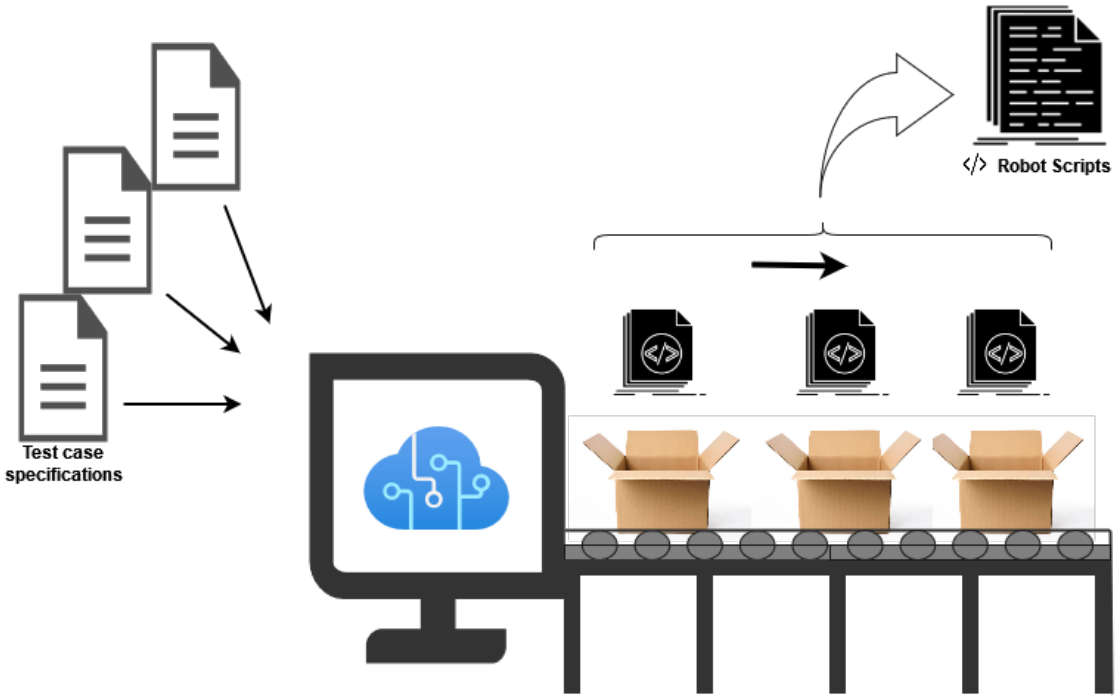
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Automatic test code generation from acceptance test cases for large-scale software products

Master's thesis in Computer Science and Engineering

KUVALAYA DATTA JUKANTI

PRASHANT KUMAR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Automatic test code generation from acceptance test cases for large-scale software products

KUVALAYA DATTA JUKANTI

PRASHANT KUMAR

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Automatic test code generation from acceptance test cases for large-scale software productsKUVALAYA DATTA JUKANTI
PRASHANT KUMAR

Supervisor: Krasimir Angelov, Department of Computer Science & Engineering
Advisor: Celso Freitas, Ericsson AB
Examiner: Regina Hebig, Department of Computer Science & Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A creative picture of the process of test case descriptions translating into test code using NLP and AI.

Typeset in LaTeX
Gothenburg, Sweden 2022

Automatic test code generation from acceptance test cases for large-scale software products

KUVALAYA DATTA JUKANTI
PRASHANT KUMAR
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The process of evaluating whether or not a software system is conformed to the requirement specifications plays an important role in a large-scale software environment. The test engineers identify the test scenarios and implement the code according to the software standards which enables the verification of required criteria for the product delivery. In the development model adopted by several companies, including Ericsson, the requirements for a product are defined first on a high level. These requirements are analysed and transformed into test plans or test cases which are tested after the product development stage. But today, Ericsson has adopted a new practice of software development which is the test-driven development process where the test cases are defined before the software is fully developed. The test engineers then write executable code for the test cases and test the product for different scenarios.

However, in the context of large-scale software development with multiple products and teams, it is not trivial to implement code for all the test cases. Today, the common way of performing acceptance testing is through manual labour by understanding the requirements, implementing the test scripts and execute them. The developers create tests to determine if the requirements are met with the contract. This serves as one of the challenges as it is a time consuming process. Thereby, it is important to automate the testing process to rely less on the test engineers and sufficiently serve the purpose. Today there are more advanced technologies and resources which could help in developing such a tool that can reduce the costs and time for the company.

This project has as its goal to automate the process of translating the test description to test code. The approach involves the use of natural language processing techniques and other deep learning techniques for analyzing the the test case specification written in natural language and generate the corresponding executable test code. This test code follows the syntax of Robot Test Automation Framework. However, as every test scenario involves different parameters to consider, the aim is to generate the code with suitable functions and be user-friendly allowing human experts' adjustments to add configurations and parameters.

Keywords: *Acceptance testing, Test case specifications, Automatic test code generation, Natural Language Processing, Deep Learning, Robot Framework.*

Acknowledgements

There are many people who helped us directly and indirectly to complete this project successfully.

Firstly, we would like to thank Ericsson for giving us the opportunity to work on this challenging and interesting thesis project. We express our sincere gratitude towards our supervisor Celso Freitas at Ericsson for guiding us and providing us with his valuable feedback throughout the thesis project. We would also like to extend our gratitude to our supervisor at Chalmers University of Technology, Krasimir Angelov for being so understanding and giving instructions during the thesis. We would also like to thank our examiner Regina Hebig at Department of Software Engineering for her support through the thesis.

Lastly, we would like to thank all our friends for their help, constructive criticism, moral support and encouragement during the project period.

Kuvalaya Datta Jukanti
Prashant Kumar
Gothenburg, June 2022

Contents

List of Figures

List of Abbreviations

AI Artificial Intelligence. 4, 8
ANN Artificial Neural Network. 8

BLEU Bilingual evaluation understudy. ix, 12, 16, 17, 41–43, 45, 49

CE Cross entropy. 14

DNN Deep Neural Networks. 11
DOBF Deobfuscation. 15

GPU Graphics Processing Unit. 29, 42
GRU Gated Recurrent Unit. ix, 10, 28, 37, 39, 41, 43–47, 49

LSTM Long Short-Term Memory. ix, x, 9, 12, 19, 28, 31–34, 37, 39, 41, 43, 44, 46

ML Machine Learning. 4, 8
MLM Masked Language Modelling. 15

NLP Natural Language Processing. ix, 2–4, 7, 8, 13, 18, 21, 22, 24, 49, 50
NLTK Natural Language Toolkit. 24–26

POS Part-of-Speech. 8

RAM Random Access Memory. 29
RNN Recurrent Neural Networks. ix, 8, 9, 28
RQ Research Question. 42, 49, 50

seq2seq Sequence-to-Sequence. 19

TDD Test-Driven Development. xi, 1, 2

1
Introduction

Software testing is the process of evaluating or verifying that a software product or application meets the expected requirements. It is considered the most critical stage of the software development life cycle as it not only ensures that the software system is error or bug free but also ensures high quality of the software system [34]. Over the past few years, the software products have grown in size and complexity making the entire process expensive in terms of cost, time and efforts.

The testing process involves a group of people analyzing the specification document, identifying the test cases and then implementing code. The requirement specification document describes what the software is intended to do and how it will be expected to perform. The software testers analyze these steps or functions and develop the corresponding code to test that the software is performing according to the expectations. In the past two decades, there have been many advances in the software development approaches. The evolution of Agile development has introduced many realistic and efficient approaches moving towards continuous delivery and high speed to customers or clients. In recent years, one such practice that has grown in popularity and has been recognized as one of the efficient approaches in the agile software development niche is test-driven development (TDD). With TDD, developers create tests that validate the functionality of the software. In simple words, developers write the test before developing the actual code. This is mainly used for acceptance testing. Acceptance Testing is a process of evaluating a system's compliance with business requirements and determining whether it is acceptable for delivery.

The major difference between the traditional testing approach and the TDD is that in the traditional approach, the code is written first, and the tests are executed at the end. TDD promotes the red-green-refactor approach. *Red* refers to the details that any feature should start with a failure of test case, *green* denotes that the code to be implemented and get the new test case pass and *refactor* signifies that the code should be cleaned up [26]. In other words, it is based on the idea that the developers use the test cases before writing the actual code. Figure 1.1 shows the same. This helps in building quality software as it helps developers understand the requirements, and with the constant feedback there is more test coverage and verification that the software works as intended.

Today, the most common method of testing is through manual labour. In today's world, the concept and nature of the software testing has changed. The industry has equipped new approaches or techniques with regard to software testing, specifically

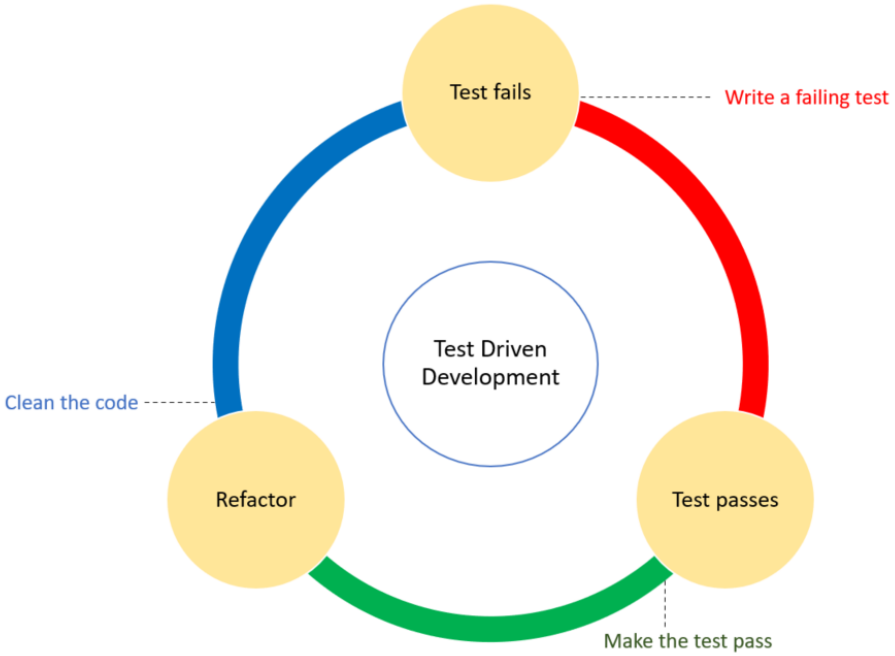


Figure 1.1: Test Driven Development (TDD) process illustrating the *red-green-refactor* approach

the changes pertaining to the testing procedure. Having said that, there is a continuous transition from manual to automated testing [37]. Test automation improves software testings' effectiveness, efficiency, and coverage. However, human intuition and deduction is not fully achieved with the current methods of test automation and therefore manual software testing is still considered as an important approach in software testing [7].

Manual software testing is a resource-intensive and time-consuming procedure, despite being one of the most common techniques of testing. As the software products grow in size, the costs and efforts also increase which makes the software testing process very expensive. Automating the process of test code generation eliminates or reduces the manual efforts, thereby aiding to substantial time and financial savings for the organizations.

One of the key sources of information for developing test cases is specifications.Because test case requirements must be simple to use and comprehend, they are frequently written in natural languages like English and hence using Natural Language Processing (NLP) techniques for understanding the test cases serves an appropriate methodology [34]. NLP approaches have demonstrated promising outcomes previously when it comes to understanding raw data pertaining to a specific area or domain. This sparked a growing interest in natural language processing approaches in a wide variety of applications, for example, machine translation, text generation, information extraction, etc. On the other hand, code generation has become an important area to predict code with structure and syntax. Deep learning models are

emerging technologies in this area to achieve the difficult code processing problems such as code generation and code summarization. The combination of NLP and deep learning will yield in analyzing the test cases and generate the corresponding test code in the current study. Several previous studies on the generation of code from text have been conducted [31, 21, 11, 28]. However, most of these solutions are implemented on general-purpose programming languages where there is abundance of data available which makes the models learn the structure and syntax better. Such languages would not be available in a domain-specific environment especially when it comes to writing test code, and if available, they require a lot of data and expert guidance in understanding the structure and syntax of them.

This thesis proposes, implements, and evaluates a method for generating test code from test case requirements or specifications expressed in natural language. The solution involves the generation of test scripts in Robot Test Automation framework (a keyword-based testing framework explained in section 2.9). This is implemented by using deep learning and NLP techniques. The proposed approach is used and tested on an existing industrial project at Ericsson AB.

1.1 Background

This thesis is done in collaboration with Ericsson AB, with the sub unit that develops real-time telecom software systems. Their software development team handles some of the major applications developed till date. Evidently, these application have huge repositories of code which contain thousands of features and functionalities. In the software development department, there are two versions of applications that are of importance for this project. First, a monolithic application which is structured in the form of a single-tiered architecture, consists of all the code composed in one piece. This version of application is old and has tests written using Robot Test Automation Framework. Having said that, there are no corresponding test case descriptions through which the executables were implemented.

There is newer version which ought to replace the monolithic application which is based on test-driven development and consists of acceptance test cases and no executables. To elaborate, the newer version only has the test descriptions whereas the robot test code is not yet implemented. Ericsson's idea is to make use of the monolithic application, understand the syntax and structure of the robot scripts and use it to generate the same for the new acceptance test cases. That is, given a test case specification in natural language, generate the corresponding robot test code with the involvement of human interpretation.

1.2 Purpose

The purpose of this study is to investigate the possibility of building a system or a model that (a) translates the test case specifications written in natural language

into robot test code by incorporating NLP and deep learning techniques; (b) be user-friendly and allow human experts' adjustments to add configurations and parameters. The proposed solution capitalizes the existing tools / technologies such as NLP and deep learning to generate the code in the syntax of Robot framework. The study is being carried out by Ericsson AB, who are interested in using Artificial Intelligence (AI) and Machine Learning (ML) in a variety of areas. Software testing is one such arena where AI and ML can be utilized to reduce the time and efforts put in by the developer teams. It will also facilitate easier maintenance and reuse of test cases if there are any changes in the test case scenarios. This study's findings could pave the way for minimizing the costs for software testing department in the company.

1.3 Problem Definition

The main objective of this thesis study is to generate code for a particular test description written in natural language for large-scale software products. As a result, the following research questions arise:

RQ1. How can the test cases of existing applications be leveraged to prepare data and use it to generate code for other applications?

RQ2. How can test case specifications written in natural language be translated to test code in robot framework syntax for large-scale software products?

RQ3. How does the quantity of the data used for training affect the generation of accurate code results?

RQ4. How can the obtained results be useful for the developers at Ericsson?

1.4 Scope

There exist several ways one can write the test case specifications. These descriptions can be both abstract and detailed. In this project, the test case specification is viewed as a series of steps, each of which corresponds to a small module in an entire block of code. Furthermore, in this investigation, the generated code will not be executed in its entirety; instead, a human expert will intervene and adjust the settings and parameters. To elaborate, since the model will be trained on the monolithic application (see section 1.1), it cannot produce the new parameters pertaining to the unseen test cases. In such cases, a human expert will have to manually alter the parameter values in order to execute it correctly. As for the expectations set for this project, the model is expected to understand the test case description and generate the keywords (section 2.9) required in the test code.

1.5 Limitations

1. **Lack of data** - The approach for this study involves the usage of deep learning models, which often requires a large amount of data to produce good results. The data provided by Ericsson is considerably not in large amounts. In addition, the new version, which is a test-driven application, contains acceptance test cases, which are far less in number.

2. **Extraction of data from different applications** - The test driven application which consists of acceptance test cases do not have any corresponding robot files implemented. Therefore, the data from the monolith version which contains the robot files is used. Having said that, the limitation is that the two versions are very different from one another, creating a risk of obtaining sub-optimal results.

3. **Human intervention** - The generated code script will consist of parameters, variables, and keywords which act as functions in Robot Framework. However, in order for the test file to run, human intervention is required to update the settings and parameters. As a result, the created file cannot be executed directly.

4. **Evaluation** - The generated result is evaluated by testing and domain experts from the company. As a result, the scoring would be determined by the individual's viewpoint. Different people would have different opinions on which to base their scores.

1.6 Thesis Outline

The report has a total of five chapters. *Chapter 1* offers an introduction to the subject, as well as a summary of the problem statement, the study's purpose, and its scope. The related work and theory for the topic is provided in *Chapter 2*. It covers various approaches dealing with code generation and the procedures or techniques followed in this study. It also explains the deep learning and natural language processing approaches which are applicable for our study. *Chapter 3* gives an outline of the methodology and implementation of the proposed approach. *Chapter 4* contains the study's findings and outcomes, along with drawing the reasons for the obtained results. *Chapter 5* includes the conclusion for the thesis.

2

Theory

This chapter describes the overview of the information required for this project. The initial couple of sections gives an overview of NLP and Deep Machine Learning. The next few sections focus on covering the theoretical concepts relevant to the thesis. Section 2.3 focuses on explaining the recurrent neural networks and its variants. Section 2.4 describes the concept sequence-to-sequence learning which is the fundamental idea for this project study. Section 2.5 describes the use of attention mechanism in deep learning models. Section 2.8 explains code deobfuscation, a technique used to mask elements in source code and recover them to their original state after training the model. The next section gives a brief idea about the Robot Framework and the benefits of using it. Section 2.10 explains the evaluation method suited for the current problem. The chapter is closed with a section of information providing the insights gained from previous related work in the area of NLP and deep learning for code generation.

2.1 Natural Language Processing (NLP)

Natural language is a language that humans use in everyday communication. It can be ambiguous because of its huge and diversified vocabulary, words with many meanings, and be spoken in a variety of accents. NLP is a field that combines computer science, artificial intelligence, and linguistics to investigate how computers can understand and process natural language text or speech [8]. The growth and research in the area of NLP has been growing rapidly, however, it is still considered a difficult process since computer interaction requires a precise and clear language, which is not the case with natural language.

Typically, a NLP system is made up of many processing stages. Lower levels include morphological analysis, syntactic analysis, and semantic mapping, whereas higher levels include discourse and pragmatic analysis [6, 34]. The majority of today's NLP systems are focused on lower-level processing probably because the lower levels focus on smaller elements, such as words and phrases. There are different kinds of steps or techniques involved when dealing with text data. It is a wide area to describe and it often depends on the type of problem dealt with and therefore, the required NLP techniques used in this project is described in section 3.3.

2.2 Deep Machine Learning

Living amidst of the big data era, in which vast volumes of data are generated across all disciplines of science and industry, presents society with unparalleled hurdles in analyzing and interpreting them. This allowed deep learning to be a new learning paradigm in the world of AI and ML [14]. It is a methodology based on the concept of Artificial Neural Network (ANN) that has been gaining a lot of attention as it is producing quality results [18]. Recent breakthroughs in the areas of image analysis, speech recognition and Natural Language Processing have sparked widespread interest in this topic. Since it refers to a group of techniques rather than a single method for learning large prediction models, it also appears that applications in a variety of other domains are feasible. In the context of natural language processing tasks, there have been immense positive outcomes in various problems such as text categorization, POS tagging, document classification, and others. NLP tasks require a very robust feature engineering which can be both time consuming and cost intensive. Deep learning techniques are extremely useful in those complex situations when domain expertise is scarce. There are a variety of representations accessible, but each is best suited to a given goal.

2.3 Recurrent Neural Networks (RNN)

A recurrent neural network is a type of Artificial Neural Network that can learn the sequential features of data and use patterns to forecast the next likely occurrence. Because most natural language problems comprises a sequence of characters, phrases or sentences with correlated elements, using RNN to process sequences makes it relevant to these type of problems. Furthermore, the key feature of RNN is the ability to use its internal memory to store past information. Like other neural networks, recurrent neural networks are divided into layers, with each layer consisting of interconnected nodes having an activation function. They can process sequences by the fact that the current state is affected by its previous states [31].

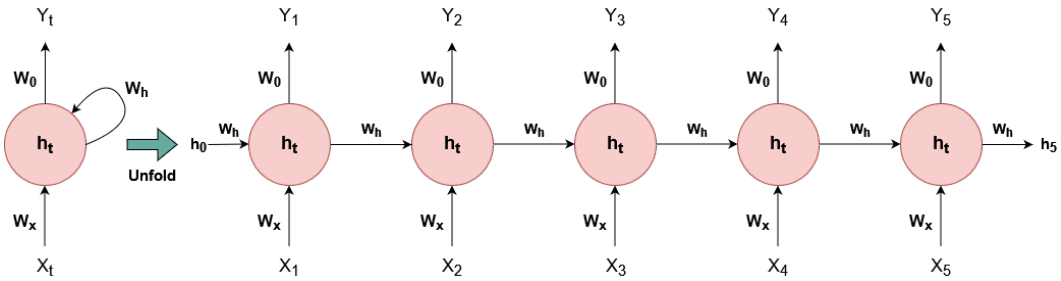


Figure 2.1: Unrolled Recurrent Neural Network

While dealing with text, the network needs to process the previous word or character in the sequence. RNNs perform the same task for every element in the sequence which gives them the ability to remember what has been done previously. Figure 2.1 demonstrates the simple structure of RNN network, where x_t is input at each time

step t, h_t is hidden state at time t, W_x and W_h are weights of input and recurrent neuron respectively, and y_t is output at time step t. It can be seen that, at each time step, the weights and biases are constant to every input and hidden state allowing RNN to handle inputs of varying lengths.

2.3.1 Long Short-Term Memory (LSTM)

RNN units have the limitation of not being able to hold long-term dependencies due to vanishing gradients. The weights of the neural network is updated using the gradient descent algorithm. As the network descends into lower layers, the gradients get smaller. A change in the gradient teaches the model something new. The output of the network is affected by this alteration. However, if the gradient difference is very small, the network will not learn anything and hence there will be no difference in the output. As a result, a network facing a vanishing gradient problem will not be able to find a good solution [24]. This problem is solved by using LSTM by using four different network layers rather than using just one as in the simple RNN. All these layers are associated with each other forming a concrete structure to deal with the problem of long-term dependency. LSTMs consist of a memory cell, an input gate, an output gate and a forget gate. They are designed to remember the essential information by the use of cell states (C_t) as shown in Figure 2.2.

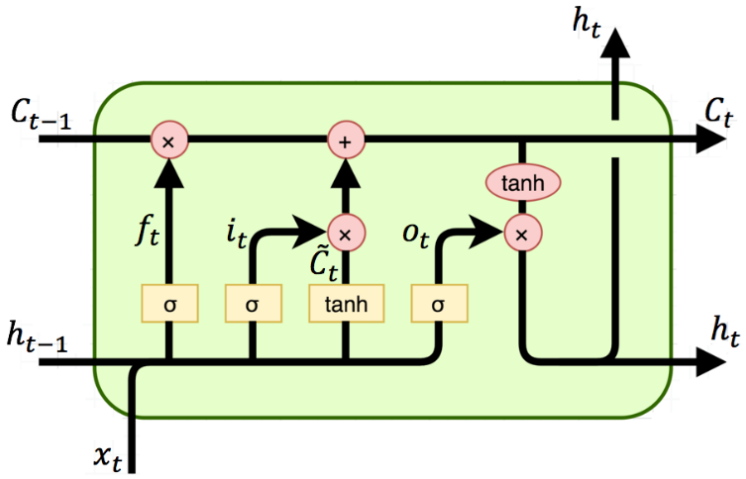


Figure 2.2: An illustration of the process done inside an LSTM memory cell from the article [13]

The first step in a LSTM cell is a sigmoid operation on (h_{i-1}) and (x_t) deciding what information will not be remembered. The output *(f_t)* from the equation 2.1 is computed for each element in the cell state (C_{t-1}) where 0 means to discard the information completely and 1 retain the complete information. The next step in the cell determines which information will be stored and is done in two steps. The sigmoid layer specifies which values have to be updated, and the new candidate values are created by a tanh layer. These both steps are computed by the equations 2.2 and 2.3 respectively. b_f, b_i, b_c, b_o are bias vectors.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.1}$$

$$I_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.2}$$

$$C'_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.3}$$

Later, the final cell state (C_t) is calculated as shown in equation 2.4. The final step is to compute the cell output (h_t) (equation 2.6) which is calculated by using a sigmoid (O_t) (equation 2.5) and a tanh activation function, so that it forced to return the values between -1 and 1.

$$C_t = F_t \odot C_{t-1} + I_t \odot C'_t \tag{2.4}$$

$$O_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.5}$$

$$h_t = O_t \odot tanh(C_t) \tag{2.6}$$

2.3.2 Gated Recurrent Unit (GRU)

GRU is another technique that solves the vanishing gradient problem of RNN. It is quite similar to LSTM except that the output of GRU consists of two gates: *update gate* and *reset gate*. The key benefit of GRU is that it preserves information for a considerably longer period of time [9]. The update gate (Z_t) specifies how much past data must be passed to future and is computed by the equation 2.7. The reset gate (R_t) indicates how much of the past data must be forgotten and is computed by the equation 2.8. A GRU cell is represented in Figure 2.3.

$$Z_t = \sigma(W^{(z)} x_t + U^{(z)} h_{t-1}) \tag{2.7}$$

$$R_t = \sigma(W^{(r)} x_t + U^{(r)} h_{t-1}) \tag{2.8}$$

Finally, a tanh layer and an update gate output are used to calculate the current state memory.

$$h'_t = tanh(W x_t + r_t \odot U h_{t-1}) \tag{2.9}$$

$$h_t = z_t \odot h_{t-1} + (1 - z) \odot h'_t \tag{2.10}$$

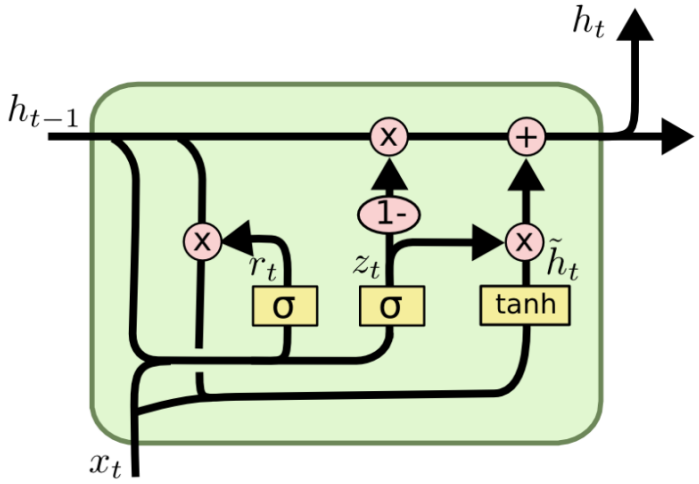


Figure 2.3: Illustration of Gated Recurrent Unit cell with the update gate and reset gate from the article [13]

2.4 Sequence-to-Sequence Learning

Deep Neural Networks (DNN) are incredibly powerful machine learning models that excel at challenging tasks such as object detection, speech recognition etc. They have the ability that to perform parallel computations and perform complex computations in less time, hence making it powerful. One of the major challenges while working with deep neural networks is dealing with sequences of variable lengths. Sequence to Sequence mechanism can be used in these scenarios of handling variable lengths and map sequences to sequences.

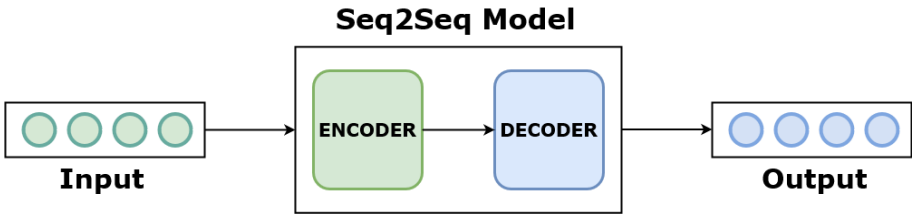


Figure 2.4: A general sequence-to-sequence architecture with encoder and decoder components

Sequence-to-sequence learning is the process of training models to translate sequences from one domain to sequences of another domain. It is build using recurrent neural networks or its variants. The main components are encoder and decoder. Encoder converts an input sequence into a series of continuous representations that are subsequently fed to the decoder. The decoder reverses the process by using the previous input and input sequence to turn the vector into a target item. An encoder and decoder can have multiple RNN cells to process the sequences where the sequences are processed one step at a time which are extracted by the next cell. The study

from the paper [36] introduces an architecture to handle the sequence-to-sequence problem using an LSTM architecture. The experiment was done to translate an English text to French and have achieved a BLEU (see section 2.10.1) score of 34.8. This introduced the concept of sequence to sequence learning which is now widely used in various text related problems.

2.5 Attention mechanism

A strategy that has recently been employed to improve translation is the attention mechanism. The idea behind attention is for the network to focus on different sections of the input for different time steps by enhancing few parts of the input sequence. The part to focus on and to decide which is more important depends on the context of the problem and is trained by gradient descent algorithm. The mechanism was introduced by Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio et al. [5]. Suppose, there are two 10-word statements that are practically identical except for two words. Encoders and decoders must be subtle to depict the change as there is a very minor shift in space. The author with this kind of imagination addressed the problem by allowing the decoder to focus on specific areas of the input because normally the fixed length vector has to encode the complete sequence.

The attention weights are calculated using the current state of the hidden layer and each encoder output, yielding a vector to be of size equal to the input sequence. These weights are multiplied by the encoder outputs to produce a context vector. The context vector is the weighted sum of encoder states, where each weight (a_{ij}) is the quantity of attention paid to the output of the encoder which is considered as the decoder input (h_j). The context vector is computed by the equation 2.11.

$$c_i = \sum_{j=1}^{T_x} a_{ij} h_j \tag{2.11}$$

where each weight a_{ij} is a normalized attention energy and is given as 2.12

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{2.12}$$

where the attention energy a_{ij} is computed using some function a using the last hidden state s_{i-1} and the decoder input and is given as in equation 2.13.

$$e_{ij} = a(s_{i-1}, h_j) \tag{2.13}$$

Minh-Thang Luong, Hieu Pham, Christopher D. Manning et al. [25] explain about "global attention" models, where the major difference is how the attentions scores are calculated. In global attention, all the states of encoder are used to generate context vector which is illustrated in the Figure 2.5. The attention weights at each time step t is computed by considering the variable length a_t based on the current

decoder state h_t and encoder states s_i. The global context vector will then be the weighted average for particular a_t.

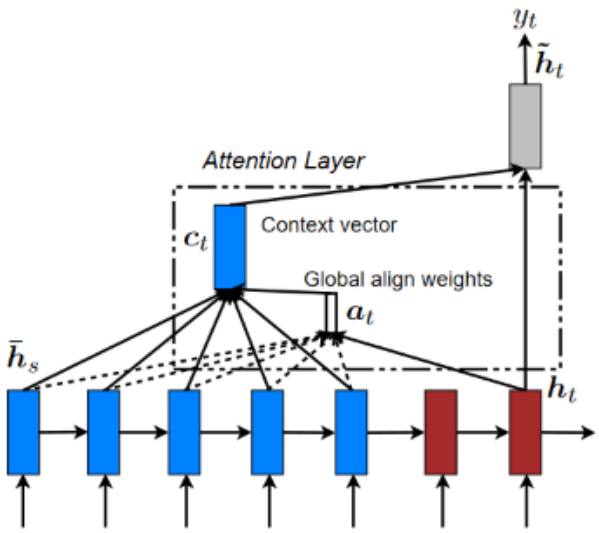


Figure 2.5: A representation of how the context vector is computed in a global attention model [25]

Attention scores can be calculated by using different functions which are of three kinds: (a) a *dot* product between the encoder state and decoder state; (b) *general*, a dot product between the hiddden state of the decoder and the encoder state; (c) *concat*,a dot product of a new parameter and the linear transform of all the states combined [10]. These scoring functions help in building specific attention modules and gives the flexibility to switch between different scoring methods. In our study, the 'general' scoring mechanism is used. The scores are defined in the equation 2.14.

$$score(h_t, \tilde{h_s}) = \begin{cases} h_t^T \tilde{h_s} & \text{dot} \\ h_t W_a \tilde{h_s} & \text{general} \\ v_a^T W_a [h_t; \tilde{h_s}] & \text{concat} \end{cases} \tag{2.14}$$

2.6 Word Embedding

NLP-powered systems generally have the ability to recognize words, grammar and other language characteristics to process and generate text. Since computer systems can only grasp numbers, the text is often converted into a numerical representation. Word embedding or word vectorization is the process of mapping words to a vector of real numbers. The transformation of text to vectors is done in such as a way that words having similar meanings have similar vector representations. This tells that the word embedding is formed by understanding the semantic meaning of the words present in the text. There are different kinds of techniques on how the embedding is performed. The most common embedding techniques are TF-IDF, based on

the word importance and frequency of words in the corpus, word2vec, a mechanism where cosine similarity is used to find similarities among words, GloVe, a pre-trained word embedding built as an extension to word2vec and many others. In our thesis study, the PyTorch embedding is used, which store the vector representation for all the words in a dictionary. It acts as a lookup table. Further explanation about the embedding is described in the methodology of the study.

2.7 Masked Cross Entropy Loss

Cross entropy (CE) loss is evolved from the information theory where entropy is defined as the measure of uncertainty for outcomes of a random variable. It is a measure of the difference between two probability distributions for a given random set of events [4]. This is a very commonly used technique in deep learning models and is used to optimize the neural networks by minimizing the loss. The probabilities of predictions are compared to the actual output after which a loss is computed where the probability is penalized so that it can minimize the differences between predicted value and actual value [22]. Cross Entropy loss can be calculated by the equation 2.15.

$$L_{CE} = -\sum_{i=1}^{n} t_i log(p_i) \tag{2.15}$$

where, n is the number of classes, p_i is the predicted probability of ith class and t_i represents the desired distribution of the classes.

There is a enhanced version of cross entropy loss which is used in sequence-to-sequence models known as masked cross entropy. In sequence generation problems where the length of the output is not fixed, the sequences are often padded or trimmed to form batches of sequences, all having the same length in order to feed to the model. Due to this padding, it leads to a challenge in the optimization phase during the computation of gradients and updating the loss through back propagation. To overcome this, the approach taken in masked cross entropy is to mask the loss i.e. set the loss to zero before back propagation if it is computed using the pad tokens, and then update the weights and biases. In our study, masked cross entropy has been used to calculate the losses during the training process.

2.8 Code Deobfuscation

Code obfuscation is an approach to modify source code in order to make it difficult for humans to understand. Code deobfuscation is the process of recovering the obfuscated code. In the context of programming code, there are elements that are unimportant and would often lead the to memory waste and poor model performance. For example, a variable name or function name can be anything in the code. This type of information is not required for the model to learn and predict. On the

basis of code obfuscation, Facebook has presented a new objective, Deobfuscation (DOBF) [33]. The main idea behind DOBF is to obfuscate various elements in the code such as the variable and function names etc., with special tokens. In other words, the elements are masked with some special tokens and are then fed to the model for training. Once the training is completed, the obfuscated elements can be recovered to their original state.

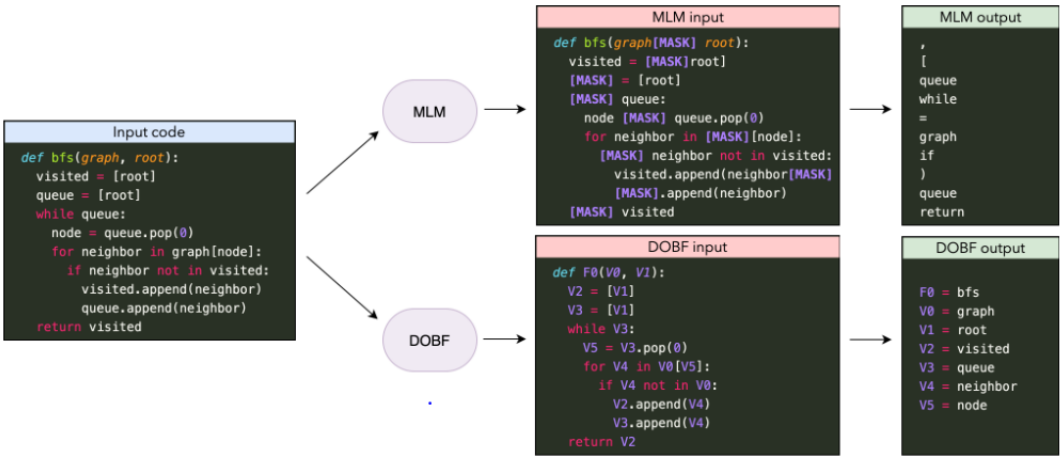


Figure 2.6: Working of MLM and DOBF approaches from the paper [33]

A similar approach was developed before named as Masked Language Modelling (MLM) where random words are masked in the input text and then recovered later. The DOBF approach differs from the MLM approach where all the occurrences of the variables are masked with the same special token [33]. It can be better visualized in the Figure 2.6

2.9 Robot Framework

Manual testing is a traditional method of testing which requires a tester to perform the operations. However, as the organizations grow and develop large products, the time taken to test all the functionalities also increases. Therefore, manual testing has become time and cost intensive for organizations. Additionally, the integration testing also becomes quite complex [35]. In recent years, industries have started to adopt test automation processes to evaluate a software's functionality and identify errors. There are a variety of commercial and open source test automation tools available, but only a few are suited for black box testing. Moreover, many of the tools available are ideal for unit tests carried out either by quality engineers or developers [27].

Robot Framework is a simple, yet powerful tool that leverages the keyword driven testing approach. Keyword-driven testing is explored as a framework since it allows test scripts to be executed at a higher degree of abstraction. It is similar to a service

or subroutine in programming, in which the same code can be run with multiple values, making it an excellent alternative for the desired automation [17]. Robot framework follows this keyword-driven methodology. It can be used for automating test processes to construct powerful and versatile automation solutions [1]. The framework helps in connecting with any other tool and can be utilized in distributed, heterogeneous situations where diverse technologies and interfaces are required for automation [2]. The support to the multi-platform environment makes it one of the popular open source tools and is widely used for acceptance testing and test-driven development.

Robot Framework uses human-readable terms and therefore has a simple syntax. Libraries can be written in Python, Java, or in a variety of other programming languages [1]. It has a highly modular architecture where the test data is presented in a simple, editable tabular format. A glimpse of how the robot codes are written is shown in the Figure 2.7.

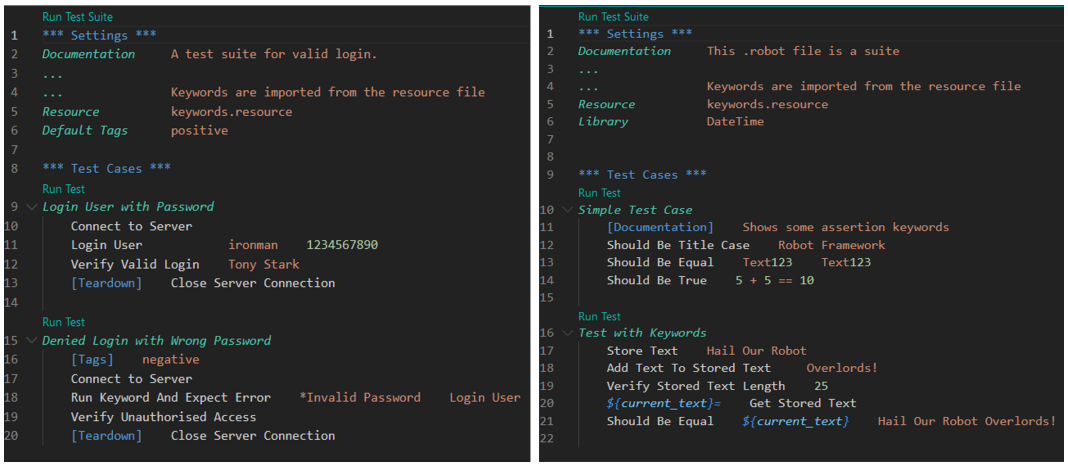


Figure 2.7: Sample examples of robot scripts from their official website [1]

2.10 Evaluation Method

There are not many standard metrics for evaluating code generation tasks. As a result, while dealing with code generation problems, the majority of people rely on human evaluation. Currently, the evaluation will be carried out using two alternative ways. (a) On basis of historical data by splitting the dataset into training and testing data. This is done by using the BLEU metric. (b) Human expert validation, where code generated from the model in the form of robot syntax will be judged by a test expert.

2.10.1 Bilingual evaluation understudy (BLEU)

BLEU is an evaluation metric that matches the n-gram of translated text to reference text and make a count of matched n-grams irrespective of position. The score is

between the range of 0and 1. The BLEU score is computed by formula shown in the equation 2.17, where BP refers to brevity penalty, a exponential decay for translation extremes i.e. too small or too long, N represents number of n-grams, w_n is the respective weight (default is 0.25 for N=4) and p_n is modified precision which is sum of all counts of n-grams in translated text which is divided by the number of n-grams in the translated text [29]. The brevity penalty, BP, is computed by,

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{2.16}$$

where c is the length of candidate translation and r is the length of the target sequence. Then,

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n log p_n) \tag{2.17}$$

2.10.2 Human expert validation

The human assessment is used while evaluating the test cases for the new version of the software application. This is because the test cases are acceptance test cases, which do not have a ground truth target output against which the predicted output can be assessed. Therefore, a human expert who has the necessary domain knowledge with respect to both the acceptance test cases and the robot framework makes it an ideal scenario to evaluate the obtained results.

Human evaluation metrics entails asking human specialists to score the generated code on a variety of criteria. The human evaluation metrics include two aspects: Content-related, and Effectiveness-related. The evaluation metrics are inspired from the paper [19]. The code is rated by using the metrics on a scale of 0 to 5, with 0 indicating the worst and 5 indicating the greatest. It is also important to mention that the evaluation is majorly focused on the keywords that are generated in the predicted code while giving less priority to other elements in the code.

1. **Content-related**: This metric is used to measure the amount of contents that have been translated from the input description to the generated code. Two content related metrics are used to evaluate the predicted code.

- Adequacy: It is the state of being sufficient. In this study, it measures whether generated code misses some essential information.

- Conciseness: The quality of being clear. This metric is used to measure whether the generated code contains unnecessary information.

2. **Effectiveness**: This metric is used to measure the predicted code is useful and readable to the developers. This is done by evaluating it by two sub metrics: usefulness and code suitability.

- Usefulness: It is used to evaluate how useful the code is. For example, if all or some of the keywords generated are useful for developers.

- Code suitability: This metric is used to evaluate to what extent the generated code is suitable for the test description.

2.11 Related Work

Generating code from natural language is commonly researched for the general purpose programming languages where massive amounts of data is available. The problem arises when a task is addressed in a less commonly used language with less data. As a result, the pre-trained models will not produce good outcomes. There have been a few studies done in the field of solving problem when the source code language is less popular or the problem statement is domain-specific. However, generating a domain specific language where the content is dependent on various functionalities is very challenging is still considered to be a very challenging task. The purpose of this literature survey was to understand the methods and steps followed for a generic code generation problem even though it is different from the current problem.

The authors, Luis Perez et al. [30] have proposed a machine learning model for generating code in Python language by using a pre-trained language model (GPT-2[1]). The utilization of the embedding approach was an intriguing aspect of their research. They discovered that employing Byte-Pair encoding, as used in GPT-2, is a far superior technique for generating code than simply using characters. The purpose of this strategy was to avoid having to understand word-level embedding. They discovered that using words from NLP models was degrading the model's performance since most of the grammar involved in creating code are not from ordinary vocabulary. The paper also describes the implementation of a fine-tuned GPT2 model which consists of 117M parameters. The dataset used was *CodeSearchNet* which consisting of around 2 million (comment, code) pairs scraped from various open source libraries such as github, stack overflow etc., having a mixture of different programming languages. The idea of using python docstrings as the natural language text has encouraged us to follow a similar strategy to build the dataset for the current study. It also helped in understanding how to mitigate the problems faced with long sequences. Apart from the pre-trained model, they have also build an RNN model as their baseline where the training was performed based on many to many sequence model by taking 50 characters in a sequence and then predict next character using previous sequence. This resulted in higher chances of gradient explosion while processing long sequences. The problem was solved by using *gradient clipping*. However, in our study, the number of tokens present in the code is much higher and the amount of data is less. Therefore, using a pre-trained model may not result in good outcomes.

[1]https://huggingface.co/gpt2

Srinivasan Iyer et al. [21] presented an approach on how to map natural language to code for a particular environment. The goal of the project was to construct class member functions in JAVA using English documentation. They employed a specialized neural encoder-decoder model that encodes natural language as well as distinct units for environmental identifiers using a two-step attention method. This way of having two step attention model helped them to match the words with the identifies in the environment, thereby enhancing the learning of the model. The application and motive for adopting Bi-LSTM aided in the comprehension of how to deal with these types of issues. Our work is fully based on domain-specific functional tasks that follow a different syntax and structure. Furthermore, the block of code in robot scripts are not contained within a single method; rather, the functions may call other functions internally, which is different from the problem they are attempting to solve.

A study proposed by Li Dong, Mirella Lapata et al. [12], had a similar challenge but instead of generating code they concentrated on generating meaningful representation with the help of semantic parsing. In other words, they tried to address the problem where minimal domain knowledge is required and try to map the natural text to effective logical form. The implementation involved the usage of attention mechanism. Another study by Sajad Norouz et al. [28] found that transformer-based seq2seq models can reach a competitive or superior performance with models specifically designed for semantic parsing. They also propose an alternative to inductive bias design for future advancement. However, both the works do not serve the purpose of generating executable code. Moreover, in the context of test code, most of the code depends on how well the description is written. This is not ideal in our study as we observed that not all the descriptions are clear and detailed.

Although mapping natural language to fully executable programs is a hot topic in the field of natural language processing and software engineering, most existing research is based on a completely structured syntax, such as generating regular expressions from test in the paper [23], or generating database queries as explained in the paper [20]. Another study has presented the usage of neural network models to link natural language to a sequence of API calls [15]. Our study is much more complicated with numerous elements involved in a robot file, such as API calls, function calls, and so on, making it difficult to frame it in a systematic manner. Therefore, our strategy is to use a transformer architecture, which is a tried-and-true method for sequence to sequence translation. When mapping text to rigorous syntax based modules, the attention technique is also highly promising. Our main goal in this thesis is to get the output text closer to natural language and build a model that is considerably more effective at learning the structure.

In the context of testing, there has been work done in generating test cases from the requirement specifications. Several studies have demonstrated the generation of test cases from specification documents. The study done by Imran Ahsan et al. [3] have investigated several NLP techniques to identify the right techniques to generate automated test cases. They have identified six NLP techniques which are beneficial for developers while working with test generation problems. This helped in under-

standing the different techniques that can be used to analyze the text from the test descriptions since the work was mainly focused on system testing and acceptance testing. The popular techniques used were parsing, tokenization, POS tagging etc.

The study by Chunhui Wang et al. [41] proposed a system to generate test cases from use case modelling for system tests (UMTG). The approach involved analyzing the behavioral information using natural language processing techniques by which the test scenarios were identified. Their approach needed a domain model (e.g., a class diagram) which enables them to create input constraints. It is also said that these requirements are very commonly used in different companies and therefore was developed using the use case specifications. This work has directed us to other works which involved generation of test cases by the usage of activity diagrams or abstract syntax trees [32, 40, 39]. The work done by David Turner et al. [39] explains an activity oriented approach where the activities and their dependencies are represented as an activity diagram which helps in generating a workflow. A model was developed which consists of domain objects and activities where objects are basically the state of the application, user details etc. while the activities are the interactions with the system. This model generates the test codes based on activity diagram and its workflow dependencies. But these works are far from the current problem because except the fact that it has helped in understanding the various NLP techniques used, there were no solid approaches in how to generate test code from the specifications. Most of the above works for generating test cases used formal models where the specifications were first translated to formal specifications which then were used to generate the test cases. The current work is involved in analyzing the test case specifications directly to generate the corresponding code which differed from the previous works.

Another interesting work which has helped in understanding how to make use of the special tokens which consisted in the on both the input and target was from the work done by Alzahraa Salman as part of their thesis study “Test Case Generation from Specifications Using Natural Language Processing”. The study involves mapping the keywords from test case specifications to label vectors of available test scripts [34]. The problem was seen as a multi-label text classification problem where the implementation involved training the model over a few test scripts and the output therefore would be from one of them. Therefore, given a test case specification, there is a possibility that more than one test script can be produced as output, thereby the problem is seen as a recommendation system. The major drawback of the study was that the model could not suggest unseen test scripts. Our study is focused on understanding the data available in the code so as to generate a new test code when a test description is inputted.

3

Methods

This chapter describes the methodology of the proposed approach and how the work has been carried out as part of the thesis. The work is centered around creating models that can generate the test code from test specifications. First, an overview of the technique is given, as well as the several phases in the proposed approach. The technique for constructing the dataset is described in section 3.2. Various pre-processing approaches are described in section 3.3 and 3.4 focusing on the NLP techniques that are applied on the data. The next section 3.5 involves the workflow and algorithms used to build the model and test on the data.

3.1 Overview

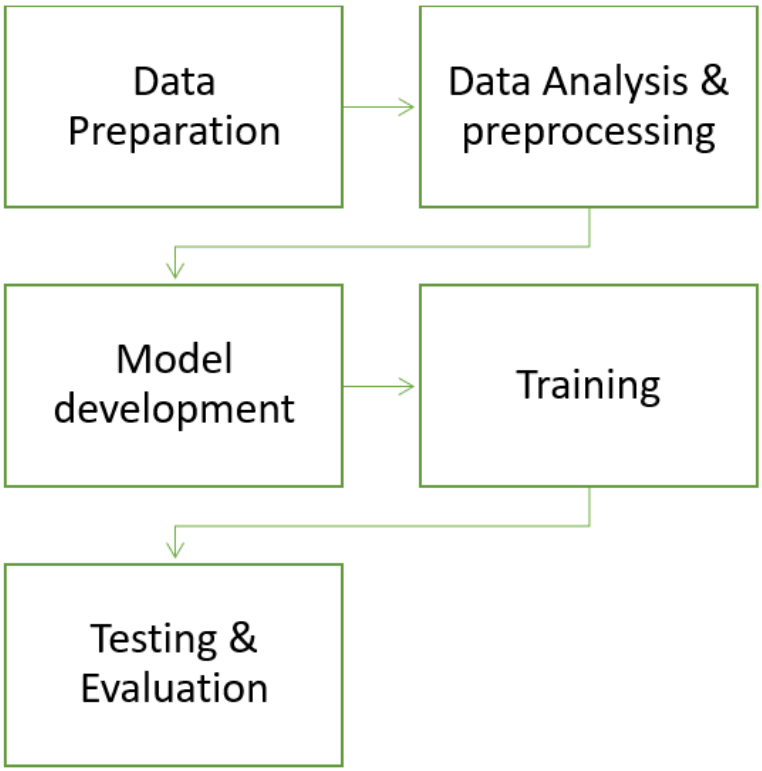


Figure 3.1: Stages of the methodology for the proposed approach

The aim of this project is to create, test, and evaluate a method for automating the generation of test code from test case specifications expressed in natural language. Figure 3.1 shows the different stages of our approach in this study. The implementation is based on using NLP and deep learning techniques. This is accomplished by parsing the data and creating feature vectors, which are then fed into the model. On the other hand, the test code is also converted to feature vectors with some additional preprocessing. The input and output are the specification and the code, respectively.

3.2 Data Preparation

At Ericsson, the data needed to complete this operation was not readily available. As a result, the first stage in this project was to extract the necessary data and create the dataset in order to examine and comprehend how the test cases and test scripts are written. As mentioned earlier in section 1.1, the product development team at Ericsson deals with two versions of applications. First, a monolithic application, which was created many years ago and describes a single-tiered program because all of the code is written in one piece. This version of the application has test cases written using Robot Framework. Having said that, there are no corresponding test case descriptions through which the executables were implemented. Every test instance, however, had a short block of information tagged as "*Documentation*" in the code. This information served as the test case's description, while the accompanying code served as the target value. Figure 3.2 shows a sample of the structure of the robot test code in which the documentation is extracted as the input and the remaining text served as the target value.

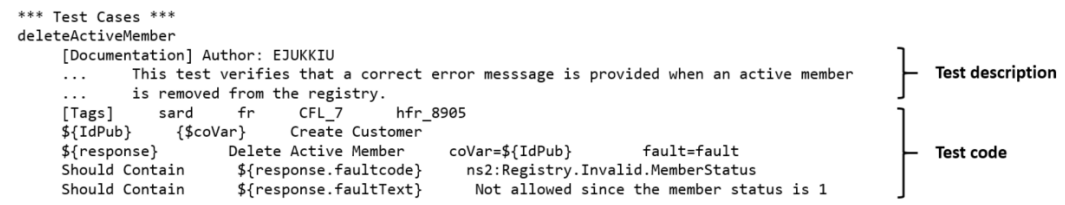


Figure 3.2: An example of Robot test code from the monolithic application. The test description acts as our input while the remaining code becomes our test code

The newer version, on the other hand, is built on test-driven development and comprises acceptance test cases, which should eventually replace the monolithic program. This version has a small amount of data and no ground truth, i.e. no executables. To elaborate, the updated version just contains test descriptions; the robot test code has not yet been implemented. This data will primarily serve as our test data i.e. will serve as unseen data, meaning the model will not be trained on it. These test case specifications in natural language are bound to be inconsistent or ambiguous at times. Furthermore, these specifications are semi-structured and can be of varied lengths. These test cases steps are considered as individual test cases each of which

corresponds to a small module in entire block of code (section 1.4).

Size of each Robot file	~10KB
Total no. of Robot files	898
Total no. of Robot scripts	4.3K

Figure 3.3: Statistics of the corpus used in the project

3.2.1 Regex Parsing

The goal was to utilize regex expression to extract the required information from the code, as demonstrated in the sample code in Figure 3.2. This facilitates the conversion of data from an unstructured to a structured format as shown in Figure 3.4. The first step was to convert all of the code scripts from *.robot* extension to *.txt*, so that they could be read and parsed. Once the text files were available, a python parser package named *pyparser* was used to parse through each file and retrieve the information needed using a regex expression. The final dataset is saved as a csv file consisting of around 4000+ elements of the data with two columns namely the test description and the test code. The summary of the data statistics is shown in Figure 3.3.

Test description	Code
Author: Adamei (EUIJTED) ... ... This test checks that correct error message is returned in case command SUPER_ADD_CHARGE is executed for a Thermo plan passing a list of Super PPs. For Thermo PPs, the list of Super PPs should be empty. The command will automatically determine the Super Application Providers.	addSuperChargeEventInvalidSuperPPs sard frt licenseON=RHS_SARD_frt_GENERAL #create Customer and Contract ${appProvingId} Set Variable SARDThermoPlanTOMV ${cfId} Set Variable SARDCFSMKO ${customerId} Create Company Holder 54 ${code} Create Form ${appProvingId} ${customerId} ${cfId} #add Super Charge ${eventStatusCode} Set Variable 99 ${appProvidingIdSuper} Set Variable SARD Thermo Super PP ${response} Add Super Charge ${code} ${eventStatusCode} appProvingId=${appProvidingIdSuper} Should Be Same ${response. code} ght4:SuperPPNotAllowedForThermoplan
Author: Condensut (EJLOTDE) ... ... This test checks that correct error message is returned in case command SUPER_ADD_CHARGE is executed for Thermo plan passing an invalid EVENT_CODE.	addSuperChargeEventInvalidEventCode sard frt licenseON=LHS_SARD_frt_u765 #create Customer and Contract ${appProvingId} Set Variable SARDThermoPlanTOMV ${cfId} Set Variable SARDCFSMKO ${customerId} Create Company Holder 54 ${Code} Create Form ${appProvingId} ${customerId} ${cfId} #add Super Charge ${eventStatusCode} Generate Random var 143 ${response} Add Super Charge ${Code} ${eventStatusCode} fault=fault Should Be Same ${response. code} ght4:SuperCharge.CodeDoesNotMatch

Figure 3.4: Sample data in tabular form after extracting the text and code separately from the robot scripts using parser and regex expressions

3.3 Data Preprocessing for Test case description

3.3.1 Overview

This section describes the transformation of the data into suitable feature vectors in order to input to the model. In order to do this, the data needs to be cleansed and

formatted. This is achieved by using several NLP techniques. The Python library *Natural Language Toolkit (NLTK)* is used to implement most of these steps. Figure 3.5 shows different kinds of preprocessing steps on the test case descriptions which are in natural language.

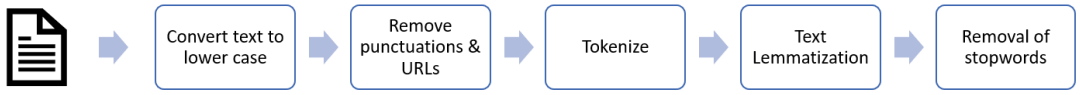


Figure 3.5: Pre-processing steps performed on the test case descriptions

3.3.2 Lowering Case

The initial pre-processing step in every text analysis problem is to lower the case of the text. The reason for normalizing the case is that the model may treat a word that starts with a capital letter at the beginning of a sentence differently than the same term that appears later in a sentence without capital letters in the vector space. This might lead to a decrease in the performance of the model. To change the case of the text in the dataset, Python's string `lower()` method was utilized.

3.3.3 Removing Noise

Articles or text that contain symbols are not often used in the English language are considered noise. In the case of this project, the special characters, as well as other non-essential information like URLs and stopwords are considered to be noise. The special characters are generally the punctuation marks, which are used to break up text into sentences, paragraphs, and to frame sentences, among other things. Non-removal of these characters will have an impact on the results, especially when using a text processing approach to find the frequency of terms and phrases. It also contributes to the noise, which causes uncertainty in the model's training. Python's string punctuation technique is used to choose which symbols or characters to keep. The implementation involves searching the data set and removing all the entries that contain symbols that are part of the string punctuation method's pre-initialized string.

Furthermore, since the project deals with test cases, it is bound that there will be a lot of parameters or configurations that are often described as part of the test descriptions. One of the most common elements observed is the presence of URLs in the text. Removing URLs helps the model to focus on the important features rather than focusing on elements which do not carry much information. Therefore, a regex expression matching the pattern of the URL is created and used for eliminating URLs. Apart from this, in any human natural language, there are plenty of stop

words. Removal of these stopwords[1] helps in eliminating the low-level information from the text, allowing to focus more on the crucial information. The elimination is done using the NLTK library.

3.3.4 Tokenization

A lot of the embedding methods such as BOW, TFIDF, and Word2Vec function by embedding text data in the form of tokens. Thereby, tokenization is required to break down the sentences in the test case description into separate tokens. A tokenizer breaks the natural language text into chunks of information that can be regarded as separate elements. These token occurrences in a document can be utilized to create a vector that represents the document. Tokenization can separate sentences, words or letters. Word tokenization is splitting a series of words into discrete elements. In the case of this study, the text is tokenized into discrete elements of words. This is achieved by using the NLTK library which contains the `tokenize` package. Figure 3.6 shows a sample text with the sequential transformation of the description to the pre-processed format.

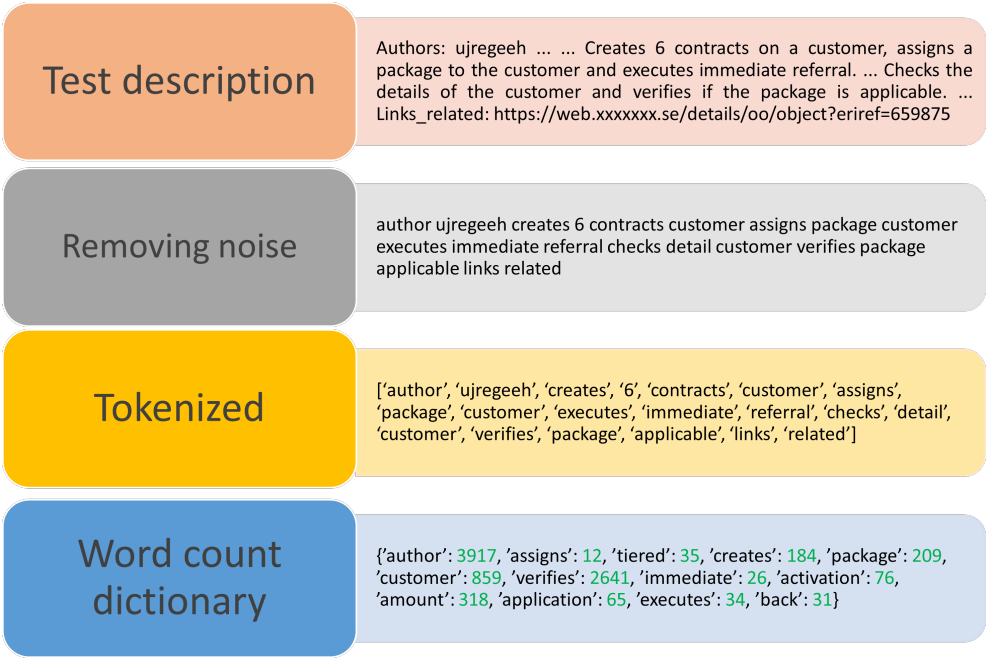


Figure 3.6: An example of the sequential transformation of the test description pre-processing

3.3.5 Normalization

The process of converting text into a standard format is known as normalization. It aids in reducing the amount of unique tokens in the text, hence eliminating text

[1]https://pythonspot.com/nltk-stop-words/

variants. Lemmatization is one of the popular normalization methods which is used in text processing. It is the procedure for finding a word's base form. This is accomplished through the use of vocabularies and word morphological analysis. The inflected components of the words can therefore be recognized by a single unit known as the lemma. The NLTK library has lemmatizer package which makes it easier to implement on any text processing problem. This is done after the tokenization of the text.

3.4 Data Preprocessing for Test Code

3.4.1 Overview

This section explains how to convert the text in the code into appropriate feature vectors for feeding to the model. The normal preprocessing approaches used in most of the text processing problems cannot be utilized in this scenario since code is built up of language with grammar, keywords, and structure, each of those pertaining a special meaning. These elements cannot be viewed as ordinary text and hence cannot be deleted. As a result, there was no preprocessing required for test code and therefore, the model development began. However, when the data loader (explained in section 3.5.1) was developed, there were a large number of tokens, the most of which were unnecessary. There were elements in the model that didn't need to be processed from the model building perspective. By analysing the code, one mechanism that emerged as being useful and had the potential to change the course of the model training was explored. The concept of code deobfuscation was used in the pre-processing of the test code.

3.4.2 Masking the code

As described in section 3.4.1, code contains different elements such as variable names, variable values, function names, keywords, and other elements. The keyword functions are critical parts for the model to predict because the code is created using a robot framework, which is a keyword-driven testing technique. Variable names and function names used in any programming language can be of any form, a letter or a word. Hence, suggesting suitable variable names or function names by the model is a difficult task as it requires the understanding of the context and the domain. Therefore, the technique of code deobfuscation is capitalized to mask various elements in the code. The concept of code deobfuscation is explained in section 2.8.

In our study, the variable names in the code are represented in curly braces preceded by a dollar sign. This can be seen as part of the example robot script shown in the Figure 3.2. It can be observed that the variable name "response" is represented as `${response}`. This pattern is used as an advantage in implementing the deobfuscation approach for the variable names in robot scripts. All occurrences of different variables are replaced with masked variables in the form of special token represented as `VAR_i`, where i is the count of the variable. This technique is imple-

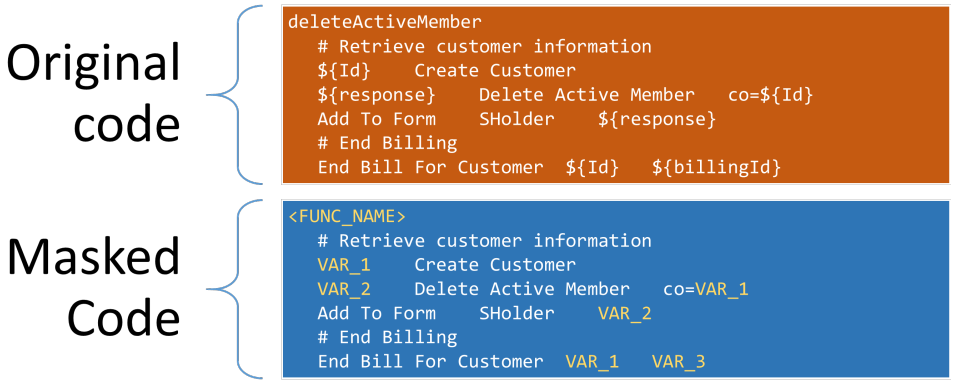


Figure 3.7: Example of a robot script masked with special token represented as `VAR_i` for variable names and `<FUNC_NAME>` for function names

mented by using the NLP techniques to tokenize the text, and regex expressions to parse through the text and search for the required pattern to mask it with a special token. Similarly, it can also be observed that every robot script starts with a unique name pertaining to that testcase. Hence, this term is also masked with a special token `<FUNC_NAME>`. This special token is used to replace all the unique function names of the robot scripts. Overall, the deobfuscation works similar to supervised machine translation, with a seq2seq model trained to map an obfuscated code into a dictionary represented as a sequence of tokens [33]. Figure 3.7 shows how the original code is transformed into a code with masked variables. A dictionary is maintained to store the mapping of the variables i.e., the original variable acts as the key and the masked variable represents the corresponding value.

Additionally, each code script includes certain parameters. These parameter values are generally mentioned in the test description. Since these configurations are removed from the test description during the preprocessing stage, they cannot be predicted in the target output. Therefore these parameters shall be removed from the code as well. But eliminating the parameter values can affect the format and structure of the code. Therefore, these parameters are masked with a special token `<PARAM_VALUE>`. The goal is to replace all parameter values with this special token so that the model wouldn't have to learn the specific parameter values but only that there is some value that exists in the code. This will help the model learn better and predict the special token rather than trying to predict the specific value of the parameter. Similarly, the numeric values are also masked as `<NUM>`. The challenge while implementing this mechanism is that unlike the variable names in the code, there is no specific format on how these parameter values are represented and

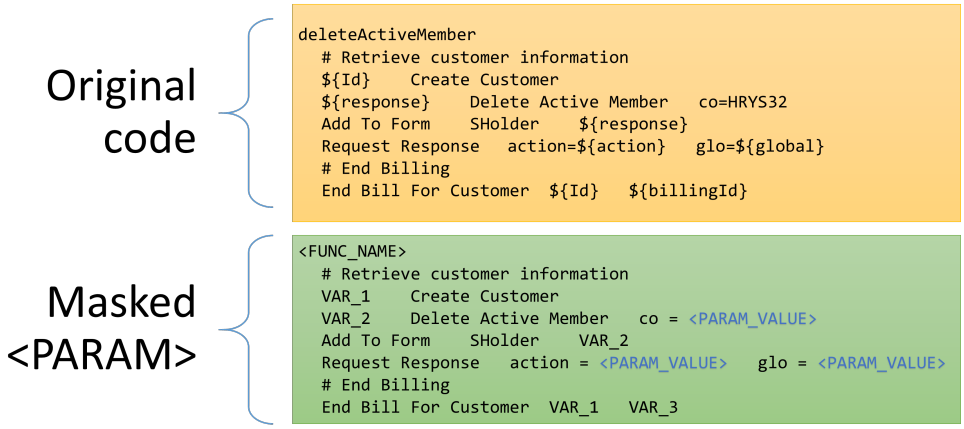


Figure 3.8: Example of a robot script masked with special token represented as `<PARAM_VALUE>` for parameter values

therefore is difficult to identify and mask them. Having said that, there might be instances where a parameter value still exists in code as that has not been identified while processing. An example of the masked parameter value is shown in Figure 3.8.

3.5 Model Training

The robot script contains various entities which are close to natural language. Therefore, this problem is seen as a translation problem dealing with a domain-specific language. Neural networks is a well-proven method for solving translation problems. Since the problem is seen as a mapping of one sequence of words/elements to another sequence of words/elements (which in our case is the robot output), any RNN model which has the ability to handle long sequences can be used. Having said that, LSTM and GRU becomes an obvious choice as described in Chapter 2. However, dealing with significant contextual mapping is an issue as RNN and its variants assign equal weight to each element in the sequence. Additionally, there is a considerable risk of missing vital information while modeling. To deal with this, an attention mechanism is also combined with the RNN algorithms used.

3.5.1 Building a dataloader

The construction of the dataset is explained in section 3.2. To use the saved csv file, the data should be loaded and fed to the model. The data is read and are formed into pairs of the input sequence and the target sequence. A crucial part of every machine learning system is data loading. Most of high-end configuration systems

won't have enough RAM to process the large data in its entirety. As a result, alternate methods are used to perform the task efficiently. Because the entire dataset cannot be loaded into the GPU memory, a dataloader is designed to sample batches from the dataset. At this stage, it is evident that the data corpus is divided into two parts: a training set and a testing set.

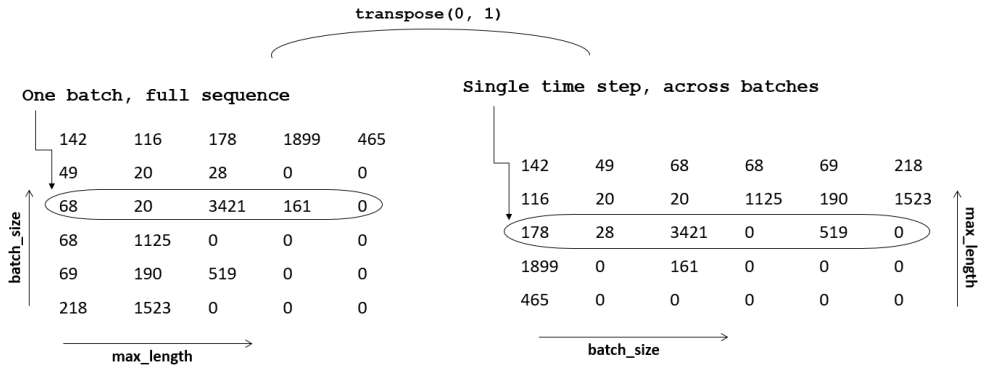


Figure 3.9: Left - LongTensor of the dimension `batch_size x max_seq_len`. Right - Tensor of dimension `max_len x batch_size` at single time step. The numbers represented are word indices that are assigned to each unique word

The train set is utilized throughout the training phase, which is batched according to the `batch_size`. The batch size can be configured manually and in our case, a batch size of 4 is used. The value is chosen based on the configurations and the memory space of the GPU used. The maximum sequence length is the length of the longest sequence in the training set. A LongTensor[2] with an array (batches) of arrays (sequences) is initialized to create a `(batch_size x max_seq_len)` tensor. Selecting the first dimension, for example, results in a single batch. However, when training the model, it requires a single time step at once, therefore the transpose, `(max_len x batch_size)` is used.

```
PAD_token = 0  # For padding short sentences
SOS_token = 1  # Start-of-instruction token
EOS_token = 2  # End-of-instruction token
UNK_token = 3  # For Unknown words

class Vocabulary:

    def __init__(self, name):
        self.name = name
        self.word2index = {"PAD" : PAD_token,  "<SOI>": SOS_token,
                           "<EOI>": EOS_token, "<UNK>": UNK_token}
        self.word2count = {"PAD": 0, "<SOI>": 0,  "<EOI>": 0,
                           "<UNK>": 0}
        self.index2word = {v:k for k,v in self.word2index.items()}
        self.num_words = 4
```

[2]https://pytorch.org/docs/stable/tensors.html

```
        self.num_sentences = 0
        self.longest_sentence = 0

    def add_word(self, token):
        if token not in self.word2index:
            # if the word (token) is making an entry for first time
            self.word2index[token] = self.num_words
            self.word2count[token] = 1
            self.index2word[self.num_words] = token
            self.num_words += 1
        else:
            # if the Word exists, increase word count
            self.word2count[token] += 1

    def add_sentence(self, sentence):
        sentence_len = 0
        for word in sentence.split(' '):
            sentence_len += 1
            word=word.strip()
            self.add_word(word)
        if sentence_len > self.longest_sentence:
            # tracking the length of longest sentence
            self.longest_sentence = sentence_len
        # Counting number of sentences
        self.num_sentences += 1
```

As seen in the Figure 3.9, the sequences are represented as numbers. To train the model, the sequences must be converted into a form that the models can process, which in our case is numbers. Numbers here are natural numbers that are assigned to each unique word. Hence, each sequence is split into words and are assigned a unique index. A helper class named Vocabulary is built to maintain a track of the words and indices, which splits the sentences, and assigns indexes to each unique word while maintaining three dictionaries: `word2index` (word to index mapping), `index2word` (index to word mapping) and `word2count` (frequency of each word). Each word's LongTensor is built using these indexes. The vocabulary for the test description and the test code are maintained separately by creating class objects for each. The implemented code is illustrated above and a sample of the dictionaries is illustrated in Figure 3.10.

In Figure 3.9, it can also be observed that not all sequences are of the same length and are padded with empty values (0) . The sequences have to be padded to the length of the longest sequence for a network to take in a batch of varied length sequences. As a result, every training sentence will have the same length, and the model's input becomes `(batch_size x max_seq_length)`. This is carried out for both the input and target sequences. To pad the sequences, a helper function is implemented and the same is shown below where the value of the `PAD_token` is 0.

```
# Padding the sequence with the PAD symbol
def pad_seq(seq, max_length):
    seq += [PAD_token for i in range(max_length-len(seq))]
    return seq
```

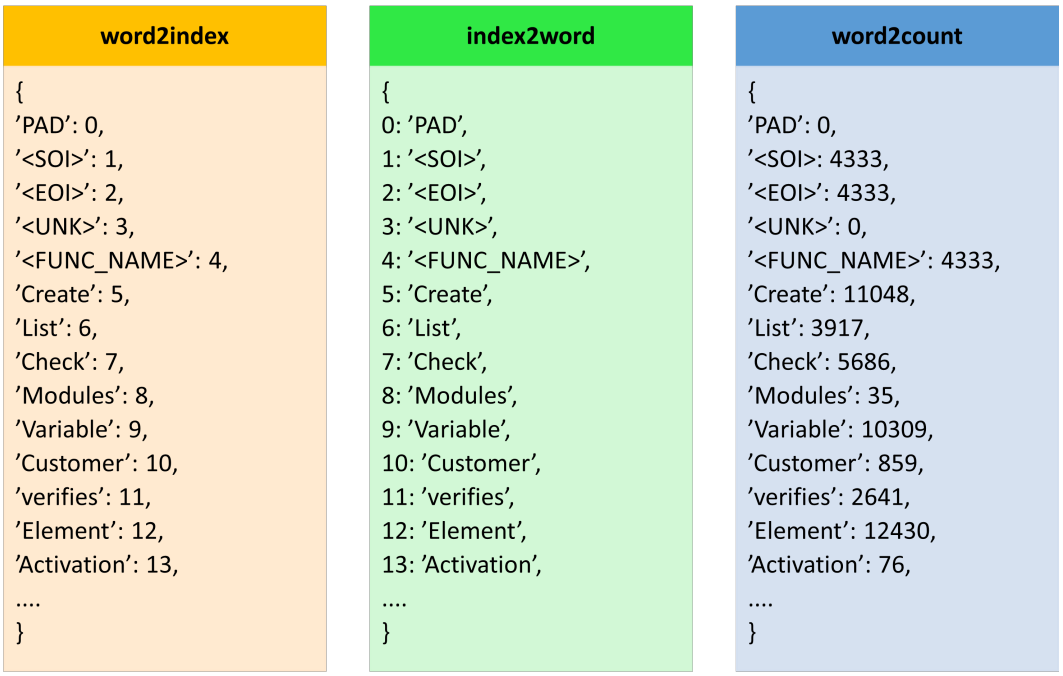


Figure 3.10: A sample of the resulting vocabulary dictionaries for the target column i.e. test code

In addition, a filtering technique is also implemented to filter pairs of a certain length. It is observed during our analysis that a significant majority of the sequence lengths are found to be between 10-500, thus any pairs that do not fall within this range are removed the train set. At each evaluation step, a random batch is used for evaluation. The evaluation frequency is set to 100 i.e. for every 100 epochs of training, there is one evaluation step.

3.5.2 Training the model - LSTM

The LSTM model as described in section 2.3.1 deals with the problem of the long-term dependency. In our study, the LSTM model is built using the encoder-decoder architecture. The encoder converts the input sequence into a single vector i.e. a context vector. The decoder then decodes the context vector, generating one word at a time to generate the target sequence. The model implemented in three modules; the encoder class, the decoder class and a seq2seq model class. The encoder class is used to process the input sequence, the decoder produces the output sequence while the seq2seq class acts as an intermediary between the encoder and decoder. The model is built with the inspiration from the works [38] and [16].

Encoder: The encoder built is a 2 layer LSTM. A high-level model architecture pertaining to our problem is shown in Figure 3.11. It can be observed that the tokens in the sequence are first transformed into numbers and are fed into the encoder hidden cells word by word. Therefore, the input to the LSTM cell would

be the embedding of the input word and the hidden state from the previous time step. Subsequently, to form the encoder network, the output from the first LSTM layer is passed as the input to the second LSTM layer. The arguments given to the encoder class are input dimension i.e. the vocabulary size of the input, size of the embedding layer and the hidden state, number of LSTM layers and dropout. The dropout value helps in avoiding over fitting.

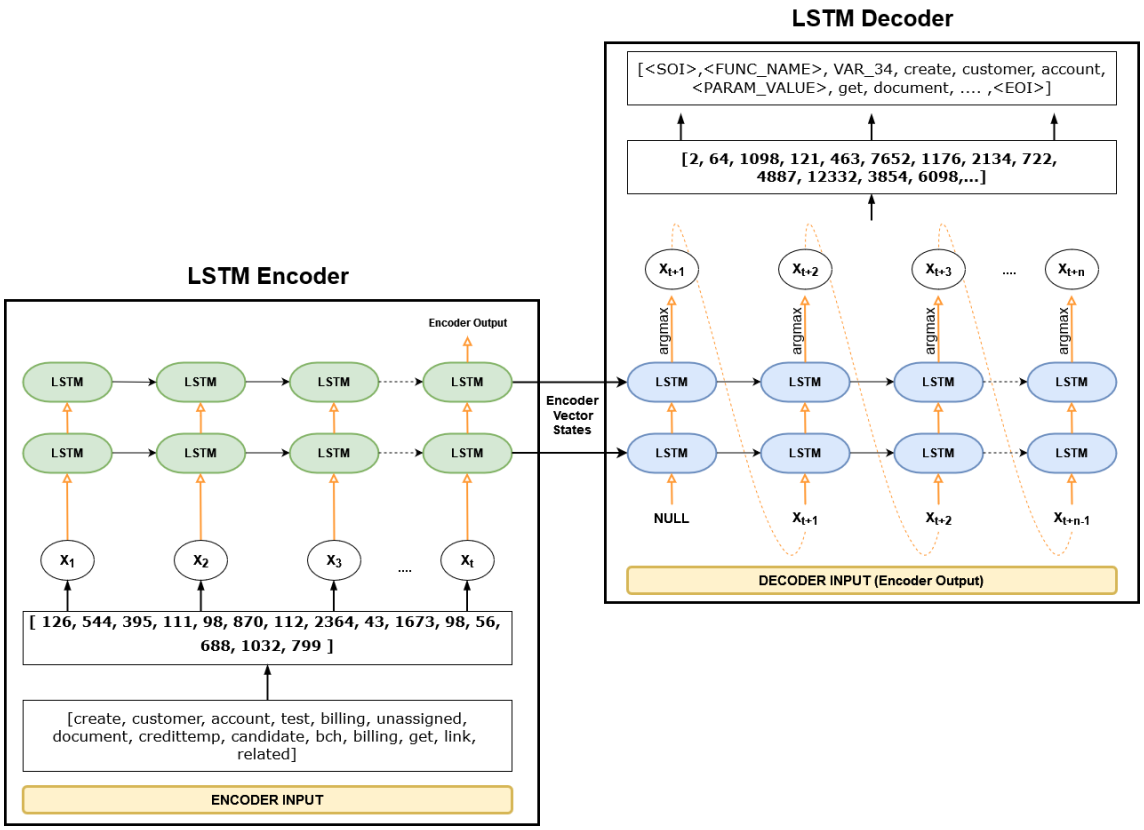


Figure 3.11: Representation of LSTM model using an encoder-decoder architecture. Left - The 2-layer LSTM Encoder block taking the input sequence. Right - The 2 layered LSTM Decoder block generating the target sequence

The model is built using the packages PyTorch and torchtext, making it easier to use various libraries to create layers and process the sequences. The libraries `nn.Embedding`[3], `nn.LSTM`[4], `nn.Dropout`[5] are used to generate the embedding, LSTM and dropout layers respectively. The encoder finally returns three values: the last state of the hidden layer for each time step, last hidden state and cell states for each layer.

Decoder: The decoder is also a 2-layer LSTM. The decoder is used to decode the sequences to generate the target sequence. Similar to the encoder, the input for the decoder is the embedding of the input word and the hidden state from the previous

[3]https://pytorch.org/docs/stable/generated/torch.nn.Embedding.htmlembedding
[4]https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html?highlight=lstmtorch.nn.LSTM
[5]https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html?highlight=dropouttorch.nn.Dropout

time step. It is important to note that the final encoder hidden state becomes the first decoder hidden state. The arguments given to the decoder class are output dimension, size of the embedding layer and the hidden state, number of LSTM layers and dropout. The output dimension refers to the vocabulary size of the target. Similar to the encoder, the embedding layer is then passed and dropout is applied. Essentially, this embedded input is fed to the decoder module together with the hidden and cell states. A linear layer is used to take the decoder output to output the prediction value at the end. The decoder finally returns the prediction together with hidden and cell states.

Seq2Seq: This class was build as a bridge between the LSTM encoder and LSTM decoder. The input to the function is the input sequence, target sequence and another parameter named teacher-forcing ratio whose value is set in the configurations. Teacher forcing ratio is a value which is used to decide the next token in the target sequence sent to the decoder. How it decides is by using the probability value. When the probability is equal to the `teacher_forcing_ratio`, the original target's next token is sent as the input to the decoder. If the probability is `1 - teacher_forcing_ratio`, the next token to the decoder would be the token that is predicted by the model. The major difference is that in the second case, the predicted token is sent to the decoder even when it is not the same when compared to the ground truth token in the sequence.

As mentioned earlier, the seq2seq class acts as the interface between the encoder and the decoder. As mentioned earlier, the input sequence is first sent to the encoder which outputs last hidden state and cell state. It then follows the following sequence of instructions for each iteration:

- pass the decoder input, previous hidden states and previous cell states to the decoder
- receive the prediction, new hidden and new cell states
- place the prediction in a tensor that is initialized prior
- decide if *teacher forcing* should be applied or not

Finally, when all the predictions are done, the method returns tensor where all the predictions are stored.

Training: After the model is built, a training function is implemented to begin training it. First, the model is initialized and the model configuration is set in the config file of the code repository. The dimensions for the embedding and the hidden states are set with respect to the size of the vocabulary. The parameters set for the LSTM model are shown in Figure 3.12.

The cross-entropy loss (explained in section 2.7) is calculated between the predicted and the ground truth tokens. The loss is back-propagated and the optimizer used is the Adam optimizer[6] with a learning rate of 10^4. On the whole, the model is run for

[6]https://pytorch.org/docs/stable/generated/torch.optim.Adam.html?highlight=adamtorch.optim.Adam

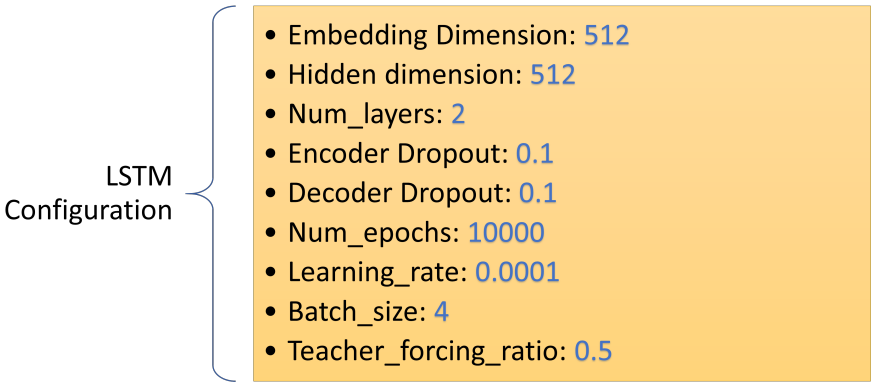


Figure 3.12: Hyperparameter configuration for LSTM model

10000 epochs. This helps in understanding how well the model is learning together with how well the predictions are made. At each iteration, the train method does the following tasks:

- generate the input and target sentences from a batch and feed the input and target into the model, first to encoder and then to decoder, to get the prediction
- compute the loss by using `loss.backward()`
- Use the optimizer to modify the parameters of the model
- total the loss and keep track of it

Finally, the mean of the loss returned computed over all batches. The loss is printed for each epoch while the evaluation is done for every 100 epochs.

3.5.3 Training the model - LSTM with attention

While a normal LSTM model can be used for sequence-to-sequence translation, a more advanced mechanism called the attention mechanism has lately been used to improve the translation as described in section 2.5. Therefore, the second model implemented in our study is by combining the LSTM with attention. As a result, the model is built in three stages, the encoder, the decoder and the attention layer for computing the attention weights. The model is build by taking inspiration from the work [10].

Encoder: The input to the encoder will be a batch of word sequences, a long tensor of size `(max_len x batch_size)`. The model architecture is shown in Figure 3.11. The inputs fed to the encoder are tensors of numbers replacing the text. Similar to the LSTM model, the arguments to the encoder class are the input size, size of the embedding layer and the hidden state, number of LSTM layers and dropout.
The libraries `nn.Embedding`, `nn.LSTM` and `nn.Dropout` are used to generate the embedding, LSTM and dropout layers respectively. The input to the encoder is the input sentence and the length of the input sequence. The embedding layer is used to transform the input sentence into dense vectors and are then fed to the LSTM

cells. The resulting output from the encoder is the last hidden state for each time step, last hidden state and cell states.

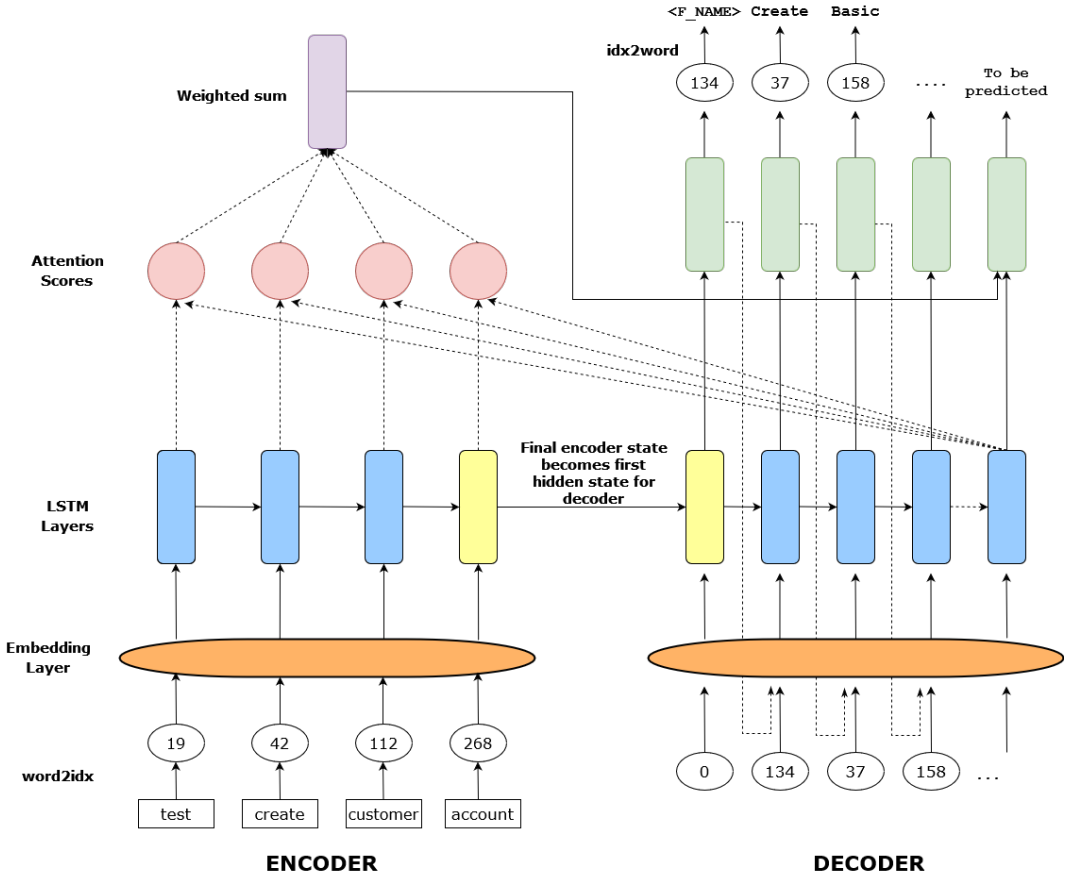


Figure 3.13: LSTM attention architecture with encoder and decoder blocks. The attention layer depicts the computation of attention score. The encoder outputs and the current decoder output are supplied to the attention layer

Attention: The attention class is implemented to compute the attention scores. There are three different kinds of score functions, a `dot`, `general` and `concat`. In our study, the scoring mechanism used is *general* (see section 2.5 for further explanation).

The attention class takes the hidden state of the decoder and the outputs from the encoder as input to compute the attention energies. First, a variable tensor of zeroes is created to store attention energies. Then, for each batch of encoder outputs, the attention score is computed by calculating the dot product between the encoder and hidden state. These energies are normalized to weights by using softmax and are resized to `1 x batch_size x seq_len`.

Decoder: The decoder is used to decode the sequences to generate the output sequence. The input to the decoder is the embedding of the input word from the

target sequence, the hidden state from the previous time step and the output from the encoder. The initial state for the decoder would be the final state from the encoder as shown in the Figure 3.13. The arguments to the decoder class are similar to that of the encoder class except that the output dimension i.e the size of the vocabulary of the target is used instead of the input dimension. First, the embedding layer is created by using `nn.embedding` on the input. Then, a dropout is applied. In addition, there is also a Linear layer that used to make the predictions from the last hidden state layer. After this, the attention weights are computed by passing the last hidden state and the outputs from the encoder to the attention method. The embedded input word and the output from the attention method are combined and run through the LSTM finally resulting the output, state of the hidden layer, and attention weights.

Training: After the model is built, a training function is implemented to start training it. To begin with, the models are initialized, optimizers and loss functions are set up. The loss is calculated by using masked cross entropy (explained in section 2.7) between the predicted and the ground truth tokens. The optimizer used is the Adam optimizer with a learning rate of 0.0001. The parameter `teacher forcing ratio` is used to decide which token will be used as the decoder's next input while training. The model is run for **4000** epochs. Similar to the previous LSTM model, the dimensions for the embedding and the hidden states are set with respect to the size of the vocabulary. The parameters set for the LSTM model with attention is shown in Figure 3.14.

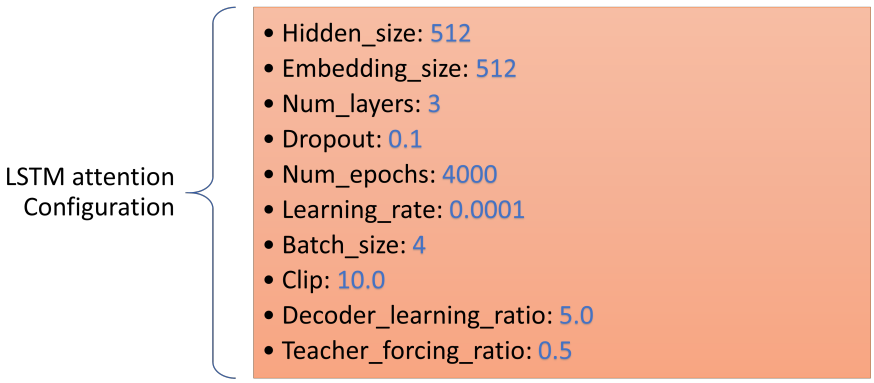


Figure 3.14: Hyperparameter configuration for LSTM attention model

At each iteration, the train method does the following tasks:

- feed the input sequence through encoder one word at a time
- Feed the decoder input, decoder hidden and the output from the encoder through decoder for each time step
- Store the decoder outputs in a pre-initialized tensor, say Y*
- compute the loss by using `loss.backward()`
- Use the optimizer to modify the parameters of the model

The returned loss for each epoch is summed and is averaged over the number of batches. Similar to the above models, the loss is printed for every epoch while the evaluation is done for every 100 epochs.

3.5.4 Training the model - GRU with attention

The third and the last model implemented in our study is GRU with attention. GRUs tend to be more computationally efficient and solve the problem of vanishing gradient. In addition, it is observed that GRU performs well when there isn't a larger training set. Since the amount of data used in our study is not large when compared to the amount of data used for training deep learning models, the idea was to make implement the GRU model using an encoder-decoder architecture where the encoder ´transforms the input sequence into a single vector which is used by the decoder to form a new sequence. To further improve the performance of the model, the attention mechanism is implemented, allowing the decoder to focus on particular elements of the input sequence. The model is build by taking inspiration from the work [10]. The model is implemented in three stages; the encoder, decoder and the attention layer for computing the attentions weights.

Encoder: Most of the architecture is similar to the LSTM attention model described in section 3.5.3, but with a few changes. The input to the encoder is a batch of word sequences, a long tensor of size `(max_len x batch_size)`. It outputs an encoding for each word, a float tensor of size `(max_len x batch_size x hidden_size)`. The embedding layer is created using `nn.Embedding` where the inputs fed are embedded for each word, with size `sequence_len x hidden_size`. This is resized to `sequence_len x 1 x hidden_size` to match the expected input of the GRU layer of `nn.GRU` [7].

The first hidden cell for the encoder is created automatically as a tensor of all zeros. Finally, the GRU encoder cell returns: output sequence of size `seq_len x hidden_size` and the last hidden state for each layer. The model's architecture is shown in Figure 3.15.

Decoder: The decoder's inputs are the input word, last hidden state and all encoder outputs. The initial decoder hidden state is the final encoder hidden state as shown in the Figure 3.15. First, the embedding layer is created by using `nn.embedding` on the input. Then, a dropout is applied. The hidden size and the number of layers are passed to the `nn.GRU` method. After this, the attention weights are computed by using the encoder outputs and the current hidden states. The embedded input word and the output from the attention method are combined and run through the LSTM finally resulting the output, state of the hidden layer, and attention weights.

Attention: The attention class is same as the attention class built for LSTM model. It takes the hidden state and the outputs of the encoder as input and compute the

[7] https://pytorch.org/docs/stable/generated/torch.nn.GRU.html?highlight=grutorch.nn.GRU

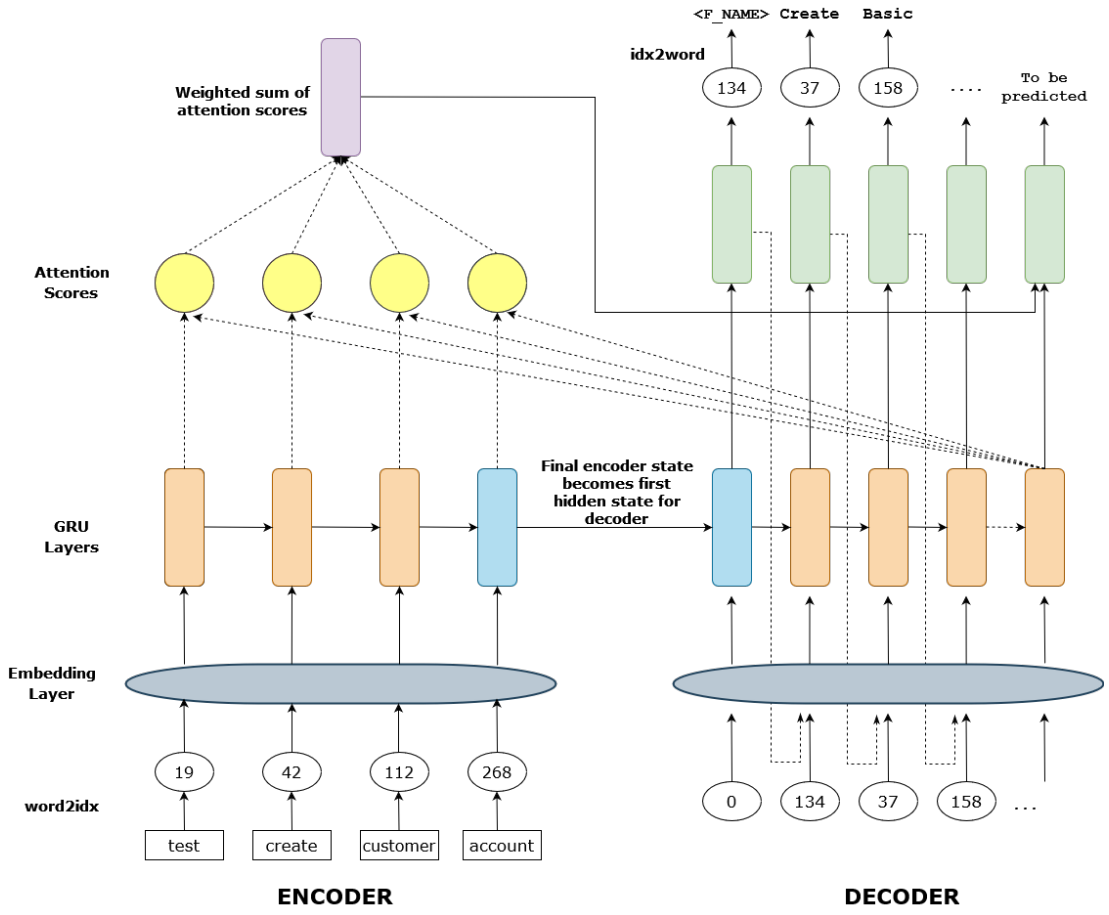


Figure 3.15: Gated Recurrent Unit with attention using the encoder-decoder architecture. The attention layer shown is by using the global attention model

attention scores. For each batch of encoder outputs, the attention score is computed by calculating the dot product between the encoder and hidden state. These scores are normalized to weights by using softmax and are resized to (`1 x batch_size x seq_len`). The attention class is implemented as below:

```
class Attention(nn.Module):
    def __init__(self, hidden_size):
        super(Attention, self).__init__()
        self.hidden_size = hidden_size
        self.attn = nn.Linear(self.hidden_size, self.hidden_size)

    def attn(self, hidden, encoder_outputs):
        max_len = encoder_outputs.size(0)
        batch_size = encoder_outputs.size(1)
        attn_energies = Variable(torch.zeros(batch_size, max_len))
        for b in range(this_batch_size):
            for i in range(max_len):
                attn_energies[b, i] = self.score(hidden[:, b],
                                encoder_outputs[i, b].unsqueeze(0))
        return F.softmax(attn_energies).unsqueeze(1)
```

```
    def score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        energy = hidden.matmul(energy.T)
        return energy[0][0]
```

Training: After the model is built, a training function is implemented to begin training it. To start with, the models are initialized, optimizers and loss functions are set up. The loss and the optimizers used are similar to the LSTM attention model implemented before. The loss is calculated by using masked cross entropy (section 2.7) between the predicted and the ground truth tokens. The optimizer is used is the Adam optimizer with a learning rate of 0.0001. The model is run for **4000** epochs. At each iteration, the train method does the following tasks:

- feed the input sequence through encoder one word at a time.
- Feed the decoder input, decoder hidden and the encoder outputs through decoder one time step at a time
- Store the decoder outputs in a pre-initialized tensor, say Y*
- compute the loss by using `loss.backward()`
- Use the optimizer to modify the parameters of the model

The returned loss for each epoch is summed and is averaged over batches. Similar to the above models, the loss is printed for every epoch while the evaluation is done for every 100 epochs. The parameters set for the GRU model is shown in Figure 3.16.

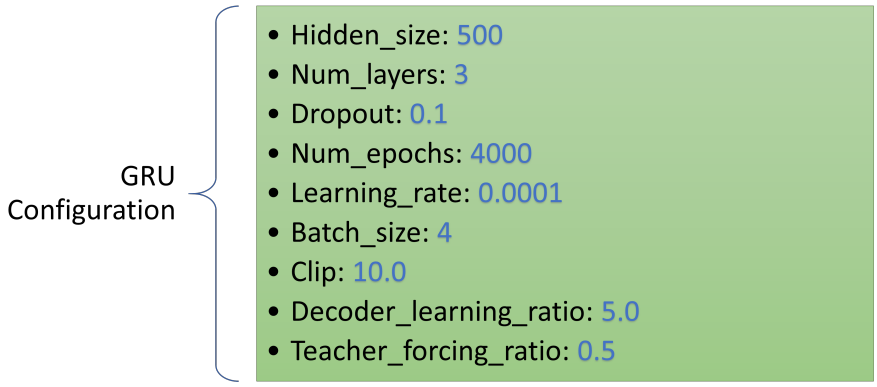


Figure 3.16: Hyperparameter configuration for GRU attention model

4
Results

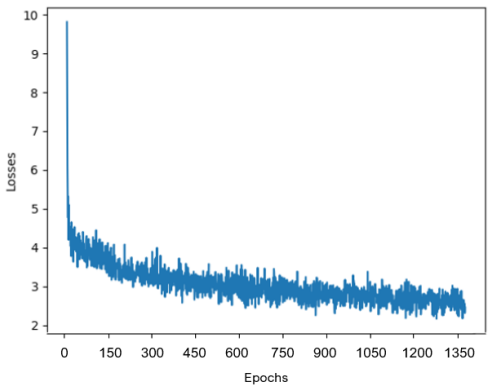
This chapter describes the results and interpretations from the models built in our thesis. The project analysis is done in two different ways. (a) Considering the data only from the monolithic application, (b) Experimenting on the acceptance test cases from the new test-driven application. The two methods of evaluation are described in sections 4.1 and 4.2 respectively. Each of the sections contains the discussion where the reasons for the obtained results are given and argued. A few examples of the results are also illustrated in the sections.

4.1 Evaluation - Method 1

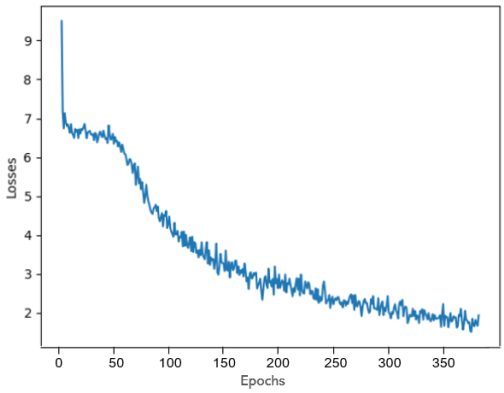
The method 1 describes the evaluation of the models only on the data of the monolithic application. The data prepared from the robot scripts belonging to the monolithic application is split into two parts - training and testing sets. The models are trained using the data from the training set and tested on the unseen data i.e. data from the testing set. Since the problem of the thesis is seen as a translation problem, a widely used metric to evaluate the language is BLEU. It evaluates the quality of text that is translated from one natural language to another. The description of BLEU is explained in section 2.10.1.

The training of all the three models is monitored using one metric which is by calculating the loss. The loss is calculated by using *masked cross entropy* as described in section 2.7. This criterion computes the entropy loss between predicted output and the original output. The loss plotted is the average masked cross entropy loss calculated for every 20 epochs. That is, for every 20 epochs of the model training, the computed loss till then is averaged. Generally, the decrease in loss indicates that the model is learning correctly and is confident in making the predictions. The less the loss, the more the model has learnt. The training loss for all the three models is depicted in the Figure 4.1.

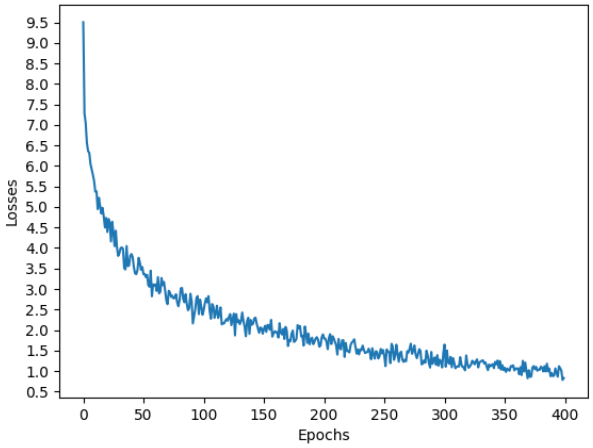
The plot illustrates the average losses obtained for each of the models after every 20 epochs. It can be observed that in all the three models, the loss is decreased significantly after very few epochs and later in order to reach to a much lesser value, it takes a lot of epochs. Due to computational constraints, all the models are trained with batch size of 4. The LSTM model is trained over 14000 epochs while the LSTM attention and GRU attention models are trained for 4000 epochs. Another observation that can be done is how far the loss has reached. It can be no-



(a) LSTM training loss



(b) LSTM Attention training loss



(c) GRU Attention training loss

Figure 4.1: Training losses for all the three models

ticed that the GRU model has achieved a lesser value of loss compared to the other two models. This means that GRU attention model has learnt a lot more and can make better predictions. The models are trained on Nvidia TITAN RTX GPU. On an average, training LSTM model took 16 hours while LSTM attention and GRU attention models took 35 hours each to achieve the loss as less as shown in the plots.

The feasibility of the approach in this thesis is evaluated by performing two experiments on the data for understanding the improvements and drawing conclusions. To understand the role of the data in training a deep learning model and to answer RQ-3, the thesis work involves two different experiments. **Experiment 1** involves the usage of only 40% of the data for training while **Experiment 2** uses 90% of the data. Both the experiments use the same hyper parameters and are executed for the same number of epochs. The summary of the results obtained is depicted in Figure 4.2. It can be observed that when the models are trained over more data, results achieved had higher BLEU score.

	Experiment 1			Experiment 2		
	Train ratio	Test Ratio	BLEU score	Train Ratio	Test Ratio	BLEU Score
LSTM	0.4	0.1	**1.2%**	0.9	0.1	**1.6%**
LSTM with attention			**7.5%**			**9.9%**
GRU with attention			**9.1%**			**12.3%**

Figure 4.2: Results of the two experiments done using the three models. The BLEU score achieved is evaluated on the test data

The results in the Figure 4.2 are computed on each instance of the testing set, i.e. the score is calculated for each test code, and averaged on the total number of elements in the testing set. It is evidently seen that the BLEU scores are much higher in the experiment 2 when compared to experiment 1. This means that the data plays an important factor in learning and predicting good results. With more data, more precise results can be achieved. It can also be observed that the in both the experiments, the GRU attention model has achieved a better BLEU score when compared to other two models. There has been a 36% increase in the BLEU score of GRU attention model from experiment 1 to 2. Similarly, there has been a 31% and 32% increase in the BLEU score for LSTM and LSTM attention models respectively when 50% of more data is used for training.

4.1.1 Discussion

The evaluation results from the method 1 say that the GRU attention model has achieved the best BLEU score of 0.12, an improvement of 31-32% over the the other two sequence-to-sequence models. In addition, the score attained was a reasonably good score when compared to the pre-trained code generation model implemented by Luiz Perez, Lizi Ottens et al. [30] who have achieved a BLEU score of 0.22. Although the pre-trained model was trained over approx. 500K python code scripts, considering the limitations of the current study, the score obtained can be considered a relatively good score. Figure 4.3 shows a sample example containing the input sequence, the original output and the predicted output from all the three models. As mentioned earlier, the predicted output from the models LSTM and LSTM attention models were not great and the same can be seen in the example. The output of the LSTM model was not at all good since the tokens generated were just being repetitive and not understandable. Even though LSTM can handle long sequences and have the capability to remember them, one of the reasons why the output isn't good is because of the input and output sizes. The length of the output in many cases is thrice or more than that of the input. This makes it difficult for the model to map and remember the long sequences and hence fail to predict correctly. It is the similar case with LSTM attention model except that the output is little better compared to normal LSTM. The reason could be the usage of the attention

mechanism and how the decoder focuses on to utilize the most relevant parts of the input sequence as explained in the methodology. However, after some period, the output becomes repetitive containing a lot of unnecessary information and the same can be seen in the Figure 4.3. A possibility for improvement is to make the test descriptions much more informative so that the model processes more information which can lead to better predictions.

test verifies packagemilestonecheck checks selected promotion package applied customer convert package points free service package change correctly		TEXT
<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 VAR_248 <NUM> Set Variable <NUM> VAR_2482 Set Variable <NUM> VAR_363 Create FME Contract With Service VAR_616 freeServiceId = <PARAM_VALUE> VAR_2479 Set Variable <NUM> Create Customer Package VAR_2479 VAR_1 VAR_2480 Set Variable <NUM> VAR_27 Check Package Milestone customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> coId = <PARAM_VALUE> <EOI>		ORIGINAL OUTPUT
LSTM	**LSTM Attention**	**GRU Attention**
<FUNC_NAME> VAR_616 VAR_616 VAR_616 Create Create Create Create Create Create Customer Customer Customer Customer Customer Customer Customer FBOCE1 FBOCE1 FBOCE1 8 FBOCE1 FBOCE1 M.SUBSCRIPT VAR_4821 VAR_4821 VAR_4821 VAR_4821 VAR_4821 M.SUBSCRIPT VAR_4821 VAR_4821 M.SUBSCRIPT VAR_4821 VAR_4821 VAR_4821 VAR_4821 VAR_4821 M.SUBSCRIPT VAR_4821 VAR_4821 VAR_4821 VAR_4821 VAR_4821 VAR_4821 VAR_4821	<SOI> <FUNC_NAME> VAR_2 Create Customer VAR_1 Get Customer ID BaseSimple VAR_233 Set Variable FBOCE1 VAR_670 Create Contract VAR_233 Set Variable <NUM> VAR_670 Active Contract VAR_670 VAR_1042 Add To Context VAR_1043 Add Random String To Context VAR_670 VAR_1042 Check Package Milestone Replace Input From Holders Context VAR_3382 Should Be Equal VAR_106 Add To Context rfIdCapture VAR_61 Add To Context rfIdCapture VAR_61 Add To Context rfIdCapture VAR_61 Add To Context	<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 Create FME Contract VAR_616 VAR_407 Get Contract ID VAR_5 VAR_2479 Set Variable <NUM> Create Customer Package VAR_247 VAR_1 Set Variable <NUM> VAR_27 Read Package Conversion customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone <EOI>	PREDICTED OUTPUT

Figure 4.3: Sample example comparing the original output with the predicted outputs from LSTM, LSTM attention and GRU attention models given the test description from experiment 2

The output from the GRU attention model seemed promising and as seen in the Figure 4.3, the predicted sequence matches most of the keywords from the original sequence. The model has learnt the format and the tokens properly and as a result, has provided much better results than the other two models. Though there was confidence in believing that the model would generate better results after looking at the training loss, it was promising to see the actual output. Figure 4.4 gives a much closer look at the comparison between the original output and the predicted output of the GRU attention model. Even though LSTM should in theory remember long sequences, GRU attention model outperformed it due to the ability of modelling long-distance relations. As mentioned earlier, the input to output sequence ratio is very large and therefore, it is difficult for the models to remember the information. GRUs work efficiently in these scenarios as it is designed flexibly with the input and output gates in the architecture. The example in the Figure 4.4 depicts a much more clear view to understand how many of the keywords were actually the same and how many were different. The tokens highlighted in *blue* are the ones that are

missing from the predicted sequence while the tokens highlighted in *yellow* are extra and considered as unnecessary information since they are not present in the original sequence. This also explains why there was a higher BLEU score for GRU attention model as there are many sequences in the test set that were closer to their original outputs.

Original Output	GRU Attention
<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 VAR_248 <NUM> Set Variable <NUM> VAR_2482 Set Variable <NUM> VAR_363 Create FME Contract With Service VAR_616 freeServiceId = <PARAM_VALUE> VAR_2479 Set Variable <NUM> Create Customer Package VAR_2479 VAR_1 VAR_2480 Set Variable <NUM> VAR_27 Check Package Milestone customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> coId = <PARAM_VALUE> <EOI>	<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 Create FME Contract VAR_616 VAR_407 Get Contract ID VAR_5 VAR_2479 Set Variable <NUM> Create Customer Package VAR_247 VAR_1 Set Variable <NUM> VAR_27 Read Package Conversion customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone <EOI>

Figure 4.4: Comparison of the original output and the predicted output from GRU attention model from experiment 2

To support the numbers shown in Figure 4.2, and to understand how much of the output has improved when the model was trained with more data, an example of GRU model is chosen and illustrated in Figure 4.5. It can be observed that the output of the GRU model from experiment 1 is lot different from the original output whereas the predicted sequence from experiment 2 is closer to the original sequence. The analysis was done to prove the fact that with more data, more accurate predictions can be achieved.

This analysis was done to answer the research question 3. This is also considered important for this study as one of the major limitations for this project is the availability of large amounts of data. By looking at the summary of the data statistics in the Figure 3.3, it is understood that the data is not sufficient to train a deep learning model and achieve higher accurate results.

test verifies packagemilestonecheck checks selected promotion package applied customer convert package points free service package change correctly		Text
<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 VAR_248 <NUM> Set Variable <NUM> VAR_2482 Set Variable <NUM> VAR_363 Create FME Contract With Service VAR_616 freeServiceId = <PARAM_VALUE> VAR_2479 Set Variable <NUM> Create Customer Package VAR_2479 VAR_1 VAR_2480 Set Variable <NUM> VAR_27 Check Package Milestone customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> coId = <PARAM_VALUE> <EOI>		Original output
GRU Attention – Experiment 1	GRU Attention – Experiment 2	Predicted Output
<SOI> <FUNC_NAME> VAR_313 Create Customer VAR_1 Get Customer ID DBaseSimple VAR_785 VAR_407 Get Contract ID VAR_121 VAR_2480 Set Variable <NUM> Add Optional PO VAR_670 VAR_1042 Replace Input CaptureHolders From Context VAR_3381 packageHolder Add Optional PO VAR_670 VAR_1042 PackId = <PARAM_VALUE> packageOfferingHolder CSDSFSRes VAR_1043 Check VAR_1046 VAR_43 action = <PARAM_VALUE> PackId = <PARAM_VALUE> PackId = <PARAM_VALUE> sn0:SMC.Package.CIRCNotAssgined Add To Context packagerfIdWrite Check VAR_34 VAR_1780 action = <PARAM_VALUE> <EOI>	<SOI> <FUNC_NAME> VAR_616 Create Customer VAR_1 Get Customer ID VAR_616 Create FME Contract VAR_616 VAR_407 Get Contract ID VAR_5 VAR_2479 Set Variable <NUM> Create Customer Package VAR_247 VAR_1 Set Variable <NUM> VAR_27 Read Package Conversion customerId = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone Check VAR_27 VAR_2480 action = <PARAM_VALUE> PackId = <PARAM_VALUE> Check Package Milestone <EOI>	

Figure 4.5: Comparison of the predicted outputs from GRU attention model between experiment 1 and experiment 2

4.2 Evaluation - Method 2

The method 2 involves an experiment to evaluate the models trained over the data from the monolithic application and test on the acceptance test cases from the newer application which does not contain the ground-truth target sequences. Since there are no original robot scripts implemented, the standard evaluation metrics cannot be used. Therefore, the evaluation for this method is done through human expert evaluation. A scoring mechanism is designed and is explained in the section 2.10.2. The analysis of the outputs is done for three different examples by three different experts from Ericsson who have the required domain knowledge with respect to the software application and also the robot framework. The metrics are scored on a scale from 0 to 5 (0 means the worst and 5 means the best).

A sample example of the predicted output for an input sequence is illustrated in Figure 4.6. The figure shows the input i.e. an acceptance test case description and the predicted output from the three models LSTM, LSTM attention and GRU attention. A general idea by looking at the predicted outputs is that the tokens in the LSTM model are not useful as they contain a lot of unnecessary and repetitive information. The LSTM attention model, though has a better output than LSTM, still contains unnecessary information and becomes repetitive after a few predicted tokens. The GRU attention model has a concise output with different keywords. The overview of the outputs seemed like a similar pattern when predicted with the evaluation method 1 above.

Read contract and verify the personalized values are stored properly on contract			Text
LSTM	LSTM Attention	GRU Attention	Predicted Output
<FUNC_NAME> VAR_2 VAR_2 Create Create Create Create Create Create Customer Customer Customer Customer VAR_232 VAR_232 VAR_232 VAR_232 VAR_232 VAR_13447 VAR_13447 rfIdHolder rfIdHolder rfIdHolder VAR_232 rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder rfIdHolder	<SOI> <FUNC_NAME> VAR_2 Create Customer VAR_5 Create Contract VAR_2 VAR_235 Get Contract ID VAR_5 Add Service To Contract VAR_5 codePub = <PARAM_VALUE> codePub = <PARAM_VALUE> Add Service To Contract VAR_5 codePub = <PARAM_VALUE> codePub = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE> Add Service To Contract VAR_5 effectiveDate = <PARAM_VALUE>	<SOI> <FUNC_NAME> VAR_2 Create Customer VAR_161 Get Customer ID VAR_2 VAR_5 Create Contract VAR_2 VAR_235 Get Contract ID VAR_5 VAR_1333 Get CFDS Code VAR_1 Check Package Request VAR_4 VAR_4 contractId = <PARAM_VALUE> <EOI>	

Figure 4.6: An example of the predicted outputs from LSTM, LSTM attention and GRU attention models given a acceptance test case

The generated outputs are reviewed by the test experts at Ericsson and are scored based on the designed metrics. Figure 4.7 gives a summary of the scores obtained. The scores illustrated are average scores for three different examples evaluated by three different experts i.e. the scores are first averaged for all the examples evaluated by one person for a model and then again averaged by the numbers of evaluators. The maximum score that can be attained is 5 and minimum is zero. It can be observed that the scores obtained are all below average showcasing that the GRU attention model is better when compared to the other two models.

	Content-Related		**Effectiveness-Related**	
	Adequacy	Conciseness	Usefulness	Suitability
LSTM	0.1	0.1	0.2	0.1
LSTM Attention	0.6	0.6	0.7	0.6
GRU Attention	0.9	1.5	1.7	1.2

Figure 4.7: Human evaluation scores averaged for each metric for each model

4.2.1 Discussion

The evaluation results from the method 2 say that GRU attention model has achieved better results compared to the other two models. This was also clearly seen when the outputs are manually seen as shown in the Figure 4.6. However, the scores attained are not high and are below the average line. There are multiple factors and

reasons that can be drawn by looking at the score sheet.

1. As stated as one of the limitations, the output is rated by different people with different expertise. So, it is possible that the scores from one person differ greatly from another. Therefore, the mean of the scores are calculated.

2. The scores are given only by looking at the input and the predicted sequence pertaining to that specific test case. Since test cases are written as a sequence of instructions, our study considers each instruction as a different test case and is inputted accordingly. This brings a confusion when the resulting output is viewed. Additionally, since there is no ground-truth available, it is much more difficult to analyse by just looking at test code once, as they are not generally implemented perfectly in one go.

3. The applications are different. As mentioned earlier, the model is trained over a different application and tested on another application. This results in lower relevancy of the output since the input and target data are from different domains.

To understand the reasons, the evaluators were also asked for their feedback. This helped in understanding more concrete reasons for the results obtained. A common feedback by all the experts is that the results are not completely relevant since the acceptance test cases tested belonged to another domain. The test cases for the domain to which the trained data belonged to, is still in the development stage at Ericsson and can be tested soon once it is completed.

One of the evaluators quoted "*The GRU_Attention is the one that had a better result in comparison with the other ones. But I think that the main question is that the data available for the analysis was not enough, I mean only Ericsson xxxx product repository information is not enough. The GRU_Attention somehow returned a good test structure without many useless things, and it has some logical order of the test case. But, to have a more general and precise result for an autonomous automation test generation, we need to have more sources of data, because the same expression can mean a lot of different things*". This means that the results obtained are not completely worst and there is scope for improvement if trained with more data and with more versatile data i.e. data belonging to different domains.

5
Conclusion

Analyzing test case specifications and manually developing the code is a time intensive and a resource intensive process. Automatic test code generation is one method for streamlining the software testing process and saving time and costs for the organization. In this thesis, an approach for automatically generating the test code in robot framework syntax from test case descriptions written in natural language is designed, implemented and evaluated. The study involves different experiments to understand the feasibility and usage of the approach. The work is done using natural language processing and deep learning techniques. Three different models are implemented to compare, analyse and evaluate the performance of the models over the data.

The data available at Ericsson from the older applications was leveraged to build a corpus and use it to train the models. Therefore, the data preparation stage played an important role for extracting information from existing robot scripts and use it for training the models. This approach has helped in obtaining good results and therefore has helped to answer our RQ-1.

The evaluation involves performing different experiments to obtain answers for the defined research questions. The method 1 of the evaluation involves the training and testing of the models only on the data from the monolithic application. Based on the results obtained, it is concluded that the GRU attention model has resulted in performing the best by achieving a BLEU score of **0.12**. Despite the fact that the problem is seen as a sequence translation problem, the data used is still in a programmatical context and therefore is complex to map natural language text to a code containing natural language elements. However, it is found that the model treats the programming language as another domain specific natural language to produce the results. This answers the RQ-2 proving that a code can be generated in robot syntax by building models using NLP and deep learning techniques. The evaluation method 2 involves the testing on the acceptance test cases where the evaluation is done manually by human experts. The GRU attention model outperformed the other two models, however, the scores achieved are still below average. Various reasons have been discussed with the major reason being that the domains are not similar. It is concluded that there is a need for more variety of data in order to attain more general and precise results.

Another experiment involved performing the training with different amounts of data so as to compare the results obtained. It has been observed that the models trained

with only 40% of the data did not produce good results when compared to the models trained with 90% data. The results are shown and described in the previous chapter. This answers the RQ-3, showing that the model performance is affected with the amount of data available.

The final research question (RQ-4) provided a thought of sensibility to our study. This question was raised while receiving the feedback on our evaluation method 2. From the meetings with the test experts helped in understanding how the generated results could help them. The generated keywords could be reused while implementing test code rather than implementing them from scratch, thereby, minimizing their time and efforts.

To conclude the thesis, it is clear that using Natural Language Processing techniques and Deep Learning methods to automatically produce test code from a test description expressed in natural language has potential and possibility. The proposed method was successful in predicting appropriate keyword functions that are required in robot scripts. It has the potential to reduce the developer's time spent on implementing test code, by considerable amounts. Further enhancements and developments to the study can aid the testers by relieving them from the manual operation of implementing test code from scratch and provide a more automated way for the testing process thereby minimizing the time and costs for the organization.

Bibliography

[1] Robot framework. https://robotframework.org/.

[2] Robot framework user guide. https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

[3] Imran Ahsan, Wasi Haider Butt, Mudassar Adeel Ahmed, and Muhammad Waseem Anwar. 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*, ICC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Allohvk. Cross entropy and classification losses — no math, few stories, and lots of intuition, 2021.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[6] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Nat2testscr: Test case generation from natural language requirements based on scr specifications. *Science of Computer Programming*, 95:275–297, 2014. Special Section: ACM SAC-SVT 2013 + Bytecode 2013.

[7] Valentine Casey. *Software Testing and Global Industry: Future Paradigms*. 01 2009.

[8] Gobinda G. Chowdhury. Natural language processing. *Annual Review of Information Science and Technology*, 37(1):51–89, 2003.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[10] Fei Ding. Sequence to sequence and attention. https://github.com/ifding/seq2seq-pytorch, 2018.

[11] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August 2016. Association for Computational Linguistics.

[12] Li Dong and Mirella Lapata. Language to logical form with neural attention, 2016.

[13] dProgrammer lopez. Rnn, lstm gru, 2019.

[14] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3, 2020.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning, 2016.

[16] Charon Guo. Sequence to sequence learning with neural networks. `https://charon.me/posts/pytorch/pytorch_seq2seq_1/`, 2020.

[17] Anne Mette Jonassen Hass. *Guide to Advanced Software Testing*. Artech House, Inc., USA, 2008.

[18] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[19] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. Correlating automated and human evaluation of code documentation generation quality. *ACM Transactions on Software Engineering and Methodology*, 05 2022.

[20] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context, 2018.

[22] Kiprono Elijah Koech. Cross entropy loss function, 2020.

[23] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[24] Infolks Pvt Ltd. Recurrent neural network and long term dependencies, 2019.

[25] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[26] Simo Makinen and Jürgen Münch. Effects of test-driven development: A comparative analysis of empirical studies. volume 166, 01 2014.

[27] Jyoti Shetty Neha S Batni. A comprehensive study on automation using robot framework. International Journal of Science and Research (JISR), 2020.

[28] Sajad Norouzi, Keyi Tang, and Yanshuai Cao. Code generation from natural language with less prior and more monolingual data, 2021.

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[30] Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. Automatic code generation using pre-trained language models, 2021.

[31] Renita Priya, Xinyuan Wang, Yujie Hu, and Yu Sun. A deep dive into automatic code generation using character based recurrent neural networks. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 369–374, 2017.

[32] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[33] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages, 2021.

[34] Alzahraa Salman. *Test Case Generation from Specifications Using Natural Language Processing*. PhD thesis, 2020.

[35] Stanislav Stresnjak and Željko Hocenski. Usage of robot framework in automation of functional test regression. In *ICSEA 2011*, 2011.

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[37] Sahar Tahvili. *Multi-Criteria Optimization of System Integration Testing*. PhD thesis, 12 2018.

[38] Ben Trevett. Pytorch seq2seq. `https://github.com/bentrevett/pytorch-seq2seq`, 2021.

[39] David Turner, Moonju Park, Jaehwan Kim, and Jinseok Chae. An automated test code generation method for web applications using activity oriented approach. pages 411–414, 09 2008.

[40] Sunitha Edacheril Viswanathan and Philip Samuel. Automatic code generation using unified modeling language activity and sequence models. *IET Software*, 10(6):164–172, 2016.

[41] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel Briand. Automatic generation of system test cases from use case specifications: an nlp-based approach, 07 2019.