software system and as the system grows in complexity, more test cases are needed and thus the time and effort required for appropriate testing are increased. Therefore, optimizing the process of test case generation is crucial. The test case generation process can be performed manually, semi-automatically, or automatically. Today, one of the most common ways of generating test cases is through manual labor. However, there is a demand for a transition from manual to automated testing. Manual testing involves a group of testers who operate a set of test case specifications manually [23]. Although manual software testing is still an important approach for the validation of computer systems [1], it is a labor-intensive process that is prone to human errors [5]. By automating the generation of test cases, manual efforts can be eliminated, which can lead to test time savings and cost reductions of software development and maintenance [6].

Several approaches and methods for generating test cases have been presented throughout the years. Often, test cases are derived from system requirements and specifications. Prasanna et al. [7] divide test case generation approaches mainly into two categories:

- **Specification-based test case generation:** aims at examining the functionality of an application based on the specifications. This type of testing is limited by the quality of the specification [24].
- **Model-based test case generation:** the test cases are derived from a model that describes the functional aspects of the system under test [25]. This type of test case generation requires formal models to perform the testing.

In this thesis, specification-based test case generation, using test case specifications, is addressed. The approach of mapping test case specifications to test scripts in C# code is an important step in fully automating the test case generation process.

2.2 Specifications

Specifications are one of the primary sources of information for deriving test cases. There exist several types of specifications, such as software requirement specifications, functional requirement specifications, and test case specifications. A formal definition of a *specification* is provided by IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-199) [22]:

Definition 2.2.1. *The specification is a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and, often, the procedures for determining whether these provisions have been satisfied.*

These specification documents can be written as formal or informal specifications [26]. Although formal specifications are important for verifying the correctness of software, their usage is often not that common in software engineering practices since producing formal specifications requires specialists (e.g. mathematicians, logicians or computer scientists/engineers) for both writing and understanding the specifications [9]. A common alternative is producing informal specifications, often written in natural language [26]. The requirements and/or steps described in specifications need to be understood by both the users and the developers. Natural language seems to be an appropriate medium of communication understood by both parties.

2.2.1 Test Case Specification

A test case specification is a textual document describing the functions of the software that are to be tested. In this thesis, test case specifications are used as input in the process of generating test cases. During manual test generation, testers operate a set of test case specifications manually to extract useful information and generate test cases [1]. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-199) [22] provides a formal definition of a *test case specification* as follows:

Definition 2.2.2. *A test case specification is a document that specifies the test inputs, execution conditions, and predicted results for an item to be tested.*

A test case specification typically includes a description of the test purpose, inputs, testing procedures, and expected results. However, since test case specifications are often written in natural language by a group of testing engineers, the specifications can suffer from ambiguities and inconsistencies [27, 28]. One reason for this is the variety of the testers' writing and testing skills.

2.3 Natural Language Processing

Natural language refers to a language that humans use in their everyday communication [29] such as English, Swedish or Spanish. Natural languages can be ambiguous [30], contain large diverse vocabulary, words with different meanings, and can be spoken in different accents. Despite these challenges, humans can understand natural language without significant difficulties. Natural Language Processing (NLP) is a field combining computer science, artificial intelligence, and linguistics to explore how computers can be used to understand and structure natural language text or speech to develop useful applications [31]. Although NLP is experiencing rapid growth [30], developing NLP applications is still a challenging task since a precise and unambiguous language is required for the interaction with computers which is not the case with natural language.

The field of NLP is divided into two sub-fields [32]:

- **Natural Language Understanding (NLU)**: this sub-field involves the analysis of languages for the purpose of producing a meaningful representation [31].

- **Natural Language Generation (NLG)**: this sub-field refers to the process of producing natural language outputs from non-linguistic inputs [32].

Moreover, NLU is considered an AI-hard or AI-complete problem [33], meaning that a solution presupposes solving the problem of constructing a general artificial intelligence: creating a computer that is as intelligent as a human.

A complete NLP system is usually composed of several processing levels [29]. There are lower levels such as morphological analysis, syntactic analysis, and semantic mapping and higher levels such as discourse analysis and pragmatic analysis [9]. Which levels are considered and implemented in a system depends on the purpose of the work. Most of the current NLP systems tend to deal with lower levels of processing. This is because there has been more thorough research on using the lower levels. Also, the lower levels address smaller units of analysis, e.g. words and sentences, whereas the higher levels utilize world knowledge [29]. The work in this thesis considers morphological analysis, syntactic analysis, and semantic mapping for the generation of test cases. Discourse and pragmatic analysis are not performed since we are dealing with specifications where the language is more domain restricted.

Promising results have been obtained when using NLP techniques for identifying desired information from large raw data. These results have lead to increased interest in NLP techniques, especially in the use of automation of software development activities such as test case generation [8]. The documents used during the test case generation process, such as test case specifications and requirements specifications, are often written in natural language, making NLP techniques an appropriate methodology for generating test cases.

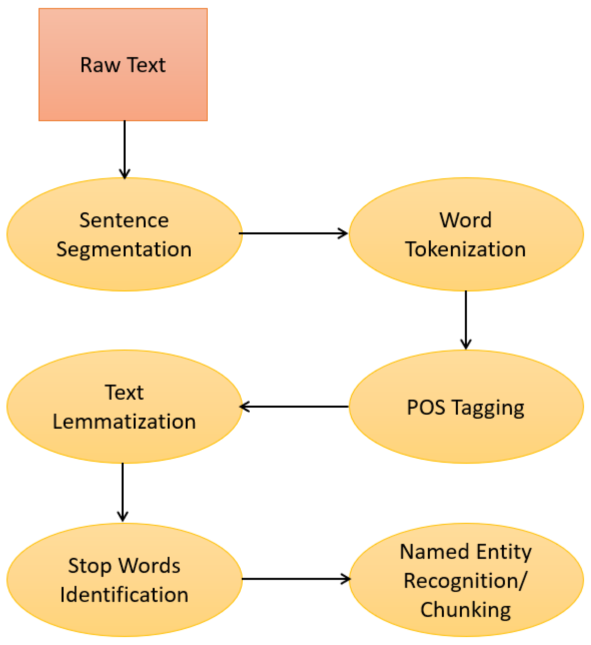


Figure 2.1: *NLP pipeline architecture including some common NLP steps.*

There exist several important NLP steps that are often applied to text documents for analysis. Figure 2.1 shows an example of a simple NLP pipeline consisting of some of the common NLP steps that can be applied to the specification documents during the process of test case generation. These steps can be applied collectively or individually depending on the application under development [8]. Text segmentation, which is considered the initial step of NLP, is the task of dividing a text into linguistically-meaningful units [34]. Sentence segmentation and tokenization are part of the text segmentation stage. Sentence segmentation is the process of dividing up a text into sentences. Tokenization, also known as word segmentation, involves breaking the sentence into individual words called tokens based on a predefined set of rules such as using white-spaces as a delimiter [30]. Text segmentation is a fundamental part of any NLP system since it is the words and sentences identified at this stage that will be passed to later processing stages such as part-of-speech taggers and parsers. Part-of-speech (POS) tagging is the task of assigning a morphosyntactic tag to each word and punctuation mark in a text. POS categories of words are defined by the morphosyntactic analysis using criteria from both morphology and syntax fields [9]. The algorithm that performs this analysis in NLP is called the POS-tagger. The NLP process can proceed with text lemmatization which is the process of mapping various forms of a word (e.g. helps, helped) to its most basic form (help), also known as lemma [30]. Furthermore, sentences in natural language often contain filler words, e.g. "and", "to", and "the", which during text analysis can contribute to noise. These words are referred to as *stop words* and are often removed from the sentences [35]. Another common NLP step is Named Entity Recognition (NER), also called chunking, where information is extracted from text such as locations, organizations, person names, et cetera [30].

2.4 Machine Learning and NLP

Machine learning techniques have become increasingly popular in NLP during recent years. Current state-of-the-art NLP approaches adopt machine learning algorithms [36]. Natural language is highly ambiguous and to manually analyze documents and extract features is a time-consuming process. To improve capturing linguistic knowledge and resolving ambiguity, machine learning models have been used for NLP in several tasks, such as POS tagging, NER and word segmentation [37].

2.4.1 Text Classification

Machine learning techniques can be divided into three categories: supervised learning, semi-supervised learning, and unsupervised learning [36]. Supervised learning is the most dominant technique for addressing problems in NLP [38]. Supervised learning tools such as classifiers play a significant role in many language processing tasks [37]. A classifier attempts to predict the class of a given object based on attributes describing the object. The supervised learning problem of text classification can be described as follows: given a text document, assign it a discrete label $y \in Y$, where Y is the set of possible labels [39]. As text documents usually belong to more than one predefined topic (class), multi-label classification is more appropriate to consider in these situations. Multi-label classification addresses the problem where multiple labels may be assigned to each instance, as opposed to the traditional task of single-label classification where each instance is only associated with a single class label [40]. In this thesis, multi-label classification is performed to solve the text classification problem. This is done to be able to achieve the goal of mapping test case specifications to one or more test scripts and make predictions for new test case specifications.

One popular technique for solving multi-label classification problems is the *One-Vs-All (OVA)* strategy. In the OVA strategy the problem is decomposed into multiple independent binary classification problems, i.e. one per class, where independent classifiers are built for each class [41]. Each classifier is then fitted to every input to determine which classes the input belongs to. The main assumption for using this strategy is that the labels are mutually exclusive as OVA treats each label individually, ignoring any possible correlations between the labels [42]. The advantages of the OVA strategy is its simplicity and interpretability since one can gain knowledge about a class by inspecting its corresponding classifier. The choice of classifiers is not limited and different classifiers may perform differently depending on the classification task at hand. In this thesis, two classifiers are applied: Linear Support Vector classifier and K-Nearest Neighbors classifier.

2.4.1.1 Linear Support Vector Classifier

Linear Support Vector Classifier (LinearSVC) works by creating a decision boundary to separate the training data into classes and classifies a test observation depending on which side of the boundary it lies [43]. The position and orientation of the decision boundary are determined by the data points closest to it, called the support vectors. The goal is to find the decision boundary that maximizes the distance between the boundary and the support vectors. This distance is referred to as the margin. Support vector classifiers allow some observation points to be on the incorrect side of the margin or of the decision boundary to increase the classifiers' robustness to individual observations [43]. The parameter C is the regularization parameter which determines the sensitivity to misclassified points. As C deceases, the margin increases, allowing more points to be on the wrong side of the margin. Figure 2.2 shows how the value of C influences the margin.

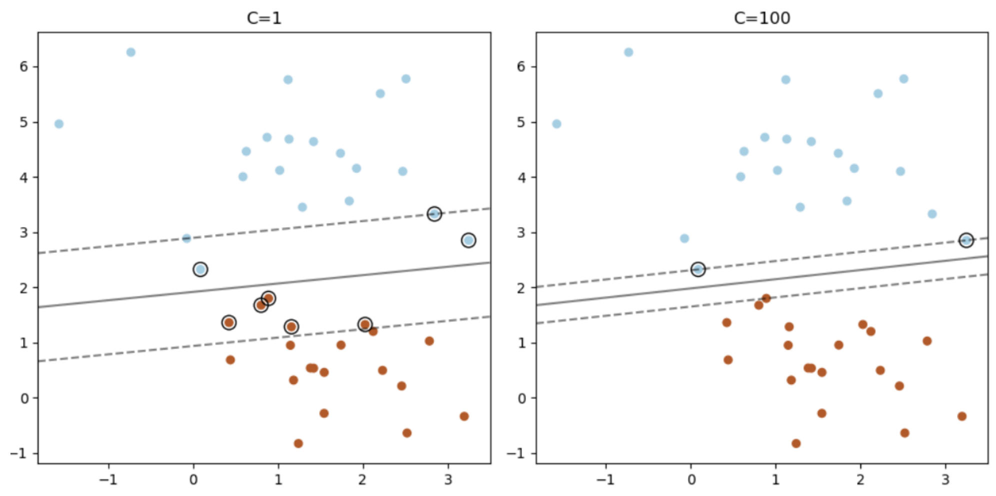


Figure 2.2: *LinearSVC with two different values of C. Support vectors are shown as the circled points. Left: C is set to 1, resulting in a wide margin and a higher tolerance for observations being on the wrong side of the margin. Right: C is set to 100 which leads to a smaller margin, reducing the tolerance for observations to be on the incorrect side of the margin. Image is taken from [44].*

2.4.1.2 K-Nearest Neighbors Classifier

K-Nearest Neighbors classifier (KNN) works by calculating the distances between a test observation x_0 and all other points in the training data set and selecting the K (a positive integer) closest neighbors to x_0 from that set. To determine the class of x_0, KNN looks at the classes of the K closest neighbors and makes a decision based on the majority vote of their classes [43]. The choice of the parameter K influences the results of the KNN classifier. Smaller values of K leads to more flexibility which corresponds to a classifier that has low bias but high variance while larger K causes the method to be less flexible corresponding to a classifier with low variance and high bias [43]. Figure 2.3 shows an example of the KNN approach with a small training data set consisting of six blue points and six orange points. The goal is to predict the class of the test observation point, shown as as a black cross, using the KNN classifier with $K = 3$.

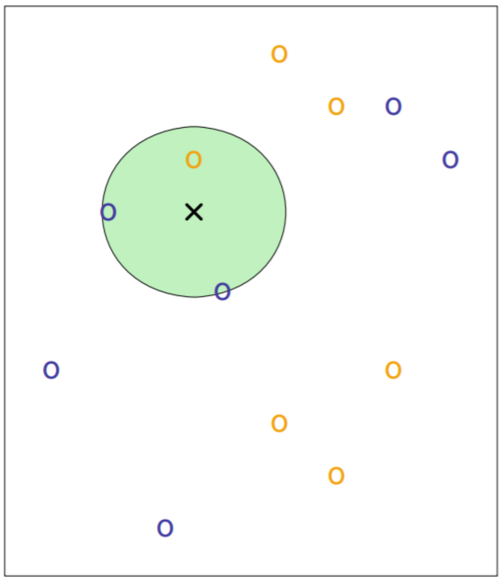


Figure 2.3: *A visualisation of the KNN approach, using $K = 3$. The data points belong to two different classes: blue and orange. The test observation at which a predicted class label is desired is shown as a black cross. The three nearest points to the test observation are identified in the green circle and it is predicted that the test observation belongs to the most commonly-occurring class, that is the blue class. Image is taken from [43].*

2.4.2 Feature Engineering

When working with machine learning models, selecting appropriate features is an important step. The quality of the classifier depends heavily on the features that are used to represent the data [30]. Selecting relevant features and representing them in a good way is a critical part of building a classifier. The process of extracting features from raw data and transforming them into formats that are suitable for machine learning models is called *feature engineering* [45]. One of the common feature engineering techniques for natural text is *bag-of-words*, which is a simplifying representation of a text document based on word count statistics [45]. A typical framework used in a supervised classification problem is shown in Figure 2.4. The process is split into two phases: training and prediction. During the training phase, feature sets are obtained by the feature extractor. To generate the model, the machine learning algorithm takes pairs of feature sets and labels them as input. During prediction, unseen inputs are fed to the same feature extractor to produce feature sets that are then fed into the model, which predicts the labels.

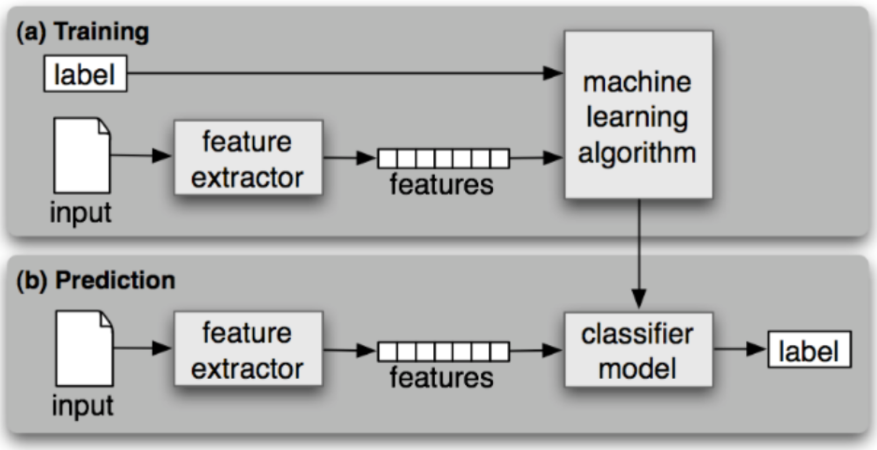


Figure 2.4: *Framework for supervised classification. Image is taken from [30].*

2.4.3 Data Augmentation

The performance of machine learning models depends highly on the size and quality of the training data [46]. However, many application domains do not have access to a sufficient amount of data. Also, preparing large data sets is often a tedious and time-consuming process. One solution for tackling such a problem is data augmentation, which is the process of increasing the amount and diversity of data by performing some operations on existing data [46]. Data augmentation is particularly popular in the field of computer vision where the amount of image samples in the training set can be increased by using strategies such as rotating, flipping or color varying the images. In NLP, performing data augmentation is a difficult task due to the complexity of languages. Kobayashi [47] explains that obtaining universal rules for transformations which assure the quality of the produced data is very difficult in NLP. The augmentation methods must be adapted depending on the type of text and the classification problem at hand. There exist some augmentation methods to tackle NLP tasks such as synonym replacement, random deletion/insertion of words and swapping words in a sentence [46].

2.5 Related Work

NLP is not a new topic in the area of software testing [48]. According to Hemmati and Sharifi [48], Specification Mining and Requirement Engineering can be considered as two main applications of NLP in software testing. Specification Mining is also highly involved in the process of test case generation. In the related literature, we can identify several previous studies [3, 8, 9, 10, 11, 49] that have addressed the generation of test cases from different types of specification documents. Table 2.1 contains a summary of the relevant research papers, where NLP techniques have been used for test case generation, including the benefits and drawbacks of the approaches.

Ahsan et al. [8] present a summary of different research papers that use different NLP techniques to generate test cases from requirements documents written in natural language. In the presented researches, various types of specification documents, mostly requirements specifications, have been used on different testing levels, mainly on system- and acceptance testing. It is stated in this paper that tokenization, POS tagging, chunking, and parsing are the major NLP techniques that were mostly used in the survey.

Boddu et al. [10] present a requirements analysis tool called RETNA that uses NLP and Text Mining (TM) technologies. The whole approach of this study is divided into two blocks: building a natural language analyzer and generating test cases automatically. RETNA takes natural language requirements as input, classifies them according to their type and complexity and uses user interaction to refine them. The user interaction helps in translating the natural language requirements to a logical format that is later used to generate test cases. The tool is dependent on heavy intervention from the user and the more the user interacts with it, the better the proposed tool performs. Also, the performance of RETNA depends on the quality of the natural language requirements specifications.

Carvalho et al. [9] propose a strategy called $NAT2TEST_{SCR}$ for model-based test generation, directly from natural language requirements specifications. The input specifications written in natural language are translated into formal Software Cost Reduction (SCR) specifications that are later used for automatic generation of test cases, using the commercial tool T-VEC. The $NAT2TEST_{SCR}$ strategy includes three processing phases: syntactic analysis, semantic analysis, and SCR generation. The syntactic analysis phase comprises a morphological and syntactic analysis of the input requirements to generate their corresponding syntax tree. The syntax trees are mapped into a semantic representation based on the case grammar theory during the semantic analysis phase. Finally, the last phase, SCR generation, delivers an SCR specification, which is to be used by the T-VEC tool to generate test cases. This approach requires however that the specifications are written according to a controlled natural language. The controlled natural language supported by this approach is limited, containing only a few lexical categories. This could reduce the usability of the $NAT2TEST_{SCR}$ strategy since requirements tend to be written in a natural language without the restriction to any controlled version of the language. The strategy is evaluated in four different domains, including the turn indicator of Mercedes vehicles, and the results were compared to a random testing technique, Randoop. The results showed that $NAT2TEST_{SCR}$ strategy outperformed Randoop.

Wang et al. [11] present an approach called Use Case Modelling for System Tests Generation (UMTG) that supports the generation of executable system test cases from use case specifications in natural language. NLP techniques are used in the approach for identifying test scenarios from use case specifications and to generate formal constraints, in the Object Constraint Language (OCL). The formal constraints capture conditions that are used for generating test input data. The main NLP analysis that UMTG relies on are: tokenization, NER, POS-tagging, semantic role labeling (SRL), and semantic similarity detection. This approach requires a domain model (e.g., a class diagram) of the system and the use case specification, which according to the authors are common in requirements engineering practices and are often available in many companies developing embedded systems. The approach is evaluated in two industrial case studies and promising results are obtained. UMTG manages to effectively generate system test cases for automotive sensor systems, addressing all test scenarios in the manually implemented test suite and also produce test cases for scenarios not previously considered by test engineers.

Verma and Beg [3] argue that expressing the requirements of a system using semi-formal or formal techniques can lead to limitations since expert knowledge is required for interpreting these requirements and only a limited number of persons possess this kind of expertise. These limitations are eliminated by using natural language to document requirements since natural language is understandable by almost anybody. The authors point out that the most used method for documenting requirements is a natural language, such as English, with around 87.7% of the documents being written in natural language. Due to these facts, the authors present an approach for generating test cases, for acceptance testing, from software requirements expressed in natural language. The approach makes use of NLP techniques such as parsing and POS tagging. After preprocessing the requirements and POS-tagging them, the requirements are parsed using parsers that are trained for processing general-purpose natural language text. The parsing step produces parsed trees that are converted to a knowledge representation graph which are combined to generate a single graph representing knowledge conveyed by requirement. By traversing the graph, important parts of a test case are identified and test cases are generated. The proposed method is applied to requirements containing a mathematical formula to demonstrate that their methodology removes the restriction of writing requirements using simple sentences only.

Table 2.1: *Summary of related work.*

Paper	Specification Document	Method	Benefits	Drawbacks
Boddu et al. [10]	Requirements specifications	NLP & TM technologies	Refines NL specifications to reduce ambiguity and inconsistency.	Need to translate requirements to logical format. Heavy human intervention.
Carvalho et al. [9]	Requirements specifications	NLP techniques, syntactic & semantic analysis	Generation of executable test cases including input data.	Requires usage of CNL. Need to translate requirements to formal SCR specifications.
Wang et al. [11]	Use case specifications & domain model	NLP techniques e.g. tokenization, NER & POS tagging	Does not impose any restricted dictionary or grammar.	Formal constraints in OCL must be generated. Domain model may not always be available.
Verma and Beg [3]	Software requirements documents	NLP techniques e.g. parsing & POS-tagging	No formal specifications are needed, only NL documents are used.	Only experimental validation. Abstract test cases are generated.

Most of the proposed methods and strategies in the related literature [9, 10, 11] require the usage of formal specifications during the process of generating test cases. In these studies, the requirements specifications written in natural language are translated into formal models that can later be used for test case generation, whereas in this thesis only test case specifications written in natural language are used without any translations. Test case specifications written in natural language are an artifact that is often available in the testing domain since natural language is easy to use and can be understood by both the users and the developers. Formal models on the other hand require expert knowledge to interpret. The work of Verma and Beg [3] make use of specifications written in natural language without translations. However, only abstract test cases are generated by their approach, as opposed to the work in this thesis where test scripts containing actual code are generated to execute the test cases. Moreover, the implementation of the proposed approach in this thesis has been inspired by the related work as some of the NLP techniques used in the previous work are used in this thesis as well. This include syntactic and semantic analysis of the specifications as in [9] and parsing, tokenization, and POS-tagging as in [3, 8, 11].

Chapter 3

Methods

This chapter describes the methodology of the proposed approach in this thesis. First, an overview of the approach is presented, including a pipeline containing the phases of the approach. This is followed by a section where the available data set is described. The next section focuses on the NLP techniques that are applied to the data for performing text analysis. Moreover, this section includes details about which features are extracted and for what reason. The upcoming section describes the preprocessing techniques that are used to enhance the quality of the data. Subsequently, the process of the classification is described and the algorithms used are presented. The chapter ends with a description of the performance measures that are used to evaluate the outcome of this study.

3.1 Overview of the Approach

The goal of this degree project is to design, implement, and evaluate an approach that automates the process of test case generation based on test case specifications written in natural language. In addition, the proposed approach is implemented as a tool that produces suggestions of C# test scripts that should be used for testing the test case in question given an unseen test case specification. Figure 3.1 shows the stages of the proposed approach.

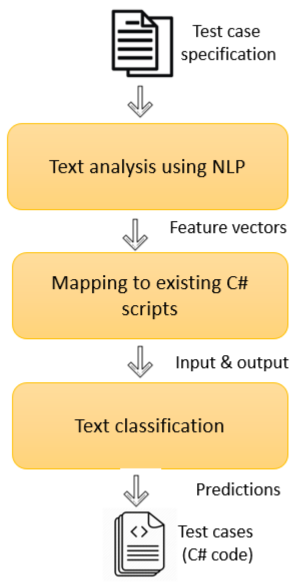
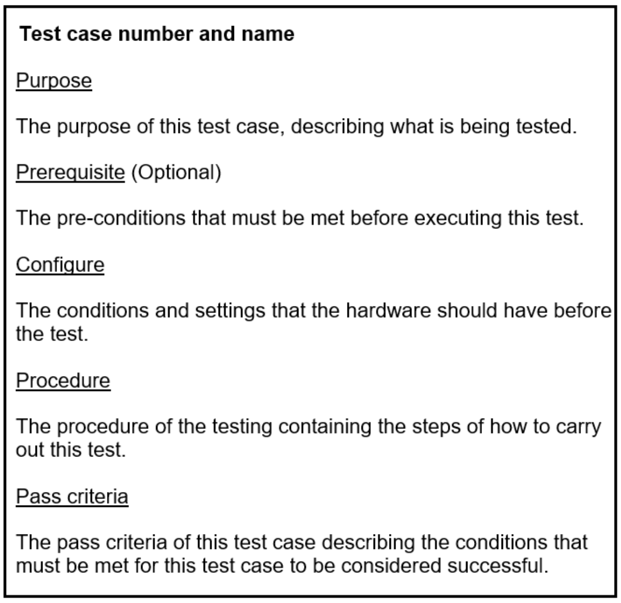


Figure 3.1: *The stages of the proposed approach.*

The approach is based on using NLP techniques in order to suggest test case scripts written in C# given test case specifications expressed in natural language. The test case specifications are first parsed and analyzed to generate feature vectors. The feature vectors, containing keywords representing the specification, are then mapped to label vectors consisting of C# code file names. The feature and label vectors are used as input and output, respectively, for the text classifier to perform training and prediction. Finally, a simple user-friendly user interface is implemented using the Python GUI (Graphical User Interface) package Tkinter [50]. The user is able to select a test case specification and the tool gives suggestions of test case scripts that are relevant for the specification in question.

3.2 The Dataset

Ericsson provided the data for this project which consisted of test case specifications (.docx files) written in natural language, specifically in English, and the C# test scripts for testing three of their products. The C# test scripts contain the actual code that is needed for running the test case described in the test case specification. In total, 130 test case specification documents with their corresponding C# test scripts (72 files) were provided. A test case specification can be mapped to one or more test scripts and a test script can be used to execute several test cases. Thus, there is not a one-to-one mapping between the test case specifications and the test scripts.



Test case number and name

Purpose

The purpose of this test case, describing what is being tested.

Prerequisite (Optional)

The pre-conditions that must be met before executing this test.

Configure

The conditions and settings that the hardware should have before the test.

Procedure

The procedure of the testing containing the steps of how to carry out this test.

Pass criteria

The pass criteria of this test case describing the conditions that must be met for this test case to be considered successful.

Figure 3.2: *The template of a test case specification used by Ericsson.*

As mentioned in Section 2.2.1, test case specifications written in natural language often suffer from inconsistencies and ambiguities. The test case specifications used by Ericsson are semi-structured. The specifications vary in length and writing style but have a similar structure as they contain the same sections: Purpose, Configure, Procedure, and Pass Criteria. Figure 3.2 shows an example of a test case specification designed by the test engineers at Ericsson. At the beginning of each test case specification document, the test case number and name are included. The first section of the specification is Purpose where the test case is described and the motivation for the test is explained. Next is an optional section, Prerequisite, which contains the pre-conditions that must be met before executing this test. A pre-condition can for example be that another test case needs to be executed before this one. The Configure section states the settings that the hardware should have for the test. In the Procedure section, the strategy of the testing is described, containing the steps of how to carry out this test. This can include for example measuring, recording and/or reading different parameters, devices, or status registers. Lastly, the specification includes a section about the pass criteria of this test case, i.e. the conditions that have to be met for this test case to be considered passed.

3.3 Tools

The approach proposed in this study was implemented using the programming language Python due to its popularity and its useful machine learning packages. In this section, the tools that are used for conducting the experiments in this study are presented.

Natural Language Toolkit (NLTK) [30] is one of the most used NLP libraries. It is written in Python and contains several useful packages. Tokenization, Stemming, Lemmatization, Punctuation, Character count, and word count are some of the packages that are included. NLTK is open-source and compatible with several operative systems. NLTK was used during the first stage of this project to parse and analyze the test case specification documents.

Stanford log-linear part-of-speech tagger (Stanford POS-tagger) [51] is a probabilistic POS-tagger developed by the Stanford Natural Language Processing Group. The Stanford POS-tagger is used to assign parts of speech to each word in a text, such as noun, verb and adjective. It is open-source and widely used in applications in NLP. The Stanford POS-tagger requires Java.

Tkinter [52] is Python's de-facto standard GUI package. Tkinter is open-source and cross-platform, meaning that the same code works on different operative systems. Tkinter was used in this project to implement the GUI of the proposed approach.

3.4 Text Analysis using NLP

The first stage of the proposed approach is to parse and analyze the test case specifications using several NLP techniques. The goal of this stage is to convert the test case specification documents to relevant feature vectors that later can be used during the classification process. Figure 3.3 shows the steps that are carried out during this stage. The Python library NLTK is used to perform most of these steps.

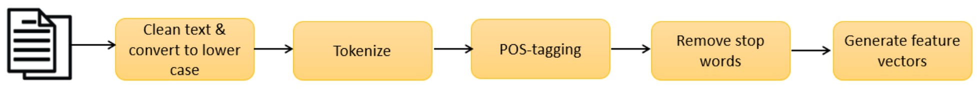


Figure 3.3: *The steps performed on the test case specification documents during the first stage of the approach, resulting in generation of feature vectors.*

The process began by cleaning the text of each test case specification document by removing all punctuation and converting all words to lower case since NLTK is case sensitive. The text of the test case specifications was then divided into word tokens by splitting on white-spaces, using NLTK tokenization. These word tokens were fed to the POS-tagger as input. It was observed that the default NLTK POS-tagger was not always able to parse all the words in a sentence correctly. Often, the problem was related to the positions of the components in the sentence, for example, if the sentence starts with a verb, such as “Check temperature ...”, the NLTK tagger identified the verb, “Check”, as a noun. To overcome this issue, the Stanford POS-tagger [51] was used instead. The Stanford POS-tagger has been trained with more imperative sentences compared to the default NLTK tagger and it yielded better results. However, the Stanford POS-tagger is slower in performing the POS tagging than the NLTK POS-tagger, but since the amount of test case specifications in hand is small, the time was not an issue. The last step before generating the feature vectors was removing all stop words. This was done to minimize noise in the data.

3.4.1 Feature Extraction

For most machine learning tasks, feature engineering, as defined in Section 2.4, must be considered to successfully train a statistical model. After parsing and analyzing the test case specification documents, features were extracted to form feature vectors. To derive the feature vectors from the test case specifications, a rule-based heuristic including a bag-of-words model was used. From observing the test case specifications provided as data set, it was noticed that these documents often contain verbs that give a good initial overview of what the test case is about. It was decided that these verbs should be collected as features. However, the amount of verbs in each specification is very low, resulting in very short feature vectors that would not be descriptive of the specification. Therefore, nouns and adjectives were collected as features as well. The frequency of each word token was calculated using the bag-of-words model and for a noun or an adjective to be regarded as a feature, it had to appear at least two times in the specification. This was done to avoid collecting irrelevant words. Figure 3.4 shows an example of a feature vector extracted from a test case specification. The words (features) in the vector represent the test case specification.

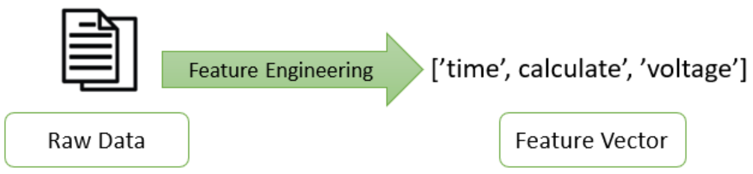


Figure 3.4: *An example of a feature vector extracted from a test case specification (raw data).*

Each feature vector was later mapped to a label vector containing the names of the C# code scripts corresponding to the test case specification that is represented by that feature vector. Two arrays were constructed, $Features$ and $Labels$, where $Features[i]$ contains the feature vector belonging to the ith test case specification and $Labels[i]$ contains the label vector belonging to the ith test case specification. The $Features$ array is used as input for the text classifier and $Labels$ as the output.

3.5 Text Classification

In this section, the process of preprocessing the data vectors, obtained in the previous stage, is described. Also, details about the classification process are included.

3.5.1 Data Preprocessing

The performance of a text classifier often depends on the size and quality of the training data. In this project, the size of the training data set was small, therefore data augmentation on the feature vectors, extracted from the test case specifications, was performed to increase the size and variety of the training data. Data augmentation can boost performance on text classification tasks [46]. Synonym replacement and swapping words in a sentence are some common data augmentation methods in NLP. However, these methods do not suit the type of text and classification problem in this project. Synonym replacement was not used since the approach would lead to higher dimensionality which would make it difficult to get statistically meaningful information on the distribution of the data. Swapping words in a sentence was not suitable either since the order of the words is not important when transforming the sentence to a feature vector. Data augmentation in this project was done by choosing elements from the power set, the set of all subsets, of a feature vector. Not all subsets were selected and added to the training data, for example subsets containing only one word were excluded. The newly generated feature vectors were labelled with the same label as the vector they were generated from. Figure 3.5 shows this data augmentation technique applied to an example feature vector. As seen in Figure 3.5, multiple new feature vectors are generated from an existing feature vector.

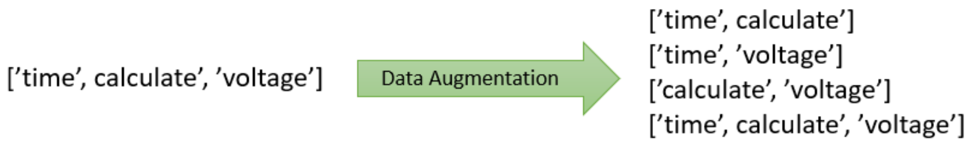


Figure 3.5: *Feature vectors obtained after performing data augmentation on a single vector.*

After performing data augmentation on the training data, all data were encoded using one-hot encoding to keep track of the categories (features and labels) in a numerically meaningful way. Each feature and label in a test case specification was transformed to either 1 or 0 depending on whether that feature or label belonged to that test case specification.

3.5.2 Multi-Label Classification

Since each test case specification can be mapped to one or more C# test scripts, a method that is able to predict multiple labels for each instance is needed. Therefore, multi-label classification was used in the text classification stage of the approach. The data from the three Ericsson products were split into training and testing sets in a stratified fashion. Test case specifications from two of the products (100 specifications) were used for training and the remaining product for testing (30 specifications). The reason for not splitting the data randomly is to avoid problems with missing labels. If the data were split randomly, there is a risk that some classes become missing from the training data set which is a common problem when having low occurrence frequency of the classes. In this project, some of the labels occurred very rarely. Therefore, when splitting the data randomly, some of these labels would be missing from one side of the split. Missing some of the labels in the training set is problematic since the classifier will not be able to train with these labels and thus will not predict them for any instances.

To perform the multi-label classification, the OVA strategy was used with the assumption that the labels in the data set are mutually exclusive. The OVA strategy was used in this project for its simplicity and because it has proven to yield good performance in practice [41]. Two classifiers were applied on the generated vectors: LinearSVC and KNN classifier. LinearSVC is one of the algorithms that performs well on a range of NLP-based text classification tasks. Grid search was performed to choose the best value of the regularization parameter, C, for LinearSVC. The C value of 0.6 gave the best results and was therefore selected. Also, LinearSVC is relatively fast and does not need much data for training. KNN classifier was also applied and its results were compared to the results achieved by the LinearSVC. KNN is easy to implement, requiring only two parameters: the number of neighbors, K, and the distance function. Two values for the parameter K were tested during the experiments, 1 and 3. Instead of using the default distance function for KNN (Euclidian distance), Sørensen-Dice distance [53, 54] was used. Sørensen-Dice distance is a metric intended for boolean-valued vector spaces which is suitable in this study since our data was converted to boolean values during the one-hot encoding.

3.6 Evaluation Method

To evaluate the approach, the predictions made by the classifier were compared to a ground truth table. The ground truth consisted of a matrix containing test case specifications as rows and C# test script file names as columns. A test script is marked with 1 in the matrix cell if it is mapped to the test case specification of that row and 0 otherwise. The number of negative elements in the data set (test scripts marked with 0) is much larger than the number of positive elements (test scripts marked with 1), resulting in an imbalanced data set. During the evaluation phase the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) were calculated to be used later for the calculation of different performance metrics. Figure 3.6 shows how the ground truth label set and the prediction set relate to each other.

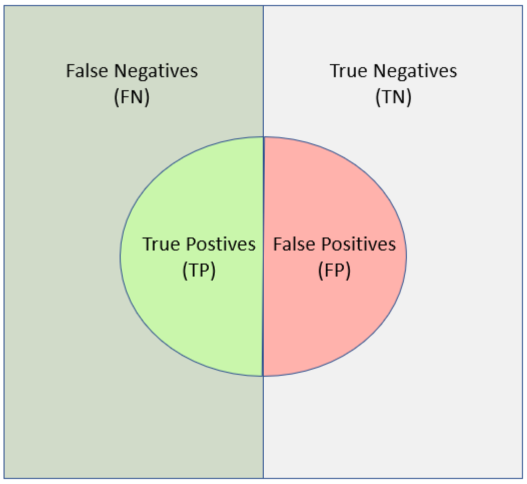


Figure 3.6: *The relation between ground truth set and prediction set. The left side of the box contains the relevant elements, i.e. all the labels that exist in the ground truth set. The right side of the box contains the negative elements of the ground truth set. The circle denotes the positive predicted elements, i.e. positive elements in the prediction set.*

The choice of a suitable performance metric is crucial since it influences the measured performance of an approach. Known metrics, such as accuracy, need to be adjusted in order to be used for multi-label classification. In this thesis, using the default Scikit-learn [44] accuracy classification score was not sufficient since this accuracy function computes subset accuracy for multi-label classification problems, meaning that the set of labels predicted for a sample must exactly match the corresponding set of labels in the ground truth. Subset accuracy is also called an Exact match and is the most strict metric. Since this accuracy indicates the percentage of samples that have *all* their labels classified correctly, it is not suitable in our multi-label classification task where a misclassification is no longer a strict wrong or right. A prediction containing a subset of the actual labels is considered better than a prediction that contains none of them. To solve this problem, the comparison between the predicted labels and the ground truth labels was done element-wise instead of vector-wise. Moreover, the data set in this project is imbalanced as it contains much more negative samples than positive ones. It is known that using accuracy as a performance metric for imbalanced data sets can yield in misleading results [55]. Therefore, it was decided to implement and use a balanced accuracy function [56] adjusted for multi-label classification. In the balanced accuracy function, the number of true positive and true negative predictions are normalized by the number of positive and negative samples, respectively. The balanced accuracy function is calculated according to Equation 3.1, whereas the usual accuracy function, i.e. the proportion of correct predictions, is calculated according to Equation 3.2.

$$Balanced\,Accuracy = \frac{1}{2}\Big(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\Big) \tag{3.1}$$

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \tag{3.2}$$

Additionally, Precision, Recall, and F1 score were also calculated and used to measure the performance of the proposed approach since these metrics put more weight on the True Positive elements which in this study are considered to be of most importance. The precision is the number of correctly predicted C# scripts divided by the total number of C# scripts predicted by the proposed approach. This indicates how many of the selected items are relevant.

$$Precision = \frac{TP}{TP+FP} \tag{3.3}$$

The recall is the number of correctly predicted C# scripts divided by the total number of existing C# scripts in the ground truth. This indicates how many of the relevant items are selected.

$$Recall = \frac{TP}{TP+FN} \tag{3.4}$$

F1 score is a harmonic average of Precision and Recall and is used as a compound measurement. F1 score reaches its best value at 1 (corresponding to perfect precision and recall) and worst at 0.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3.5}$$

F1 score was calculated using the Scikit-learn's *f1_score* function [44] where the *average* parameter was set to 'samples' in order to calculate metrics for each instance and find the average. This is preferable for multi-label classification.

Chapter 4

Results

The feasibility of the proposed approach in this thesis is evaluated by performing experiments on products from an ongoing testing project at Ericsson. The data of the project was analyzed and a summary of the statistics collected is included in the first section of this chapter. In the next section, the performance metrics obtained from the classification process are presented. The last section presents the tool implementation of the proposed approach.

4.1 Data Analysis

This section includes some statistics collected on the data, test case specifications and their corresponding test scripts. Figure 4.1 shows the relationship between the number of test case specifications and the number of test scripts. There are no one-to-one mappings between the test case specifications and the test scripts, i.e. a test case specification can be mapped to several test scripts and a test script can be used for several test case specifications. However, as can be seen in Figure 4.1, most of the test case specifications (97/130 specifications) are mapped to only one C# test script (not the same one). In the data set, a test case specification is at most mapped to 6 C# test scripts and on average there are 1.3 C# scripts corresponding to a test case specification.

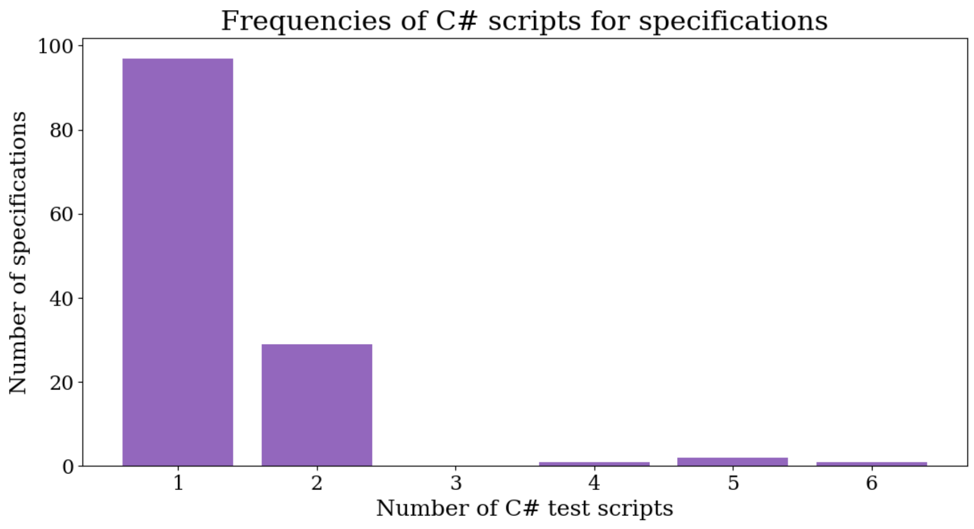


Figure 4.1: *Frequencies of the C# test scripts for test case specifications.*

There were 437 features (keywords describing a specification) collected from the 130 specifications in the data set. The maximum number of features per specification is 86 and the minimum number of features is 2. On average, 17.9 features were extracted from each specification. Figure 4.2 shows the number of specifications that the ten most frequent features appear in. Even though the natural language in the specifications in this project is not restricted, it can be seen that some of the most frequent features occur in almost half of the available test case specifications. This may indicate that there is some underlying structure in the vocabulary used for writing the specifications.

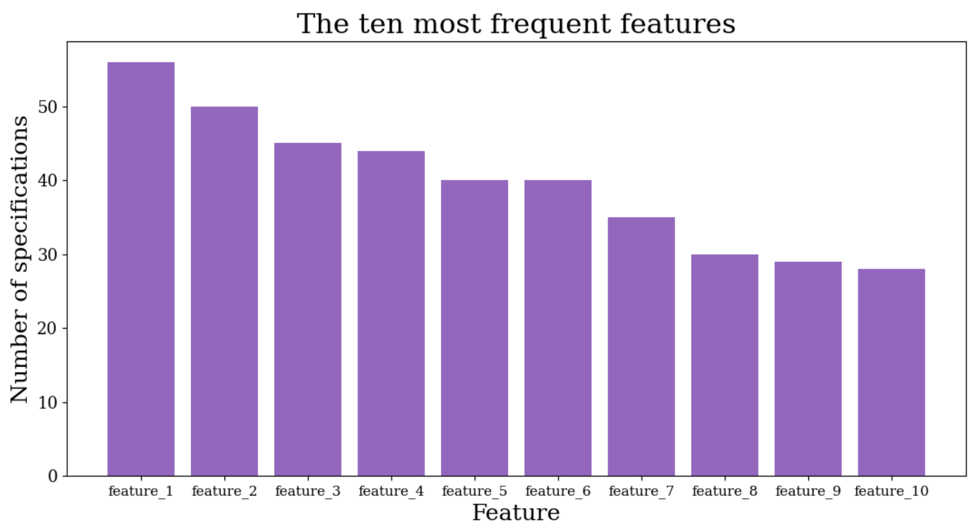


Figure 4.2: *The frequency of the ten most frequent features in the test case specifications.*

4.2 Performance Evaluation

The machine learning model was trained with two products, containing 100 test case specifications and 72 test scripts, and tested on one product. The performance of the proposed approach is measured by using Precision, Recall, F1 score, and balanced Accuracy as described in Section 3.6. Tables 4.1 and 4.2 show the computed results of the mentioned performance metrics for two classifiers LinearSVC and KNN, respectively.

Table 4.1: *The F1 score, Precision, Recall, and balanced Accuracy metrics using the data from three Ericsson products. The results are obtained* ***with data augmentation****.*

Classifier	**F1 score**	**Precision**	**Recall**	**Accuracy**
LinearSVC	0.893	0.850	0.983	0.982
1-NN	0.794	0.766	0.850	0.909
3-NN	0.794	0.766	0.850	0.909

The presented results in Table 4.1 are obtained when performing data augmentation on the training set whereas the results in Table 4.2 are retrieved without using data augmentation.

Table 4.2: *The F1 score, Precision, Recall, and balanced Accuracy metrics using the data from three Ericsson products. The results are obtained* ***without data augmentation****.*

Classifier	**F1 score**	**Precision**	**Recall**	**Accuracy**
LinearSVC	0.816	0.800	0.850	0.920
1-NN	0.806	0.767	0.883	0.938
3-NN	0.788	0.783	0.833	0.909

The performance results shown in Tables 4.1 and 4.2 indicate the the best performing model for the proposed approach is the one using LinearSVC as classifier with data augmentation on the training set which achieved an F1 score of 89.3% and a balanced accuracy of 98.2%.

The results in Tables 4.1 and 4.2 are calculated for each instance of the testing set, i.e. for each test case specification, and averaged over the total amount of test case specifications in the testing set. For the best model, the number of true positives, true negatives, false positives, and false negatives for each test case specification are shown in Table 4.3. The results for LinearSVC in Table 4.1 are calculated using the values in Table 4.3 by averaging the values of this table row-wise.

Table 4.3: Comparison of the results of LinearSVC and the ground truth, for each of the test case specifications in the testing data set, based on 72 test scripts. TP: True Positive, TN: True Negative, FP: False Positive and FN: False Negative.

Specification	TP	TN	FP	FN
1	2	68	2	0
2	1	71	0	0
3	1	71	0	0
4	1	70	1	0
5	2	69	1	0
6	1	71	0	0
7	2	70	0	0
8	1	71	0	0
9	1	71	0	0
10	1	71	0	0
11	1	71	0	0
12	1	70	1	0
13	1	69	2	0
14	0	70	1	1
15	1	71	0	0
16	1	71	0	0
17	1	69	1	1
18	1	71	0	0
19	1	71	0	0
20	1	70	1	0
21	1	71	0	0
22	1	70	1	0
23	1	71	0	0
24	1	70	1	0
25	1	71	0	0
26	1	71	0	0
27	1	71	0	0
28	1	71	0	0
29	1	71	0	0
30	1	71	0	0

The scalability of the approach was tested by training the best model with different data sizes, i.e. with different number of test case specifications. LinearSVC was used as classifier during this experiment and the F1 score obtained for each size was recorded. Figure 4.3 shows the performance of the approach for different sizes of the training set. It is visible in Figure 4.3 that by increasing the training set size, the performance increases as well. The algorithm converges when it reaches around 70 test case specifications.

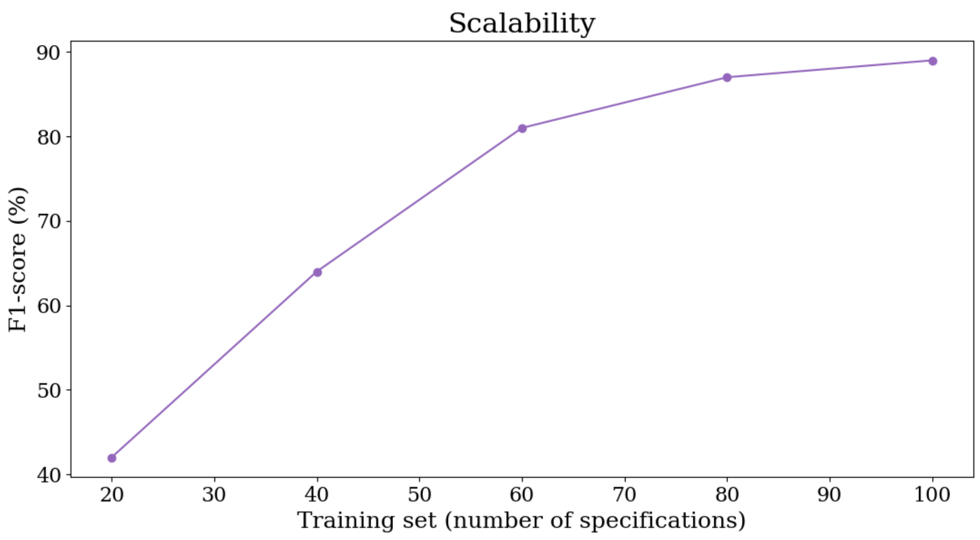
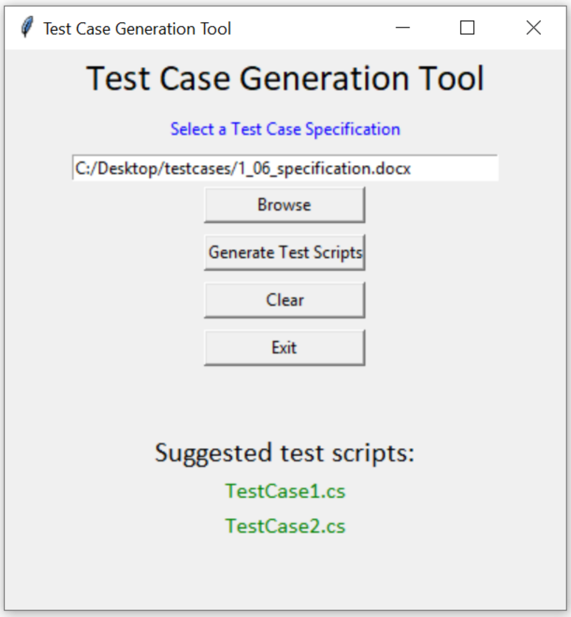


Figure 4.3: *The scalability of the approach showing the F1 scores obtained when training the model with different amount of test case specifications. LinearSVC is used as the classifier.*

4.3 The Tool

The proposed approach has been implemented as a tool for generating test cases from test case specifications. The tool uses the methodology that yielded the best results on the provided data set, that is LinearSVC as classifier with data augmentation performed on the training set. Figure 4.4 shows the GUI implemented for this approach.


Test Case Generation Tool
Test Case Generation Tool
Select a Test Case Specification
Browse
Generate Test Scripts
Clear
Exit

C:/Desktop/testcases/1_06_specification.docx

Suggested test scripts:
TestCase1.cs
TestCase2.cs

Figure 4.4: *The graphical user interface of the approach.*

The user can easily select a new test case specification file as input for the tool. The specification document must be written in English and have a .docx format. The expected output of the tool is suggestions of known C# test scripts file names that are relevant for the test case specification in question. These test scripts contain the code that should be run to execute the test case described in the selected test case specification. The tool has been able to suggest test scripts for all test case specifications from the products provided by Ericsson. It is important to have in mind that the tool is only able to suggest test scripts that the model has trained with, i.e. test scripts belonging to the 72 C# test scripts that were provided as data in the beginning of the project.

Chapter 5

Discussion

In this chapter the results of the proposed approach and the different performance metrics are discussed. The chapter includes a section discussing the threats to validity, limitations and challenges faced when conducting this study. Also, the sustainability and ethical aspects of this thesis work are reviewed. The chapter closes with a section presenting the possibilities for future work of this thesis.

5.1 Analysis of the Results

The software testing at Ericsson is performed manually where a group of testers operate a set of test case specifications to generate integration test scripts. The proposed approach in this thesis improves this process by automatically suggesting relevant test scripts, for integration testing, given a new test case specification. It is clear from the results, in Chapter 4, that the proposed approach has effectively managed to generate relevant test cases, using test case specifications written in natural language. The research questions of this thesis, concerning to what extent NLP can be used on test case specifications for test case generation and how well automatic test case generation, from natural language test case specifications, can replace manual test case generation, have been answered. It is safe to say that this degree project has shown how NLP techniques can be incorporated in the test case generation process and an evaluation of the method's performance was made. Both syntactic and semantic analysis have been performed on the test case specifications to derive feature vectors, which were an essential part of the approach as they were used as input during the text classification stage. The results were promising, showing that the automatic test case generation approach has achieved high accuracy and F1 scores. This means that the approach has managed to produce relevant test scripts for the selected test case specifications. The best performance results were obtained when using LinearSVC as classifier with data augmentation on the training set, yielding a best F1 score of 89.3%. Moreover, the tool, presented in Section 4.3, which implements the proposed approach, can aid the testers by relieving some of the manual operations performed on test case specifications. Despite the promising results, there are some scenarios where the proposed approach can not fully replace manual test case generation, which will be further discussed in this section.

It is interesting to discuss how the data have effected the results of this study. One important aspect of the data in this project is that there are no one-to-one mappings between the test case specifications and the test scripts. Figure 4.1 shows that the majority of the test case specifications (97/130) are mapped to only one test script. Since there only exists 72 test scripts in the whole data set, it can be inferred that some test scripts are used for more than one test case. The model used in the proposed approach is trained with only these 72 C# test scripts and therefore, when the user selects a new test case specification, the tool is only able to suggest test scripts that belong to the set of 72 scripts. Thus, the tool can replace manual test case generation to a large extent as long as the desired outcome is a subset of the 72 test scripts. The model can not suggest unseen test scripts. If that is the case, the testing engineers must generate the script manually. The proposed approach acts as an important first step in supporting a fully automated test case generation process. As discussed, for some cases, human intervention is needed to generate the test cases. However, the model can be extended further by including more data in the training phase.

In Table 4.3, it can be seen that the number of true negatives is large, in comparison to the other values, for all test case specifications. A true negative represents that the model correctly omitted a test script from the prediction. Since the data in this project is imbalanced, accuracy is a poor measurement to evaluate the model. Considering the default accuracy function, shown in Equation 3.2, the score increases in conjunction with the number of true negatives. With an imbalanced data set, this would lead to misleading results. Therefore, a balanced accuracy function was calculated instead where true positive and true negative predictions are normalized. Furthermore, in many studies, the number of false positives is regarded as having a negative impact on the quality of the model. However, in this study, a falsely predicted test script could still be of use to the testers at Ericsson since it could give insights on test scripts that may not have been obvious for the specific test case. This means that the number of false positives does not pose as big of a negative impact on the evaluation of the model as one might think. The number of false positives has a direct affect on the Precision score, as a lower rate of false positives will lead a higher Precision. But since the goal in this study is not to minimize the number of false positives, Precision alone is not sufficient to describe the quality of the approach. In this study, it is more important to get as few false negatives as possible. The number of false negatives denotes the number of scripts that the model did not manage to predict, which is something we want to avoid. The model has managed to achieve this as can be seen in Table 4.3, where the model failed to predict the test scripts only for two test case specifications. This affects the Recall score since the lower the number of false negatives is, the higher Recall score is. However, similar to Precision, Recall alone is not sufficient to determine the performance of the approach. If the model were to suggest all of the test scripts, then the model would achieve a perfect Recall score. This is also not a desired property of the model. Therefore, F1 score was also calculated as a harmonic mean of Precision and Recall to give a more balanced measure of quality of the approach. Moreover, all the performance measurements used in this study are affected by the number of true positives, i.e. number of test scripts that were correctly predicted by the model. The number of true positives is considered to be the most important factor when evaluating the quality of the model since it is highly important that the model manages to predict the correct test scripts for a specific test case specification. With all this information in mind, it is possible to say that using only one of these metrics to determine the approach's performance is not enough. By combining and analyzing all of them, one can acquire more useful insights about the approach.

Two classifiers, LinearSVC and KNN, were used during the experiments and their results were compared. According to the results in Tables 4.1 and 4.2, LinearSVC outperforms KNN and the highest F1 score is achieved using the LinearSVC classifier. One reason for why LinearSVC outperformed KNN is the large number of features in the data, i.e. the high dimensionality. With large numbers of features, KNN tends to have a deterioration in the performance. As described in Section 2.4.1.2, KNN only uses observations that are near the test observation, for which a prediction must be made, to make the prediction. In a high dimensional space, the K observations that are nearest to the given test observation may be far away from it, resulting in a poor KNN fit. On the other hand, LinearSVC tends to perform well with a limited set of points in a high dimensional space since it finds the decision boundary, to separate the data points, by only using the most relevant data points, i.e. the support vectors, as described in Section 2.4.1.1. For these reasons, LinearSVC outperforms KNN when the number of features is large and the size of the training data set is small, which exactly describes the data in this study.

For most machine learning models, more data can lead to better results since the availability of data allow for more and better insights. In this study, the data set that was provided was small and collecting more data was not feasible. To combat this issue, data augmentation was used to increase the size and diversity of the available data for training the model. Performing data augmentation has affected the results of the approach and the differences in the results can be seen in Tables 4.1 and 4.2. For LinearSVC, better results were achieved when performing data augmentation on the training set. The F1 score for LinearSVC increased from 81.6%, without data augmentation, to 89.3%, with data augmentation. For KNN, the performance improvement was not as clear since the results of 1-NN were slightly better without data augmentation. One possible reason for this is that the extra generated data points, from data augmentation, may clutter the vector space, causing wrong data points to be closer to the test observation. Furthermore, in Table 4.1, it can be seen that the exact same results were achieved for 1-NN and 3-NN when using data augmentation. This is due to the consequences of data augmentation, where new data points are generated by slightly modifying existing data points. These new data points will increase the density in the region in which the data point that they were modeled from resides. It is important to have in mind that these newly generated data points will have the same label as the data point they were modeled from. The KNN classifier makes predictions by finding the *K* closest neighbors to the given test observation. In the high density regions created as a result of data augmentation, there will be a small variance in the distances between the test observation and the other data points in that region. The similar distances and the fact that the data points in that region have the same label, makes the value of K less significant. It is likely that different values of K will lead to the same results if data augmentation techniques are used.

5.2 Threats to Validity

The threats to validity, limitations and challenges faced in conducting this study are discussed in this section. The validity of a study indicates the trustworthiness of its results and it should be considered during all phases of the study. Runeson and Höst [57] describe four aspects of validity, which are: Construct validity, Internal validity, External validity, and Reliability. These validity aspects and the threats to them are discussed in this section from the perspective of this study.

The aspect of construct validity addresses the relation between theory and observation [58]. This aspect concerns whether the measurements that are studied really represent what the researcher intended to investigate [57]. One possible construct validity threat in the present study is the source used for generating the test cases. In this study, only test case specifications are used in order to generate the test cases. It is possible to use other types of specifications written in natural language, such as Software Requirement Specifications (SRS) or Use Case Specifications. Combining and analyzing different specification documents to generate test cases may yield better results or give a more accurate picture of the intended research. However, other natural language-based specifications were not available for this testing project and deriving them would have been very time-consuming. Furthermore, the aspect of construct validity has been taken into account during the course of this project. For example, it was concluded that accuracy would be a poor measurement of the approach's performance, i.e. it would have poor construct validity. Therefore, other more suitable performance metrics were selected, such as balanced accuracy and F1 score, leading to a better construct validity.

Internal validity is the aspect of validity that concerns the conclusions of the study [57]. The main threat to internal validity in this study is the dimensionality of the data set as it was small and had a low frequency of labels. The data set was divided into training and testing sets only once and that was done manually to ensure that all labels were present in the training set, as described in Section 3.5.2. It was not possible to obtain a standard deviation of the performance measurements since the performance measurements were obtained for only one division of the data. This might suggest that the approach is dependent on the division of the data and might perform differently for different data sets. Another potential threat to internal validity is the structure of the test case specifications. In this project, NLP techniques were performed on a set of semi-structured test case specifications where NLTK and Stanford POS-tagger were used to parse and analyze the documents in an efficient amount of time. There is no guarantee that for more complicated structures of test case specifications, NLTK and Stanford POS-tagger will perform similarly, which may impact the results of the approach.

External validity addresses the ability of generalizing the findings [57, 58]. The proposed approach has been applied on a limited number of test case specifications in only one industrial testing project. Despite that, the approach's findings are relevant for other cases since the approach is applicable to other contexts with common characteristics. Also, the approach can be extended to other testing domains and testing levels, for example unit, regression and system testing levels. To allow for comparison to other approaches in the testing domain, all necessary context information has been provided in this report.

Lastly, the reliability of the study concerns the extent to which the data and the analysis are dependent on the specific researcher. Runeson and Höst [57] mean that if this study were to be conducted by a different researcher, the outcome should be the same. To avoid threats to reliability as much as possible and to better allow replication of the study, the methodology of the approach is described in detail in Chapter 3. However, there exist some issues in this study that could pose as threats to reliability. The major reliability threat in this study is the way the features are extracted from the test case specifications. As described in Section 3.4.1, the test case specifications were first observed and examined to determine which features are relevant to extract. This makes the feature extraction process somewhat subjective and specific for this testing domain, which may influence the outcome of the study. The ten most frequent features shown in Figure 4.2 would probably be different if other conditions were used for extracting the features. Furthermore, since the test case specifications are written in natural language by a group of testers, these specifications can suffer from spelling issues. If an important keyword in the specification is misspelled, one or several times, the approach will not be able to select this keyword as a feature for the test case specification. This issue can directly impact the results of the study since the feature extraction stage is an essential part of the whole approach.

5.3 Ethics and Sustainability

Whenever conducting research and developing products, it is important to consider the work's ethical impact and also review the sustainability aspects of it. This is important since even if the work is conducted with good intentions, it can still have a negative impact on society.

This study involves optimizing the software testing process. Software testing is a part of any software system's life cycle. A software system is applicable in many different domains and can have an impact on all aspects of society.

A developed system and its products can therefore affect all three components of sustainability: social, environmental and economic. Therefore, it is always important to keep sustainability in mind when developing new products. Optimizing the software testing process will in turn optimize the whole software, since testing is a large part of its life cycle. Ensuring higher quality products and faster deliveries will amplify the software's effect on sustainability. How sustainability is affected and to what degree depends on the domain of the system under development. For example, if a software system has a positive affect on the environment, then optimizing and improving the testing process of that system will increase its positive environmental effect. In this study, the products used are part of the telecommunication domain where the host company is trying to develop sustainable artificial intelligence solutions that scale globally.

Vinuesa et al. [59] state that artificial intelligence has a great effect on the achievement of the Sustainable Development Goals (SDG) since artificial intelligence has potential to enable the accomplishment of 134 (79%) targets across all the goals. However, artificial intelligence may present some limitations in the achievement of 59 targets [59]. The work in this degree project make use of artificial intelligence to help in automating the software testing process which leads to reduction in costs and resources. From an economic perspective, the automation of software testing will benefit the company since it can potentially lead to cutting costs in the long run. This will increase the revenue of the company and also increase the possibility of enhancing the product's quality. However, from a social point of view, automating different tasks in several domains can lead to reduced need of workers. Automation can replace people and thus lead to increased unemployment in society. Nevertheless, this is not always the case in the testing domain since artificial intelligence may replace the human work but not the humans themselves. Experienced and educated people with domain knowledge are always needed to design, implement, and validate the results of such automation algorithms and methods. Moreover, the results of this thesis will contribute towards test engineers spending more time on more valuable and rewarding tasks, rather than repetitive jobs. The goal of the approach is to aid the testers in their work rather than replace them in any way.

From an ethical point of view, one has to consider the extent to which such automation algorithms and approaches should be trusted. In some scenarios, where vital situations can be involved, a fully automated testing process could raise ethical issues. The tool implemented in this degree project is intended to act as a supplement for the test engineers. However, this approach can be developed further to an extent where test engineers might fully rely on it without assessing the generated test cases. This is problematic in situations where human lives are directly or indirectly involved, for example in self-driving cars. Therefore, human intuition and inductive reasoning cannot be fully replaced and is required to ensure safeness of the developed system.

5.4 Future Work

Although the evaluation of the proposed approach shows promising results, there is room for improvement. In this section, possibilities for future work and improvements of this study are explored and discussed.

In the related work, described in Section 2.5, the strategy implemented by Carvalho et al. [9] is evaluated in four different domains, including the turn indicator of Mercedes vehicles. This ensures the strategy's capability to perform well in different domains. An interesting future extension of the proposed approach in this study would be to test it in different domains to get a more complete picture of the approach's abilities. Furthermore, Figure 4.3 shows that the approach's performance improves as the size of the training set increases. This indicates that the approach has potential in handling larger data sets efficiently. Therefore, another possible extension could be to apply the proposed approach on more products, i.e. on more data, to investigate the scalability of the approach. However, if a larger data set is used for this study, an important thing to consider is that as the amount of data grows, so will the number of different features and labels. One might think that as the number of features used to fit a model increases, the quality of the fitted model will increase as well, but this is not necessarily true. The model may perform worse when the dimensionality of the problem increases. This phenomenon is referred to as the *curse of dimensionality* [43]. To avoid this problem, different techniques can be used for dimensionality reduction such as Principal Component Analysis (PCA). One should research what techniques are suitable for this specific study's problem.

Furthermore, there exists possibilities for improvements in the methodology of the proposed approach. For example, the multi-label classification in this study was done by using the OVA strategy for simplicity purposes. Transforming the multi-label classification problem to a single-label problem, by using the OVA classifier, enables the usage of traditional well-known already existing single-label algorithms. Other strategies for performing multi-label classification include adapted algorithms which concentrate on modifying a single-label classifier to directly perform multi-label classification, rather than transforming the problem into different subsets of problems [60]. The OVA strategy treats each label individually, as it generates one classifier for each label, and thus ignore any possible correlations among the labels. One the other hand, the adapted algorithms can capture some correlations between the labels, which may influence the results of the approach. Therefore, it would be interesting to run the proposed approach with such algorithms to investigative how it affects the outcome of the approach.

The work of this degree project can be seen as an important first step in the pursuit of a fully automated test case generation process. Currently, the proposed approach produces file names of already existing C# test scripts as output given an unseen test case specification document, as can be seen in Figure 4.4. This proposed approach works with categorical labels as output using only the file names as identifiers of the test scripts. A future direction of this study could be to investigative the possibility of producing totally new test scripts containing unseen code for executing the test cases. In that case, the model must train with a different output space than that used in this study. A suitable output could be the building blocks of the test scripts such as the code features. Such an approach would be more flexible and generalized in dealing with unseen test case specification documents.

Chapter 6

Conclusions

Automatic test case generation is one way of optimizing the software testing process, which can help in minimizing the required time and effort for testers. In this thesis, an approach for automatic test case generation from natural language test case specifications has been designed, applied, and evaluated. The effectively implemented approach consists of a pipeline of three stages. The first stage of the approach involves parsing and analyzing the test case specification documents using different NLP techniques. Feature vectors, containing keywords representing the specification are generated during this stage. In the next stage, the feature vectors are mapped to label vectors consisting of C# test scripts file names. The feature and label vectors are used as input and output, respectively, in the last stage in the text classification process. Finally, the proposed approach is applied and evaluated on an industrial testing project at Ericsson AB in Sweden and the approach has been implemented as a tool for aiding testers in the process of test case generation.

Regarding the research questions of this thesis, it is concluded that this degree project has shown how NLP techniques can be incorporated in the test case generation process and that the proposed approach has achieved promising results. The results of evaluating the approach showed that the proposed approach has managed to produce relevant test scripts for the selected test case specifications, achieving high accuracy and F1 scores. The best F1 score of 89.3% is achieved when running with the OVA strategy with LinearSVC classifier and performing data augmentation on the training set. Furthermore, it was clear that LinearSVC outperformed the KNN classifier given the data set of this project. It was discussed that a possible reason for this is the properties of the data in this project. The data contains a large number of features, i.e. the data is high dimensional, and the training set is small, which are conditions where LinearSVC outperforms KNN.

Future research of this study may include applying the proposed approach on more products, i.e. more data, and evaluating it in different testing domains to get more helpful insights about the approach. Another potential future direction of this study is to investigative the possibility of producing totally new test scripts containing unseen code for executing the test cases. Such extension of the approach would increase the approach's ability in dealing with unseen test case specification documents.

To conclude this thesis, it is evident that there is potential in using NLP techniques on natural language test case specifications to automatically generate test cases. The proposed approach has managed to suggest relevant test scripts for the selected test case specifications. Furthermore, the approach has potential in replacing manual testing since the tool, which implements the approach, can aid the testers by relieving some of the manual operations performed on test case specifications.

Bibliography

[1] S. Tahvili. “Multi-Criteria Optimization of System Integration Testing”. PhD thesis. Malardalen University, Dec. 2018. ISBN: 978-91-7485-414-5.

[2] S. Tahvili et al. “Cost-Benefit Analysis of Using Dependency Knowledge at Integration Testing”. In: *The 17th International Conference On Product-Focused Software Process Improvement*. Nov. 2016.

[3] R. P. Verma and M. R. Beg. “Generation of test cases from software requirements using natural language processing”. In: *2013 6th International Conference on Emerging Trends in Engineering and Technology*. IEEE. 2013, pp. 140–147.

[4] S. Tahvili et al. “sOrTES: A Supportive Tool for Stochastic Scheduling of Manual Integration Test Cases”. In: *Journal of IEEE Access* 6 (Jan. 2019), pp. 1–19.

[5] A. Isabella and E. Retna. “Study paper on test case generation for GUI based testing”. In: *arXiv preprint arXiv:1202.4527* (2012).

[6] A. Singh and S. Sharma. “Automated Generation of Functional Test Cases and Use Case Diagram using SRS Analysis”. In: *International Journal of Computer Applications* (2015).

[7] M. Prasanna et al. “A survey on automatic test case generation”. In: *Academic Open Internet Journal* 15.6 (2005).

[8] I. Ahsan et al. “A comprehensive investigation of natural language processing techniques and tools to generate automated test cases”. In: *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*. 2017, pp. 1–10.

[9] G. Carvalho et al. “NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications”. In: *Science of Computer Programming* 95 (2014), pp. 275–297. Elsevier.

[10] R. Boddu et al. "RETNA: from requirements to testing in a natural way". In: *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.* IEEE. 2004, pp. 262–271.

[11] C. Wang et al. "Automatic Generation of System Test Cases from Use Case Specifications: an NLP-based Approach". In: *arXiv preprint arXiv:1907.08490* (2019).

[12] R. Singh, A. Singhrova, and R. Bhatia. "Test Case Generation Tools-A Review". In: *International Journal of Electronics Engineering (ISSN: 0973-7383)* (2018).

[13] "ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions". In: *ISO/IEC/IEEE 29119-1:2013(E)* (2013), pp. 1–64.

[14] M. Ellims, J. Bridges, and D. C Ince. "The economics of unit testing". In: *Empirical Software Engineering* 11.1 (2006), pp. 5–31. Springer.

[15] M. A. Ould and C. Unwin. *Testing in software development*. 1986. Cambridge University Press.

[16] D. Di Nardo et al. "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system". In: *Software Testing, Verification and Reliability* 25.4 (2015), pp. 371–396. Wiley Online Library.

[17] S. Tahvili et al. "Dynamic integration test selection based on test case dependencies". In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2016, pp. 277–286.

[18] S. Tahvili et al. "Towards Earlier Fault Detection by Value-Driven Prioritization of Test Cases Using Fuzzy TOPSIS". In: *13th International Conference on Information Technology : New Generations (ITNG 2016)*. Apr. 2016.

[19] J. A Jones and M. J. Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage". In: *IEEE Transactions on software Engineering* 29.3 (2003), pp. 195–209. IEEE.

[20] A. Vahabzadeh, A. Stocco, and A. Mesbah. "Fine-grained test minimization". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 210–221.

[21] H. Hourani, A. Hammad, and M. Lafi. “The Impact of Artificial Intelligence on Software Testing”. In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. IEEE. 2019, pp. 565–570.

[22] IEEE Standards Coordinating Committee et al. “IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)”. In: *IEEE Computer Society, Los Alamitos. CA* (1990).

[23] S. Tahvili et al. “ESPRET: A tool for execution time estimation of manual test cases”. In: *Journal of Systems and Software* 146 (2018), pp. 26–41. Elsevier.

[24] D. Richardson, O. O’Malley, and C. Tittle. “Approaches to specification-based testing”. In: *Proceedings of the ACM SIGSOFT’89 third symposium on Software testing, analysis, and verification*. 1989, pp. 86–96.

[25] L. Apfelbaum and J. Doyle. “Model based testing”. In: *Software quality week conference*. 1997, pp. 296–300.

[26] K. Johannisson. *Formal and informal software specifications*. 2005. Citeseer.

[27] S. Tahvili et al. “Cluster-based test scheduling strategies using semantic relationships between test specifications”. In: *Proceedings of the 5th International Workshop on Requirements Engineering and Testing*. 2018, pp. 1–4.

[28] C. Landin et al. “Cluster-Based Parallel Testing Using Semantic Analysis”. In: *The Second IEEE International Conference On Artificial Intelligence Testing*. Apr. 2020.

[29] E. D Liddy. “Natural language processing”. In: *Encyclopedia of Library and Information Science, 2nd Ed* (2001). NY. Marcel Decker, Inc.

[30] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. 2009. " O’Reilly Media, Inc."

[31] G. G Chowdhury. “Natural language processing”. In: *Annual review of information science and technology* 37.1 (2003), pp. 51–89. Wiley Online Library.

[32] P. Semaan. “Natural language generation: an overview”. In: *Journal of Computer Science & Research (JCSCR)-ISSN* (2012), pp. 50–57.

[33] R. V Yampolskiy. “Turing test as a defining feature of AI-completeness”. In: *Artificial intelligence, evolutionary computing and metaheuristics*. 2013, pp. 3–17. Springer.

[34] D. D Palmer. “Tokenisation and sentence segmentation”. In: *Handbook of natural language processing* (2000), pp. 11–35. New York: Marcel Dekker, Inc.

[35] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*. 2011. Cambridge University Press.

[36] W. Khan et al. “A survey on the state-of-the-art machine learning models in the context of NLP”. In: *Kuwait journal of Science* 43.4 (2016).

[37] D. Jurafsky. *Speech & language processing*. 2000. Pearson Education India.

[38] A. Daud, W. Khan, and D. Che. “Urdu language processing: a survey”. In: *Artificial Intelligence Review* 47.3 (2017), pp. 279–311. Springer.

[39] J. Eisenstein. *Natural language processing*. 2018. Cambridge, MA: The MIT Press.

[40] J. Read et al. “Classifier chains for multi-label classification”. In: *Machine learning* 85.3 (2011), p. 333. Springer.

[41] L. Tang, S. Rajan, and V. K. Narayanan. “Large scale multi-label classification via metalabeler”. In: *Proceedings of the 18th international conference on World wide web*. 2009, pp. 211–220.

[42] R. Rifkin and A. Klautau. “In defense of one-vs-all classification”. In: *Journal of machine learning research* 5.Jan (2004), pp. 101–141.

[43] G. James et al. *An introduction to statistical learning*. Vol. 112. 2013. Springer.

[44] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[45] A. Zheng and A. Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. 2018. " O’Reilly Media, Inc."

[46] J. W. Wei and K. Zou. “Eda: Easy data augmentation techniques for boosting performance on text classification tasks”. In: *arXiv preprint arXiv:1901.11196* (2019).

[47] S. Kobayashi. “Contextual augmentation: Data augmentation by words with paradigmatic relations”. In: *arXiv preprint arXiv:1805.06201* (2018).

[48] H. Hemmati and F. Sharifi. “Investigating nlp-based approaches for predicting manual test case failure”. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 309–319.

[49] V. A. de Santiago Junior and N. L. Vijaykumar. “Generating model-based test cases from natural language requirements for space application software”. In: *Software Quality Journal* 20.1 (2012), pp. 77–143. Springer.

[50] J. W. Shipman. “Tkinter 8.4 reference: a GUI for Python”. In: *New Mexico Tech Computer Center* (2013).

[51] K. Toutanova et al. “Stanford log-linear part-of-speech tagger”. In: *The Stanford Natural Language Processing Group, Stanford University Std.* (2000).

[52] F. Lundh. “An introduction to tkinter”. In: *URL: www. pythonware. com/library/tkinter/introduction/index. htm* (1999).

[53] T. Sørensen et al. “A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons”. In: (1948).

[54] L. R. Dice. “Measures of the amount of ecologic association between species”. In: *Ecology* 26.3 (1945), pp. 297–302. Wiley Online Library.

[55] S. Tahvili et al. “Automated Functional Dependency Detection Between Test Cases Using Doc2Vec and Clustering”. In: *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE. 2019, pp. 19–26.

[56] K. H. Brodersen et al. “The balanced accuracy and its posterior distribution”. In: *2010 20th International Conference on Pattern Recognition*. IEEE. 2010, pp. 3121–3124.

[57] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (Dec. 2008), p. 131. ISSN: 1573-7616.

[58] C. Wohlin et al. *Experimentation in software engineering*. 2012. Springer Science & Business Media.

[59] R. Vinuesa et al. “The role of artificial intelligence in achieving the Sustainable Development Goals”. In: *Nature Communications* 11.1 (2020), pp. 1–10. Nature Publishing Group.

[60] ML. Zhang and ZH. Zhou. “A review on multi-label learning algorithms”. In: *IEEE transactions on knowledge and data engineering* 26.8 (2013), pp. 1819–1837. IEEE.

TRITA -EECS-EX-2020:592