UCLouvain

epl

École polytechnique de Louvain

# Deep Learning for Software Engineering

Author: **Gildas Mulders**
Supervisor: **Siegfried Nijssen**
Readers: **Olivier Goletti, Kim Mens, Siegfried Nijssen**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

# Abstract

Automated Program Repair (APR) consists in automatically fixing buggy codes. In this research field, some state-of-the-art approaches rely on Deep Learning to perform Sequence to Sequence learning, which aims at designing models mapping sequences from one domain to another, such as from French to English for instance. In particular, the `SequenceR` technique shows very promising results in terms of the variety of bugs for which it is able to find perfect fixes. In this context, this master's thesis investigates the limitations and possible improvements of the `SequenceR` method. More precisely, we mainly focus on trying to find out whether or not this approach might benefit from a richer representation of code, hereby taking into account the structural information held by Abstract Syntax Trees. This line of research is explored by quantifying the impact of adding features alongside the words of source code as input to `SequenceR`-like models. Our experiments on several datasets suggest that APR performances are hardly affected by augmenting the input data with annotations. Yet, this finding only applies to the specific features studied and does not rule out the utility of such an approach. Finally, our analysis of the `SequenceR` limitations indicates that the expected fixes of around 50% of a test set of 4711 samples cannot be predicted by the model because it simply cannot generate some of the expected tokens. Therefore, we identify that the main challenge to be tackled by this APR technique is the Out Of Vocabulary problem.

# Acknowledgements

First, I would like to express my sincere gratitude to my supervisor, Prof. Siegfried Nijssen, for his continuous guidance, his valuable advice and comments as well as the time he devoted to this master's thesis. I also thank him for giving me access to appropriate computing resources in order to conduct this work.

I would also like to thank Olivier Goletti and Prof. Kim Mens for accepting to read this master's thesis.

Finally, I would like to thank my family for their support and proofreadings. With special thanks to Tanguy, for helping me understand Docker, Dounia and Cyril, for their insightful discussions about the process of research, and Victoria, for her continuous support, enthusiasm and patience.

Computational resources have been provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie Bruxelles (CÉCI) funded by the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11 and by the Walloon Region.

# Contents

**Introduction** **1**
General context . . . 1
Motivations . . . 1
Goals and structure . . . 2

**1 Machine learning background** **3**
1.1 Theoretical concepts . . . 3
1.1.1 Deep Learning . . . 3
1.1.2 Sequence to Sequence Learning . . . 5
1.1.3 Extensions to the basic seq2seq model . . . 9
1.1.4 Geometric Deep Learning . . . 13
1.2 Existing application: OpenNMT . . . 14
1.3 Conclusion . . . 16

**2 Automated program repair** **17**
2.1 State of the art . . . 17
2.2 SequenceR . . . 18
2.2.1 Data . . . 18
2.2.2 The model . . . 20
2.2.3 Results . . . 21
2.2.4 Implementation . . . 21
2.3 Conclusion . . . 21

**3 Analysis of the SequenceR dataset** **22**
3.1 Qualitative analysis . . . 22
3.2 Quantitative analysis . . . 24
3.2.1 Unknown source tokens . . . 24
3.2.2 Unknown target tokens . . . 25
3.2.3 Difficult samples . . . 26
3.3 Conclusion . . . 28

**4 Improvements** **29**
4.1 Word features . . . 29
4.1.1 Features definitions . . . 29
4.1.2 Numerical features in OpenNMT-py . . . 31
4.1.3 Using the original dataset . . . 32

4.2 Word embeddings: Word2Vec . . . . . . . . . . . . . . . . . . . . . . . 33
4.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**5 Testing** **34**
5.1 Testing process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

**6 Difficulties Encountered** **37**
6.1 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.2 Technologies used . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.3 Problems of substance . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**Conclusion** **40**
Lessons learned . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
Further works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**Bibliography** **42**

**Appendix** **45**

# Introduction

## General context

Writing reports is a long process and we often have to take some time to read everything we wrote and check for mistakes in the syntax or in the meaning of our own words. In fact, for long reports such as this master's thesis, we can proofread ourselves several times and still find mistakes after a few passes. Imagine a world, where text editors would not only check spelling and basic grammar, but also the meaning, the very content of reports. This sounds like a handy feature, saving us from sending silly things to our teacher, boss or supervisor. That embarrassing situation has a tougher equivalent when dealing with source code rather than reports. The goal of source code is to be executed, not just read. For instance, it is some source code that handles our privacy on the web, that secures our financial transactions and that now drives some of our cars. Sending a report containing a silly mistake is like deploying code with a vulnerability or a bug in it, however the consequences can be much more harmful in the second case. Moreover, we often practice the languages we write reports in all day long while this is not the case for programming languages. Therefore, we are even more likely to make mistakes in code. A tool enabling to automatically correct source codes would greatly help beginning programmers to learn and senior programmers to work in a better and more efficient way.
Automated Program Repair (APR) is the exciting emerging field aiming to design such handy tools for source code (Goues et al. (2019)). The general principle of APR approaches is to take a source code as input, to locate potential bugs in it and to propose fixes for these bugs. There exist different ways to tackle that task, the newest one being based on Deep Learning (DL).

## Motivations

DL solutions are especially relevant because they thrive on huge amounts of data. In the framework of source code analysis, tremendous open source projects can be found on version control platforms such as Github (Chen and Monperrus (2018), Tufano et al. (2019)). These resources can all be leveraged by machine learning algorithms. In this context, this master's thesis starts from an existing DL APR solution called `SequenceR` (Chen et al. (2019)). This approach is based on what is called Sequence to Sequence (seq2seq) learning (Sutskever et al. (2014)), which is

what Google Translate relies on. `SequenceR` uses seq2seq learning to 'translate' buggy lines into fixed lines. This approach has already shown promising results. However, this technique treats code as a sequence of words, therefore not taking its structural information into account. Source code is indeed by nature easier to parse than natural language. It is meant to be compiled and executed and can be organised in a tree structure called parse tree or Concrete Syntax Tree (CST).
This weak representation of code can be improved thanks to the existence of several tools and technologies such as the followings.

1. Seq2seq learning frameworks such as OpenNMT (Klein et al. (2017)) that support annotating data with features.

2. Libraries such as Spoon (Pawlak et al. (2015)) or Javaparser (Hosseini and Brusilovsky (2013)), allowing the creation, traversal and manipulation of the Abstract Syntax Tree (AST), which is a simplified CST, of Java code.

3. Machine learning techniques like Word2Vec (Mikolov et al. (2013)) and its existing implementation (Řehůřek and Sojka (2010)), which are able to automatically learn meaningful vector representations of words.

4. Geometric Deep Learning, a new branch of Deep Learning with increasing support (Fey and Lenssen (2019)) that takes graphs as inputs.

The nature of source code and the availability of these technologies are motivations to explore potential improvements to the `SequenceR`.

# Goals and structure

This master's thesis analyses the limitations of the APR state of the art `SequenceR` technique. It also investigates ways to mitigate these limitations mainly by proposing richer representations of code and studying the effect of such representations.
This work is structured as follows. Chapter 1 lays the foundations of the proposed methods by presenting the Deep Learning and Sequence to Sequence learning approaches. Chapter 2 dives into the Automated Program Repair state of the art and details the `SequenceR` application. Next, chapter 3 analyses the limitations of the `SequenceR` from a qualitative and quantitative point of view. Then, chapters 4 and 5 respectively propose potential improvements and discuss the effects of these improvements. Finally, the difficulties encountered during this work are summarized in chapter 6 before concluding the thesis.

# Chapter 1

# Machine learning background

This chapter summarizes the machine learning background of this master's thesis. It first explores the theoretical concepts and then goes through an existing framework implementing those concepts. In addition to being a fully worked out example of the theory, this framework was used for this master thesis' work.

## 1.1 Theoretical concepts

### 1.1.1 Deep Learning

`Deep Learning` is a field of machine learning involving neural nets. This master's thesis' application uses deep learning in the context of supervised learning, that is, we have some input data and corresponding target labels, and we want to train a model to be able to assign a fitting target label to new input data. The way deep learning approaches this problem is by taking a vector representation of the input and passing it through several different cells before producing an output. Figure 1.1 illustrates a simple neural net where the green circles represent the elements of one input vector and the blue circles represent cells. The cells are usually grouped into layers and each cell of a layer takes all - in the case of fully connected layers - or some of the outputs of the previous layer as inputs, this is represented by the arrows on figure 1.1. The number of cells and the number of layers are parameters of the model. The notion of deep learning refers to neural nets with 2 or more hidden layers. In the simplest form of these blue cells, what happens is that a weighted sum of all the inputs is performed. The result of that linear combination is then passed through a non linear activation function.

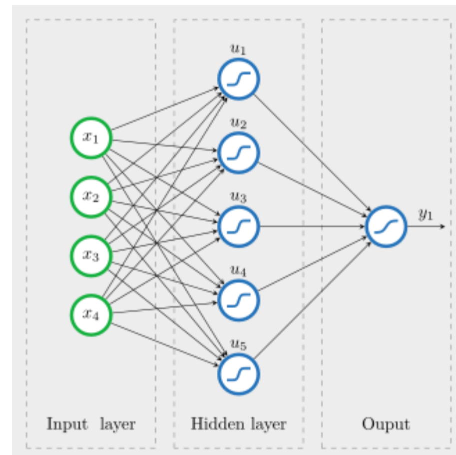


Figure 1.1: A simple neural net, taken from Pitie (2020).

The mathematical equation behind these blue cells is the following:

$$u_i = f(w_0 + \sum_{j=1}^{4} w_{ij}x_j), \tag{1}$$

or in matrix notation:

$$\mathbf{u} = f(\mathbf{b} + \mathbf{W}\mathbf{x}). \tag{2}$$

In equation 1, $u_i$ stands for the output of one of the blue cells of figure 1.1, $f$ is a non-linear function such as $tanh$ or $ReLU$, $w_{ij}$ is a weight and $x_j$ is an element of the input vector. Equation 2 is the same as equation 1 in matrix notation, with $\mathbf{W}$ being a matrix of weights, $\mathbf{u}$ the vector of hidden states, $\mathbf{b}$ the bias vector and $\mathbf{x}$ the input vector.
The key to having a useful model is to train it, that is, learn the appropriate weights $w_{ij}$ that should be put in each cell in order to produce the expected results from the given inputs. Training a neural net is an iterative process. It works like this:

1. First, the weights need to be initialized with some values; a simple solution is to assign them random values.

2. Then, the training inputs are passed through the neural net, therefore producing a result $y_k$ for each of the $k$ different inputs.

3. A loss is computed, that is, a value expressing how far each of the predictions $y_k$ are from the known correct values associated with each of the $k$ inputs.

4. The weights are updated in function of the loss, this step is called back-propagation, because we propagate the update information from the very last cell of the neural net to the first ones. This update relies on gradient descent and can be optimized using different approaches and meta-parameters.

5. After updating the weights, we start back at step 2 and we loop like this until reaching a loss that is low enough.

There exist different loss functions such as the cross-entropy or the Kullback-Leibler divergence. Most of the time, the loss is not directly computed over the predictions from the whole dataset, but rather over a batch of data of a certain size. That `batch size` is a parameter fixed by the programmer. This means that one iteration of the algorithm mentioned earlier does not represent a pass through the whole training dataset.
Imagine we have a dataset of 1000 samples and a batch size of 100. That means that, at each iteration, the loss is computed from 100 predictions and the weights are updated accordingly. The algorithm will therefore need 10 iterations before having gone through 1 `epoch`, that is, 1 pass through the whole training dataset. The training time is often determined by either a number of iterations or a number of epochs fixed by the user.
A frequent problem of machine learning models is over-fitting, that is, when the model gets very good at predicting the data it was trained to predict, but very bad

at predicting new data. To avoid that, training is often performed through a cross-validation loop. The idea is to split the available data in three parts: a training, a validation and a testing set. The training set is the data that actually goes through the training process previously described. The validation set is used to optimize the eventual meta-parameters of the model and finally, the test set is meant to provide a fair and unbiased evaluation of the model's performances. During this testing phase, multiple metrics are monitored as one metric alone is not enough to characterize how well a model is trained. In practice, the metrics used at this step are different than the loss functions, because loss functions are not easily interpreted. Therefore, we often prefer metrics such as the perplexity, the accuracy, the precision, the recall...

This gives a basic understanding of how neural networks work. They are powerful tools, as Hornik (1991) proved that

> a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units[1]

However, in practice, it is useful to use multiple hidden layers, because deeper neural networks allow to model some aspects of the data that are more complex more easily. Such models are also harder to train, one of the reasons being the problem of vanishing gradients. Vanishing gradients essentially mean that some cells of the network are not getting updated anymore, so the model is stuck. This is a problem that also occurs with Recurrent Neural Networks, which are mentioned in the next subsection.

### 1.1.2 Sequence to Sequence Learning

`Sequence to sequence learning` is a specific type of deep learning introduced by Sutskever et al. (2014) that, as its name indicates, uses sequences as inputs and target data. An example application of sequence to sequence learning is machine translation. In this master's thesis, just like in machine translation, we are working with sequences of words.

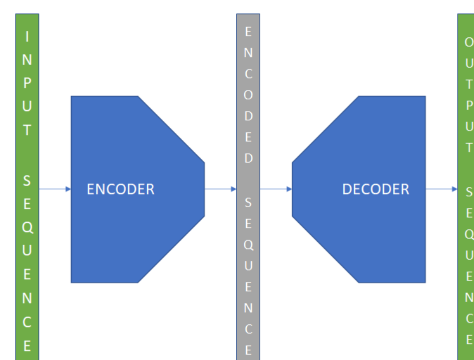


Figure 1.2: Encoder-decoder architecture.

#### Encoder-decoder architecture

The top level view of a sequence to sequence model is an encoder-decoder architecture such as the one represented in figure 1.2. The encoder and decoder, here presented as blue boxes, are actually neural nets.

[1]quoted from Pitie (2020)

The goal of the encoder is to encode the input sequence, that is, turn it into a vector of a certain size. That size is a parameter of the model which is referred to as the `hidden dimension`.
The role of the decoder is then to take this vector representation and use it to build an output sequence.
This type of model can be used to translate a sentence from a language into another or to teach a model to respond to questions for instance.

### Recurrent Neural Networks

Now, let us dive a little bit more into what happens inside these black boxes. As we are working with sequential data, we use Recurrent Neural Net cells instead of the basic cells described by equation 2. Here is how RNN cells work.
For every word $t$ in an input sequence, an equation similar to equation 2 is used. An extra term is introduced to account for the previous word $t-1$ in the sentence for the model to be able to learn not only from the word itself, but also from the context around it. So we end up with an equation like this:

$$\mathbf{u}_t = f(\mathbf{W}\mathbf{x}_t + \mathbf{b} + \mathbf{U}\mathbf{u}_{t-1}) \tag{3}$$

With $f$, $\mathbf{W}$ and $\mathbf{b}$ still referring to the same things as in equation 2. The extra term contains $\mathbf{U}$, which is another weight matrix and $\mathbf{u}_{t-1}$, which is the hidden state of the previous word in the sequence, computed similarly. Finally, $\mathbf{x}_t$ is the $t^{th}$ word in the sequence and $\mathbf{u}_t$ is the hidden state of word t. Note that the weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$ are shared across all the words. The output of the encoder is the hidden state $\mathbf{u}_n$ of the last word of the sequence. The first cell of the decoder takes the encoder's output as its previous hidden state and a special start of sequence token as input $\mathbf{x}$. Next, in addition to being propagated from cell to cell, the hidden states of each of the decoder's cells are used to perform a prediction for the next word in the output sequence. That prediction is also fed as the input $\mathbf{x}$ of the next cell until a special end of sequence token is predicted.

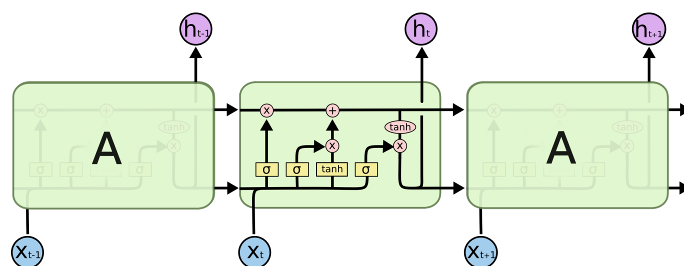


Figure 1.3: LSTM network, figure by Olah (2015).

In practice, RNNs are often replaced with more elaborate recurrent cells because they are otherwise too hard to train. This difficulty comes from the problem of the vanishing gradients. Since each cell is using the previous one as an extra input, the path of the gradients can get very long as if we were training a very deep network.

Long Short Term Memory[2] and the more recent Gated Recurrent Unit[3] cells solve that by introducing parallel paths, such that there is a shortcut for the gradients during back-propagation. That shortcut also allows the network to learn long term dependencies between the different words forming a sequence. Figure 1.3 shows that the inside of an LSTM cell is essentially a small neural sub-network. More details on how this works can be found in Olah (2015).

### From words to vectors

RNN and its derivatives allow building richer models that take advantage of the sequential nature of data, but all these models still expect numbers and vectors as input. So how do we go from a word to a vector?
The first step is to build a vocabulary, that is, making a list of the different words one might use in a sentence. In practice however, this vocabulary is limited to a certain size, which is a parameter of the model. Then, very much like a dictionary, each word is matched, not to a definition, but to a vector. This vector representation of words is called the `embedding` and the embedding dimension is another parameter of the models. The simplest way to understand this mechanism is by using one-hot encoded vectors. For a vocabulary of $n$ words, each word is associated to a different vector of size $n$ filled with exactly $n-1$ times the value 0 and only one time the value 1. The only difference between each of these vectors is therefore simply the position of that value 1. This creates an easy association between words and vectors, but that representation is not very rich. The true embedding of a word can be seen as a new cell that is inserted at the very start of the encoder and which is exactly the same as the simple cells described by equation 2 without the non-linear functions. This cell therefore holds a trainable weight matrix and bias vector. Moreover, because of the nature of the one-hot encoded vectors that this cell is taking as inputs, it can be understood that exactly one line from the weight matrix and one value from the bias vector is linked to exactly one word. A well trained model would then have learned weights that link each word to a meaningful representation so that related words are close to each other in the vector space for instance.
Although this is an approach that works, it is not optimal because it takes quite some space to store a lot of useless zeros. This model allows for a clear understanding of how embeddings work, but there exist better approaches using lookup tables.

### From vectors to words

On the other end of the encoder-decoder architecture, the decoder has the opposite task of turning vectors into words. The decoder is a recurrent neural net very similar to the encoder. It takes two inputs: the embedding of the previous word it generated and another vector, coming from the encoder, which informs it about the context. The decoder is then usually terminated by a fully connected layer such as

[2] Hochreiter and Schmidhuber (1997)
[3] Cho et al. (2014)

the one illustrated on figure 1.1. Its final output is a vector of the size of the target vocabulary, giving each different word a probability of being generated.

**Search strategies :** There are 3 different strategies to choose which word should actually be generated.

| Time step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| A | 0.5 | 0.1 | 0.2 | 0.0 |
| B | 0.2 | 0.4 | 0.2 | 0.2 |
| C | 0.2 | 0.3 | 0.4 | 0.2 |
| <eos> | 0.1 | 0.2 | 0.2 | 0.6 |

Figure 1.4: Greedy search.

| Time step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| A | 0.5 | 0.1 | 0.1 | 0.1 |
| B | 0.2 | 0.4 | 0.6 | 0.2 |
| C | 0.2 | 0.3 | 0.2 | 0.1 |
| <eos> | 0.1 | 0.2 | 0.1 | 0.6 |

Figure 1.5: Different path leading to better results.

Figure 1.6: Ineffectiveness of greedy search, illustration by Zhang et al. (2021).

The first, most basic one, is a `greedy search`: at the end of each step of the decoder, we just take the word with the highest probability of being generated. The problem with that is that it does not lead to the optimal solution. The optimal solution is defined as the sequence with the maximum conditional probability meaning the sequence that maximizes the product of each of its words' probabilities. As explained by Zhang et al. (2021), each output of the decoder influences the rest of the sentence.
Figure 1.6 shows a toy example where the target vocabulary is composed of { A, B, C, <eos> }. On figure 1.4, the greedy approach is applied and yields a final conditional probability for the sentence of $0.5 \cdot 0.4 \cdot 0.4 \cdot 0.6 = 0.048$. However, on figure 1.5, a different word is chosen at step 2 of the decoder, therefore changing the probability distribution for the following steps. The result then becomes $0.5 \cdot 0.3 \cdot 0.6 \cdot 0.6 = 0.054$, which is better than the greedy search.
Another strategy is the `exhaustive search`, which consists in trying everything and then finally keeping the sequence with the optimal score. This approach can be very time consuming.
The third and final strategy is a compromise between greedy search and exhaustive search and is called `beam search`. It depends on a parameter $k$. The idea is to keep the best $k$ sequences at each step. So at step 1 of the decoder, the $k$ words with the best probabilities are kept. Then, at step 2, each of these $k$ words can be combined with another word of the vocabulary to form a 2-word sequence. So for a vocabulary of $n$ words, we have to keep the $k$ best-scoring 2-word sequences from $k \cdot n$ candidates. And this goes on until all candidates are terminated by the `<eos>` special token, meaning the end of the sequence.

### 1.1.3 Extensions to the basic seq2seq model

The next three subjects are not essential to the definition of the notion of sequence to sequence learning, but they are useful extensions. These extensions allow to cope with some limitations of the base seq2seq models that are particularly relevant when the data consists of big code chunks.

#### Dealing with long sequences

An old challenge of sequence to sequence learning is to learn long range dependencies between words. Even though RNNs give information about what came before in the sequence at each step, this information strongly focuses on what came right before the current word and tends to forget the impact of earlier words. That problem is partly answered by LSTM thanks to the shortcut they introduce on the top line of figure 1.3. But this solution still presents two big limitations:

1. The contextual information embedded by the output of the encoder only contains knowledge about what comes before in the sequence and none about what comes after. This problem is solved by bi-directionnal LSTMs.

2. The encoder's hidden state passed from the last of its LSTMs to the first LSTM of the decoder as context is a vector of fixed length. This vector is supposed to hold all the information needed for the decoder to interpret the whole input sequence. Such an information is hard to push into a fixed length vector, especially with long sequences. This problem is solved by the Attention Mechanism.

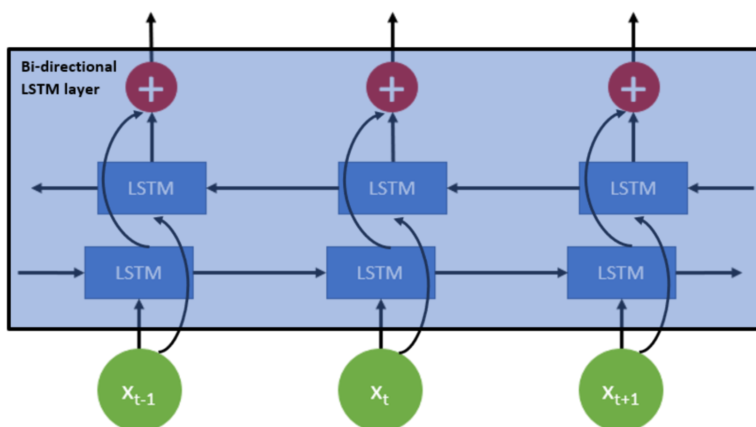


Figure 1.7: Bi-directional LSTM layer.

**Bi-directionnal LSTMs :** Figure 1.7 shows that, as explained by Brownlee (2017), bi-LSTMs allow the gathering of the surrounding context of each word simply by combining two LSTM layers, one parsing through the sequence in forward direction and the other one backwards. This means that each word ends up with

two hidden states, one embedding the context of the words that come before the current word in the sequence, and the other one containing the context of the words that come after. The red disks from figure 1.7 then represent the merging operation of these two vectors. There exist different approaches when it comes to the nature of this merging operation. For instance, the PyTorch[4] library just concatenates the vectors, but the Keras[5] library gives its users the choice between concatenation, sum, multiplication and average. Although one could just extract both vectors from the concatenation and then merge them in yet another creative way.
Bi-LSTM layers do not really define a new type of cell, but are rather a wrapper for a particular way of combining existing cells.

**Attention Mechanism :** The attention mechanism was proposed by Bahdanau et al. (2016). It essentially consists in adding a component between the encoder and the decoder. The input of this component is the encoder's hidden states, that is, the vectors representing each word of the input sentence. The intuition is then that, at each step of the decoder, the attention gives more importance to certain parts of the sentence. We can better visualize the role of the attention by looking at the following example of an English to French translation:

What are you doing ? $\rightarrow$ Que fais-tu ?

At the first step of the decoder, it would make sense for the model to learn that the relevant input word to look at while trying to generate the first word "Que" is "What". Then at the second step of the decoder, the focus should be put mostly on the word "doing", and probably also on the words "are" and "you" to generate the French word "fais". Finally, at the last step of the decoder, the attention mechanism would focus on the word "you" to generate "tu".

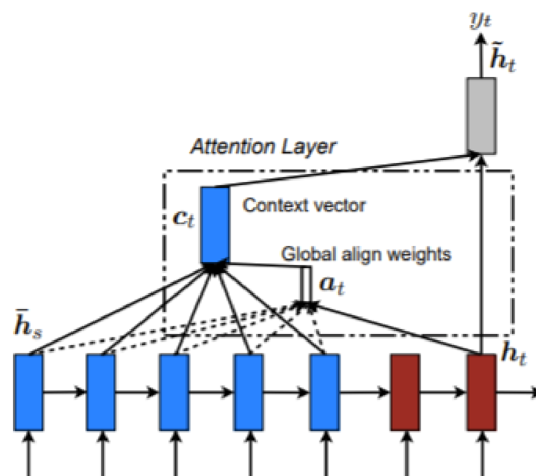


Figure 1.8: Attention mechanism, illustration from Luong et al. (2015).

[4]Paszke et al. (2019)
[5]Chollet et al. (2015)

As explained by Loye (2019), the attention mechanism does that by going through the following process:

1. Calculating alignment scores: At each step of the decoder, each encoder's hidden state will be given an alignment score, which essentially represents its relative importance. In figure 1.8, $\overline{\mathbf{h}}_s$ is the encoder's hidden states, $\mathbf{h}_t$ is the previous decoder's hidden state and $\mathbf{a}_t$ is the alignment scores vector. To compute this alignment score, $\mathbf{h}_t$ and $\overline{\mathbf{h}}_s$ both go through a different linear layer[6]. Then, the results are merged by adding the decoder's weighted vector to each of the encoder's weighted hidden states vectors. These merged vectors then go through a non-linear activation function. Finally, the output of this function is itself weighted by a third trainable matrix yielding the so-called alignment scores vector, $\mathbf{a}_t$.

2. Normalizing the alignment scores: the previously obtained alignment scores are then normalized by a softmax function[7]. At this step, we end up with a vector of size equal to the number of encoder's hidden states, that is, equal to the number of words in the input sentence. That vector sums up to 1.

3. Building the context vector: finally, the vector that will be used by the decoder's LSTM as a context, called context vector and referred to as $\mathbf{c}_t$ on figure 1.8, is computed by making a weighted sum of all the encoder's hidden states with the weights being the normalized alignment scores.

Loye (2019) also mentions a variation of the attention mechanism proposed by Luong et al. (2015) and represented by figure 1.8 which mainly differs in two ways:

- The way it combines the decoder's previous hidden state and the encoder's hidden states to produce the alignment scores.

- The position in the neural net where the context vector is used. In Luong's case, the context vector never goes through the decoder's LSTM. It is in fact combined to its output to be fed to the final feed-forward neural net which is in charge of choosing which word to generate.

It is worth noting that this attention mechanism is very efficient, so much that a new kind of model emerged with the famous "Attention is all you need" paper by Vaswani et al. (2017). This paper defines a model called the "Transformer model" which relies only on attention mechanisms and does not use any recurrent cell. This model is getting more and more popular as it appears to outperform the classical seq2seq models.

---

[6]that is, a layer without activation function, which is equivalent to a simple product by a trainable weight matrix

[7]$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}}$ with $x_i$ an element of a vector $\mathbf{x}$ of size $n$.

### Dealing with OOV words: the copy mechanism

Another major problem arises from the fact that everything the model is ever able to predict is restricted to a certain vocabulary. When building a machine translation model, the first task is to establish one vocabulary for the input language, and another one for the target language. What we call the "Out Of Vocabulary Problem" occurs when a model has to predict a target sequence that uses words not listed in its target vocabulary. For that reason, vocabularies always contain a special `<unk>` token that is generated when the model estimates that the most likely next word in the sentence is unknown to it.
A frequent cause of the OOV problem is the case of proper nouns, because it is impossible to list all proper nouns. For instance if you need to perform such a translation:

Marie Dupont likes waffles. → Marie Dupont aime les gauffres.

At best, a good machine translation model would learn to predict something like this:

Marie Dupont likes waffles. → `<unk>` `<unk>` aime les gauffres.

Which is sad, because most often, proper nouns can just be copied from a language to another without changing. The copy mechanism[8] can improve this situation by simply copying words from the input sequence. The way it works is that each predicted word can either be generated, that is, chosen from the target vocabulary, or copied from the input sequence. In practice, the target vocabulary becomes $T \cup I \cup \{\texttt{<unk>}\}$ with $T$, the original target vocabulary and $I$ the list of tokens present in the input sequence. In the end, the model gives probabilities to each of the words of this new joint vocabulary. These probabilities are computed for each word as the sum of each word's probability of being generated from $T$ with its probability of being copied from $I$. On the one hand, the generation probability is obtained the same way any seq2seq model computes its final probabilities. On the other hand, the copy probability is computed from the encoder's hidden states and the previous decoder's hidden state, very much like the attention mechanism works. The exact computation details are explained by Walsh (2020) or in the original Gu et al. (2016) paper. The main intuition is that the encoder's hidden states ideally hold information about the word embedding as well as the word's location in the sequence, which is how the model learns to choose which words to copy.

### Better word embeddings: the Word2Vec model

Finally, Word2Vec models do not really solve any problem or limitation of seq2seq architectures, but they can improve the step of going **From words to vectors** described in section 1.1.2. These models rely on neural networks to build word embeddings. This technique was proposed by Mikolov et al. (2013) and comes in two shades: C-BOW and skip-gram. Here is how Word2Vec models work.

[8] proposed by Gu et al. (2016)

As illustrated on figure 1.9, the CBOW model, which stands for continuous bag of words is a neural net that learns to predict a word from its surrounding context. The embedding of each word is then the weight vector that multiplies its context words. The Skip-gram is similar but works the other way around: for each word, the task of the model is to give each of the vocabulary words a probability of being in the word's context.

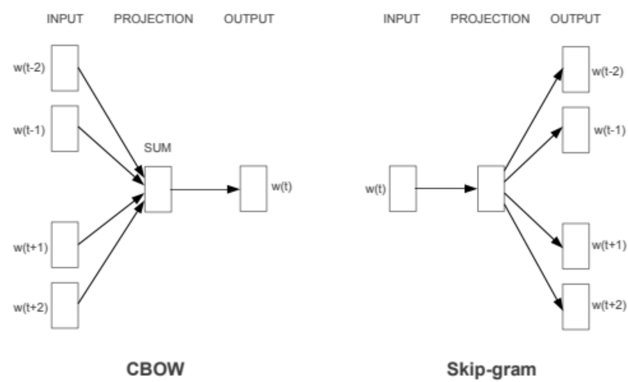


Figure 1.9: Word2Vec architectures, from Mikolov et al. (2013).

That list of probabilities is then the word's embedding.
These models work very well as they are able to map words to the vector space in a way that related words are close to each other.

### 1.1.4 Geometric Deep Learning

Geometric Deep Learning is a term first introduced by Bronstein et al. (2017). This field aims at generalizing Deep Learning approaches so that they can be applied to non-euclidean data. Moreover, as explained in Bronstein et al. (2021), it provides a unified mathematical understanding of a range of different Deep Learning architectures such as Convolutional Neural nets, Transformers, Recurrent Neural nets, etc.
The easiest way to understand the notion of non-euclidean data is through the idea that the shortest path between two points is not necessarily a straight line. Graphs can represent such data as a set of nodes linked by edges. Concrete examples of data best represented as graphs are molecules, social networks and source code. An illustration of a graph representing a social network is shown in figure 1.10 where people are nodes and their relationships with each other are edges. Note that the AST of source code is a tree, which is a particular case of graph without any cycles in it.

Figure 1.10: Graph representation of a social network, from Taylor (2019).

Graphs could be fed directly to classical Deep Learning models by being represented by their adjacency or incidence matrices, but these are not ideal representations because they arbitrarily define an order of the nodes. This means that the model would then associate a different weight to a given node depending on this arbitrary ordering.
Geometric Deep Learning provides solutions to handle that kind of data. The most

basic approaches rely on message-passing, which means that each node simply sends the same information to all of its neighbours. However, there is now a range of different and more complex Geometric Deep Learning techniques. An overview of these techniques can be found in Tong (2021).

## 1.2 Existing application: OpenNMT

OpenNMT is

> an open source ecosystem for neural machine translation and neural sequence learning[9]

started by Klein et al. (2017). In a nutshell, it can be seen as a Python library that wraps the main components of a sequence to sequence model. Figure 1.11 represents the typical kind of model that can be built from this library. The red part is the encoder, consisting in an embedding layer followed by a recurrent layer. The plus sign represents an attention mechanism, the blue part is the decoder and the yellow part is the final feed-forward neural net where the actual prediction happens. The OpenNMT implementation comes in two shades, namely `OpenNMT-py`, implemented with PyTorch[4], and `OpenNMT-tf`, powered by Tensorflow[10]. This master's thesis relies on `OpenNMT-py`.

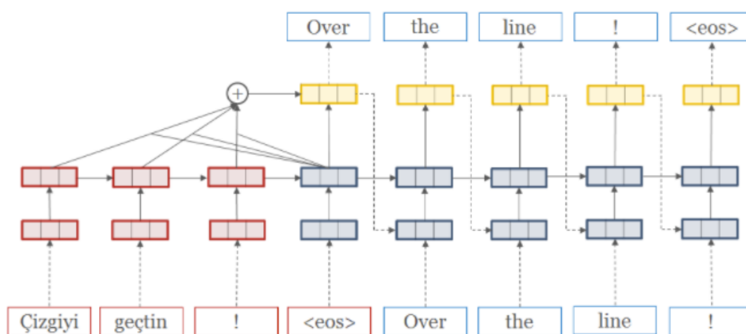


Figure 1.11: Encoder-Decoder illustration from `https://opennmt.net`.

OpenNMT-py also comes with a set of Python scripts allowing to train and use complex models specified by a range of comprehensive parameters directly from the command-line. That is actually the most documented way to use this project. For the most part, the range of available command-line arguments were flexible enough for this master thesis' purposes, so that is mainly how it was used. However, a few ideas required some tweaking of these scripts, so it is useful to understand how they work. A typical application would use 3 scripts: `preprocess.py`, `train.py` and `translate.py`.

[9] definition from `https://opennmt.net`

[10] Abadi et al. (2016)

### Preprocessing

The most important role of the first script is to build vocabularies for the input and for the target. This is not a huge script, but it already comes with a few parameters such as the size of the vocabulary and whether to merge the target and input vocabularies or not. If one wishes to use the copy mechanism, it should also be specified. Each sentence's vocabulary will then be extended with the sentence's out of vocabulary words, so that these can be copied later.

### Training

This is the main script. We can break it down to two main parts, namely the model creation and then the training itself.

**Model creation :** OpenNMT-py as a library contains classes wrapping a whole encoder and a whole decoder. To create a model from Python code, here are the steps a programmer would go through:

1. Create an embedding layer: the main parameter of the OpenNMT Embedding class is the embedding dimension, that is, the size of the vectors representing the words.

2. Create an encoder object: the encoder class allows specifying the number of layers, the size of the hidden states and the type of cells to use. Interesting cell type options are recurrent cells such as LSTM, bi-LSTM and GRU or Attention mechanisms to build a Transformer model.

3. Create a decoder object: the decoder class has the same parameters as the encoder class, but additionally allows to use an attention mechanism and/or a copy mechanism between the encoder and the decoder.

4. Specify the loss function.

Apart from the parameters of each of these steps, the process to build an encoder-decoder architecture is always the same. The `train.py` script therefore starts by doing exactly that with parameters chosen by the user and specified from the command-line.

**Model training :** The training process of a neural network is explained in section 1.1.1. The parameters left to the user for this part include the number of steps during which the model should be trained, the batch size, the optimizer[11] and which GPU should be used if any. If only one GPU is used, the number of steps correspond to the number of iterations, else, batches are processed in parallel. 3 GPUs would therefore make 3 iterations per step.

[11] this determines how the weights are updated at each iteration, see Ayushi (2021) to get a glimpse of optimizers' state of the art.

### Translation

Finally, the `translate.py` script can generate predictions from a given dataset without looking at its labels. This is useful to evaluate the actual performances of the model without the bias of over-fitting. The main parameters include the beam size and whether the copy mechanism is used by the model, for the same reason as in the preprocessing step. The obtained predictions can then be compared to the expected labels to assess the model.

### Interesting features

In addition to this flexible model creation and training, OpenNMT also provides interesting additional features.

**Adding features to data** OpenNMT gives the possibility of providing more information alongside the words. This functionality is called "word features". This can be useful in many ways. Let us take the example of a machine translation model again. It could be interesting to feed the model information about the grammar of a sentence for the model to better understand and then translate it. That is known as Part Of Speech Tagging, which means that each word is annotated with a tag. For instance a subject will have a "s" tag, a verb a "v" tag and a complement a "c" tag. The way to feed this extra information in the OpenNMT framework is by simply appending the `FFE8` unicode character followed by the appropriate tag to each word of the dataset. Multiple word features can then be appended to each word separated by the special `FFE8` character.
The `preprocess.py` script will then build a vocabulary for each different word feature. In the POS tagging toy example, the vocabulary would be {s, v, c}. The `train.py` script would learn embeddings for these features. Then, when the encoder reads a sentence, it will not only get the embedding of the word it is reading as input, but a longer vector consisting in the concatenation of the word embedding with its word features' embeddings.

**Specifying initial weights** Another useful OpenNMT-py functionality is the ability to start the training with initial weights. If this functionality is not used, the model starts its learning process with random weight matrices. Taking advantage of pre-trained vectors is known as transfer learning and is a powerful idea since training a model can be very time and resource consuming.

## 1.3 Conclusion

In this chapter, we started by giving a basic understanding of Deep Learning in general, then diving more precisely into Sequence to Sequence learning and its different components. Next, we introduced the notion of Geometric Deep Learning before presenting the OpenNMT framework. This gives an understanding of the main concepts and technologies that are used and discussed throughout this work.

# Chapter 2

# Automated program repair

Automated Program Repair - APR - refers to the task of fixing programming mistakes without human intervention. It is also useful to optimise code that is already functional. APR can be used to help programmers, provide educational support or bridge the gaps of security systems. This chapter first provides a summary of the existing APR techniques and then focuses on one approach called the `SequenceR`.

## 2.1 State of the art

As shown in Monperrus (2018), listing all the papers related to the field of APR, this is a growing research topic that has been approached in different ways. We distinguish three main types of approaches, illustrated in figure 2.1: heuristic repair, constraint-based repair and learning-based repair.

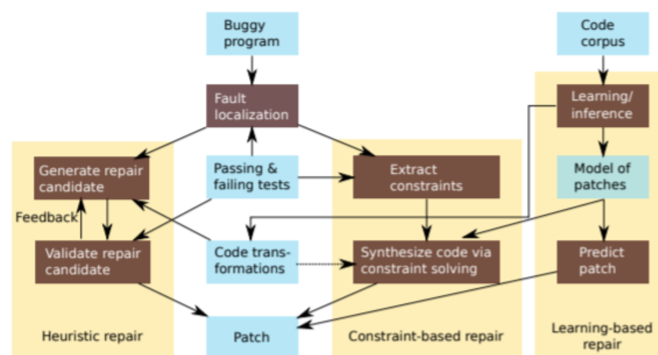


Figure 2.1: Overview of repair techniques, from Goues et al. (2019).

These different approaches are explained in Goues et al. (2019) and can be summarized as follows.

**Heuristic repair:** Candidate patches are generated by changing parts of the Abstract Syntax Tree of the code. These candidates are then evaluated by a test suite which amounts to a fitness function to maximize.

**Constraint-based repair:** First, constraints are extracted from the code using symbolic execution for instance. Symbolic execution aims at determining which constraints are imposed on the inputs for certain parts of the program to be reached. In this approach, candidates are generated to solve these constraints, so the key is to extract the right constraints.

**Learning-based repair:** This approach relies on machine learning to learn relevant code transformations. Some techniques just use machine learning to improve one of the steps of heuristic or constraint-based repair approaches while others use it as an end-to-end solution. This master's thesis uses sequence-to-sequence learning as an almost end-to-end solution for APR.

The process of APR can be broken down in two independent steps. First comes the fault localization and then comes the actual patch generation. This master's thesis focuses on the second step only and therefore considers that the line of the bug is known.

## 2.2 SequenceR

The main basis of this master's thesis is the work done by Chen et al. (2019) who wrote a paper in which they describe the so-called `SequenceR`. The `SequenceR` is a technique for automatic bug-fixing relying on Sequence-to-Sequence learning. In addition to their paper, Chen et al. (2019) also made a full implementation and dataset available through GitHub[1].

### 2.2.1 Data

Instead of taking a natural language sentence as input, the `SequenceR` takes Java code. More precisely, the task of the `SequenceR` is to take a code with exactly one buggy line in it and to produce a fix for that line. The `SequenceR` also requires that this buggy line should be in a method. For this to work, feeding the buggy line only would not be enough as the model needs more context to get an understanding of the way the code works and to propose a fix. That context is referred to as the Abstract Buggy Context (ABC). The abstract buggy context is built the following way:

Listing 2.1: Initial code.

```
1  public class Main {
2      public double add(double a, double b) {
3          return a + b;
4      }
5      public double sub(double a, double b) {
6          return a - b;
7      }
8      public static void main() {
9          Main mainObj = new Main();
10         int a = 3.14;
11         double b = 5.2;
12         double c = mainObj.add(a, b);
13     }
14 }
```

Listing 2.2: Abstract Buggy Context.

```
1  public class Main {
2      public double add(double a, double b) {
3      }
4      public double sub(double a, double b) {
5      }
6      public static void main() {
7          Main mainObj = new Main();
8          int a = 3.14;
9          double b = 5.2;
10         double c = mainObj.add(a, b);
11     }
12 }
```

[1] `https://github.com/KTH/chai`

1. First, everything except the buggy class definition, the class methods definitions and the buggy method body are removed from the code. This is done by a Java script, identifying these interesting parts by manipulating the Abstract Syntax Tree of the code created with the Spoon[2] library. Listing 2.1 shows an example of code with a bug at line 10 (variable **a** should be a double instead of an int). Listing 2.2 shows the abstract buggy context of that code as it would be after this first step.

2. Then, the remaining code is tokenized by a Python script, that is, each meaningful unit is space separated. So for instance:

   *System.out.println("Hello");* → *System . out . println ( "Hello" ) ;*

3. The Buggy line location is marked by surrounding it with "<START_BUG>" and "<END_BUG>" before the full tokenized code is put on a single line. In practice, steps 2 and 3 are done together by the same Python script, the result of these two steps is illustrated by listing 2.3.

Listing 2.3: Tokenized ABC.

```
public class Main { public double add ( double a , double b ) { } public double sub ( double a , double b ) { } public static void
main ( ) { Main mainObj = new Main ( ) ; <START_BUG> int a = 3.14 ; <END_BUG> double b = 5.2 ; double c = mainObj .
add ( a , b ) ; } }
```

4. Finally, a Perl script truncates that line of code to a thousand tokens. This is done while trying to keep twice as many tokens before the buggy line as after the buggy line.

Once this is done for each code in the dataset, this data is ready to be fed to a classical seq2seq model.
In practice, the `SequenceR` comes with a prepared dataset assembled from the work of Tufano et al. (2019) and from the CodRep competition[3]. These datasets were both created by collecting commits from open-source projects coded in Java and hosted on Github.

- Tufano et al. (2019) describes an approach similar to `SequenceR`, except that instead of dealing with real code, it takes and outputs normalized code where all the functions and variables are replaced by names such as "METHOD_X" assigning a different number "X" to each method. This experiment was performed on a diverse dataset named "Bugs2Fix" and containing codes with bugs on more than one line, others with bugs outside of a method, etc. After some filtering, the SequenceR was left with 17% of the nearly 100 000 samples Bugs2fix dataset.

- The Codrep competition

[2] A library allowing to create, parse and manipulate Java ASTs proposed by Pawlak et al. (2015).
[3] Chen and Monperrus (2018)

> aims at encouraging scientific and technological progress in the domain of machine learning over source code.[4]
>
> In this contest, participants are given a code with exactly one buggy line as well as a fixed line to replace that line. Their task is to create a model that is the best at deciding which line of the buggy code should be replaced by the fixed line. Five datasets[5] originating from different open-source projects are provided to the participants to help them train their models, which are then evaluated using another, hidden, dataset. Each of these 5 datasets is made of two parts: a Tasks folder and a Solutions folder. The tasks folder holds several numbered files, each containing a fixed line as first line, then a space, and then a full code in which this fixed line must be placed. The Solutions folder holds the same numbered files, this time simply containing a number corresponding to the line index of the corresponding task file where the fixed line should go. The creators of this competition are the same people who built the `SequenceR`.

These datasets have been merged and the abstract buggy context of each code has been built. The corresponding fixed lines have been tokenized to be used as targets during the training phase. Finally, the resulting dataset of abstract buggy contexts was split in 3 parts for cross-validation: a training set of 33798 samples, a validation set of 1780 samples and a test set of 4711 samples.

### 2.2.2 The model

Figure 2.2 summarizes the structure of the `SequenceR` model. It is an encoder-decoder with two layers of bi-LSTMs in the encoder, two layers of LSTMs in the decoder, an attention mechanism and a copy mechanism. The copy mechanism is essential since the vocabulary of Java code is infinite. Indeed, programmers can name their classes, methods and variables however they choose. A vocabulary of 1000 words is built from the most frequent tokens in the abstract buggy contexts with the aim of getting all the reserved keywords of the programming language plus a few of the most frequent variable names. Only these 1000 words will have a trained embedding vector. The embedding dimension and the hidden dimensions of the encoder and the decoder are all set to 256.

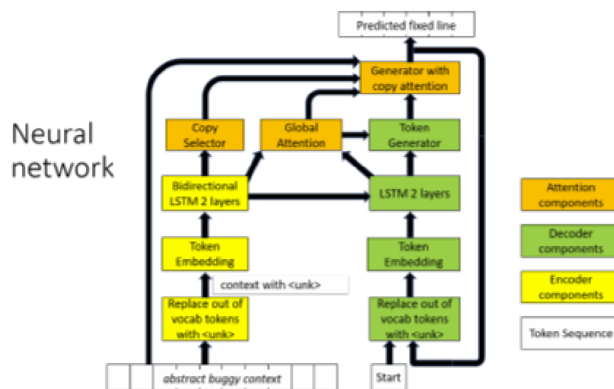


Figure 2.2: The SequenceR model, from Chen et al. (2019).

[4] quoted from Chen and Monperrus (2018)

[5] `https://github.com/KTH/CodRep-competition`

### 2.2.3 Results

The Golden model was obtained after 10000 iterations with a batch size of 32. It was used to predict the test set using a beam search of size 50, meaning that for each of the 4711 abstract buggy contexts of the test set, the model generated 50 candidate fixes. As this type of neural networks introduces a lot of randomness with initial embeddings and the way samples are combined to form batches, training a model with the exact same parameters can produce varying results. Therefore, 10 models were trained for this set of parameters. The final result was that, at best, the Golden model of the `SequenceR` found the exact expected fix within 50 proposed fixes for 950 out of the 4711 cases. The mean result for the 10 runs of this set of parameters is 859 perfect fixes out of 4711 samples. An ablation study was operated on this Golden model to see if certain training parameters could improve the performances, but it proved not to be the case.
In Chen et al. (2019), a qualitative study of the results shows the kind of patches found by the model. Generally speaking, the model appears to be able to perform token deletion, addition and replacement. Some practical examples show that the `SequenceR` is able to change the called method, the calling object, to add arguments, etc...

### 2.2.4 Implementation

The `SequenceR` is implemented using OpenNMT-py through its command-line interface. It relies mainly on 3 shell scripts:

- `sequencer-train.sh` : simply calling the OpenNMT-py `train.py` script with the Golden model parameters.
- `sequencer-test.sh` : generating predictions from the test set by calling the `translate.py` script.
- `sequencer-predict.sh`: a more elaborate script allowing an end to end use of a model. Using that script, a user can feed a Java source code file along with the number of a buggy line to get one or multiple fixes for that line. This script creates an abstract buggy context, passes it through a previously trained model doing a beam search of the desired width, and finally "un-tokenizes" the predictions to fit them in the original code, therefore creating as many new files as there are predictions.

## 2.3 Conclusion

In this chapter, we showed an overview of the APR state of the art and then focused on the `SequenceR` datasets and model. This model brought a lot of the concepts introduced in chapter 1 to the field of APR. It also relies on the OpenNMT implementation described earlier. Knowing how the model works as well as its performance, we can now study its limitations.

# Chapter 3

# Analysis of the SequenceR dataset

This chapter analyses the dataset used to evaluate the `SequenceR`. Two approaches have been considered: a quantitative analysis and a qualitative one. This analysis focuses on what the `SequenceR`'s Golden model was not able to predict, since the opposite was already discussed in the original paper by Chen et al. (2019). The aim of this process is to find ideas for improvement.

## 3.1 Qualitative analysis

This analysis is done by looking at the outputs of a Python script. This script looks at the Golden model's predictions as well as the test dataset and outputs 10 randomly selected samples for which the correct fixes were not found in the predictions. It actually outputs 4 lines: the abstract buggy context of the sample, the buggy line, a "<BUG2FIX>" delimiter and then the expected fix. This script is run multiple times to extract and classify interesting observations. Three main phenomenons can be observed.

- Some types of corrections listed in the qualitative approach of the `SequenceR` paper as the kind of mistakes that the model can fix are not found by the model. This means that even though the model proved to be able to perform the required code transformation, that particular transformation is not part of the top 50 predictions of the beam search. This is a situation that might be improved by providing some extra information to the model. Listings 3.1 and 3.2 illustrate this issue. In red, we can see the buggy line and the corresponding expected fix for that line is in green. For convenience, the full Abstract Buggy Contexts can be found in the appendix.

  Listing 3.1 shows an example where the called method had to be changed. The expected method name is not in the vocabulary, but it is used close to the buggy line in the abstract buggy context, so it could be copied. We can notice that the called method is replaced by another method with a very similar name, so it might be interesting to give the model a notion of how similar the different words are.

Listing 3.2 shows examples where the model failed to change the binary operators. It might be interesting to tag the different types of tokens so that the model learns the list of operators for instance.

Listing 3.1: Method change failure.

```
gl . glUniformMatrix2x4fv ( location , count , transpose , value ) ;
gl . glUniformMatrix4fv ( location , count , transpose , value ) ;
```

Listing 3.2: Operator change failures.

```
if ( len > ( bits . length ) ) {
if ( len >= ( bits . length ) ) {
fieldType . setOmitNorms ( ( ( fieldType . omitNorms ( ) ) || ( ( boost ) != 1.0F ) ) ) ;
fieldType . setOmitNorms ( ( ( fieldType . omitNorms ( ) ) && ( ( boost ) == 1.0F ) ) ) ;
```

- A lot of cases are actually impossible to predict by the model, because the expected fixed line uses words that are not in the vocabulary nor in the corresponding abstract buggy context. We call this problem the true OOV problem, because it is an OOV problem defined on the merged vocabulary of the ABC and the fixed-length vocabulary. We can distinguish different causes for that problem.
  - First, listing 3.3 illustrates a case where the correct fix uses a string representing a path to a file which was never used anywhere else in the code. This category of problem can be generalized as the fact that fixes sometimes use new values[1].

Listing 3.3: Target using an unkwown string value.

```
Pixmap pixmap = new Pixmap ( files . internal ( "data/badlogicsmall.jpg" ) ) ;
Pixmap pixmap = new Pixmap ( files . internal ( "data/stone2.png" ) ) ;
```

  - Another case of true OOV problem occurs when the fix contains the creation of an object defined in some imported libraries. This case is illustrated by listing 3.4, where the class *JsonModelLoaderTest* is not defined nor used in the Abstract Buggy Context.

Listing 3.4: Target using an unknown object.

```
GdxTest test = new BatchRenderTest ( ) ;
GdxTest test = new JsonModelLoaderTest ( ) ;
```

[1]that is, strings or numbers.

- Finally, even though some usage of external classes can be found in the ABC, fixes sometimes need an unknown method or attribute from these classes. Listing 3.5 illustrates that problem with the unknown *GENERIC* attribute and *setField* method.

Listing 3.5: Target using unknown methods or attributes.

```
final String executor = Names . CACHED ;
final String executor = Names . GENERIC ;
requestBuilder . field ( "document.simple" ) ;
requestBuilder . setField ( "document.simple" ) ;
```

This true OOV issue is further discussed in the quantitative analysis section.

- Finally, some cases are technically possible to fix with the model, but the correct line would be really hard to generate in practice because the buggy line is either very long or very different from its fix. Listing 3.6 shows an example of a very long buggy line. The expected fix simply consists in adding a "null" argument to a function call, but the length of the buggy line makes it hard for the model to guess which part should be altered. This issue is also discussed quantitatively in the next section.

Listing 3.6: Long buggy line.

```
return new JsonIndexQueryParser ( new Index ( "test" ) , EMPTY_SETTINGS , newMapperService
( ) , new org . elasticsearch . index . cache . filter . none . NoneFilterCache ( index ,
EMPTY_SETTINGS ) , new org . elasticsearch . index . analysis . AnalysisService ( index ) , null
, null , "test" , null ) ;
return new JsonIndexQueryParser ( new Index ( "test" ) , EMPTY_SETTINGS , newMapperService
( ) , new org . elasticsearch . index . cache . filter . none . NoneFilterCache ( index ,
EMPTY_SETTINGS ) , new org . elasticsearch . index . analysis . AnalysisService ( index ) , null
, null , null , "test" , null ) ;
```

## 3.2 Quantitative analysis

This approach relies on another Python script which contains a set of functions, each producing different graphs and statistics about the data.

### 3.2.1 Unknown source tokens

The first concern investigated pertains to the tokens that are part of the abstract buggy contexts but not part of the vocabulary. These words can therefore be generated by the copy mechanism. As mentioned in the original `SequenceR` paper, the copy mechanism is used intensively by the model. However, such Out Of Vocabulary tokens are all replaced by <unk> by the model, so they are mainly distinguished by their contexts.

The graph on figure 3.1 reports the results of the following experiment. For a value $n$ varying between 0 and 5000, a vocabulary of size $n$ is built by taking the $n$ most frequent words in the training set. Then, all the OOV words of the test set are replaced by the <unk> token. Finally, the <unk> percentages of each abstract buggy context of the test set are computed and averaged. This experiment shows that, as the vocabulary size grows, the percentage of <unk> tokens in the abstract buggy contexts decreases and reaches a plateau a bit lower than 20%. This means that, at best, one out of five tokens of the abstract buggy contexts is out of the vocabulary on average. This represents a big lack of information.

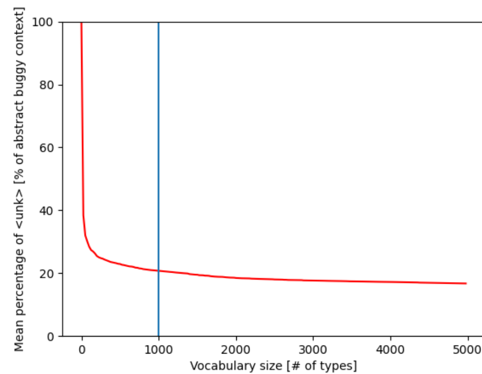


Figure 3.1: Vocabulary size vs percentage of <unk> in the test ABCs.

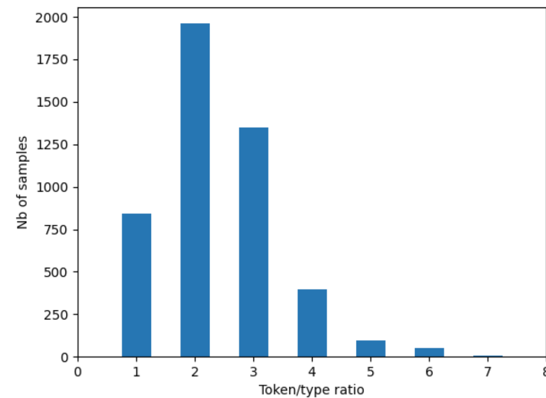


Figure 3.2: Type/token ratios for OOV words in the test abstract buggy contexts.

The graph in figure 3.2 illustrates a consequence of that lack of information. It shows the distribution of mean token/type ratios of the OOV words in the test abstract buggy contexts. Tokens and types are terms coming from Natural Language Processing. The number of tokens is the number of words in a sentence and the number of types is the number of different words in the sentence. For instance, in the sentence 'life is life', there are three tokens and two types. This ratio can therefore be interpreted as the number of times each different OOV word is used. The results show that, in most samples, OOV words are used multiple times. This information is lost by the model because the model replaces all these OOV words by the <unk> token, therefore masking the fact that some words appear multiple times.

### 3.2.2 Unknown target tokens

The most important discovery about the dataset is that around 50% of the test set is actually impossible to predict with the `SequenceR` model. The exact number is that 2316 out of the 4711 test samples are impossible to predict by the model. This limitation comes from the fact that the only tokens that can be generated by the model must either be in the fixed size vocabulary or in the abstract buggy context. We can distinguish two particular kinds of problems. On the one hand, for some samples, the expected fix uses a not so frequent token which does not appear in the

corresponding Abstract Buggy Context but which appears in other samples of the training set. On the other hand, some fixes use tokens that simply never appear in the whole training set. While the latter problem is truly impossible to solve using the same data and techniques, the former problem could be reduced by building a bigger vocabulary. This idea can be tested by building vocabularies of different sizes from the training set, exactly the way it was done in the previous subsection's experiment, and then measuring the percentage of the test set that is unpredictable by the model.

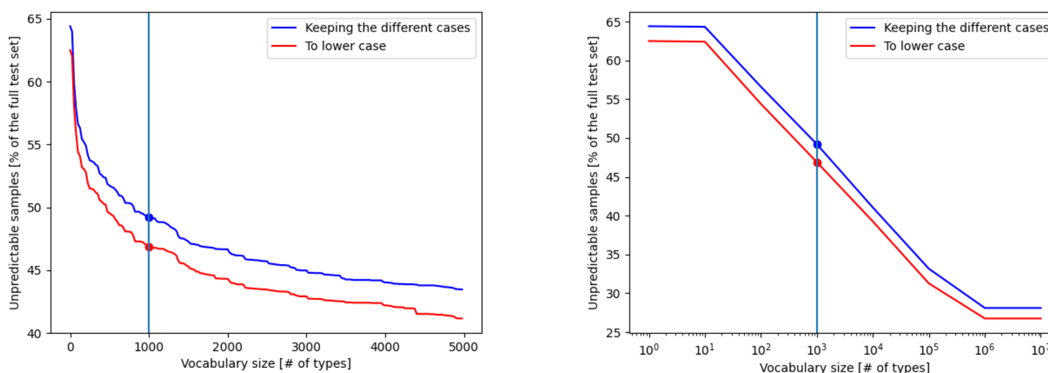


Figure 3.3: Effect of the vocabulary size on the OOV problem.

The result of that experiment is shown on the left of figure 3.3 and on the right in log scale. It shows that an increasing vocabulary size can help lower that percentage of impossible samples from around 50% for a size of 1000 words to around 28% at the cost of a vocabulary of a half-million words. The reason why it does not continue improving after that is that a half-million words vocabulary amounts to putting all the different types of the training set in the vocabulary. The graphs also show that the problem can be slightly reduced by converting all the words to lowercase. However, this discovery can also be used to put things in perspective. The `SequenceR` is able to predict 950 samples out of 2395 doable test samples, which means it predicts perfectly approximately 40% of what it technically can predict. Another way to look at this situation is by building a comprehensive target file for the test set such that the words that are impossible for the model to generate are replaced by <unk>. Evaluating the golden model's predictions with that target file leads to an accuracy of 2048/4711 instead of 950/4711. This is still a pretty interesting result, since it means that the model perfectly predicts the changes that must be made to fix the code while indicating where the solution uses unknown words. The model learns to solve this relaxed problem thanks to the 15840 unpredictable samples present in the 33798 samples of the training set.

### 3.2.3 Difficult samples

This subsection discusses the problem of technically possible but hard samples mentioned in the qualitative analysis. Two aspects defining the difficulty of a sample

are considered: the buggy line length and its distance from its corresponding fix.

**Buggy line length** From the model's point of view, each token in the buggy line is a candidate for deletion or replacement. Moreover, the longer the buggy line, the more possibilities there are for inserting a new token. The task of the model is to produce the exact set of transformations within 50 guesses. Therefore, it can be understood that the length of some of those buggy lines might be problematic. Figure 3.4 shows the distribution of buggy line lengths of the test set. We can see that some lines are very long, with more than 50 tokens. In such cases, it could be argued that a better fault localization is needed for the model to have a reasonable chance of making the right change.

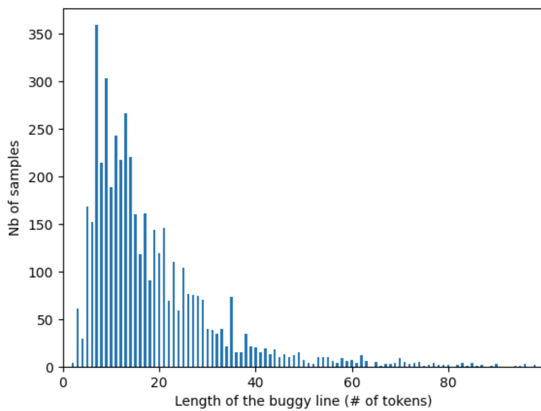


Figure 3.4: Lengths distribution of the test buggy lines.

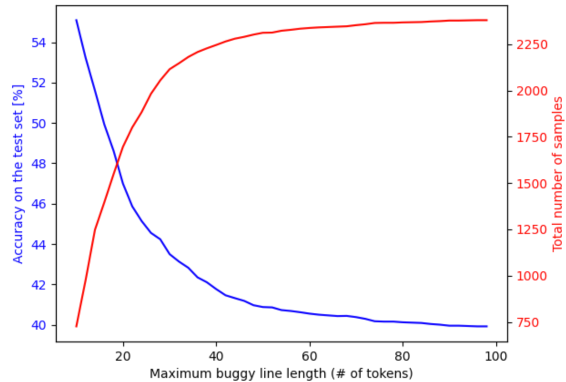


Figure 3.5: Effect of the buggy line lengths on the accuracy of the Golden model on the test set.

Figure 3.5 aims at analysing the actual effect of the buggy line length on the accuracy. In this analysis, samples with a buggy line over a certain length were removed from the test set and the accuracy was re-computed for this subset of the test set. The blue line represents the accuracy and the red one the number of samples left in the test set. The experiment was stopped when the total number of samples reached 750, because it then drops very fast, giving less meaningful results. Note that the total number of samples is never close to 4711 because the impossible samples discussed in the previous subsection were removed from the dataset before this experiment. The results show that the longer buggy lines are harder to train since the accuracy decreases while the maximum buggy line length increases.

**Edit distance** The other metric chosen to characterize these hard problems is the Levenshtein distance, invented by Levenshtein (1966). The Levenshtein distance is

> a string metric for measuring the difference between two sequences.[2]

It corresponds to the number of character deletions, insertions or replacements needed to go from one word to another. In this case, the model is performing the

[2]quoted from `https://en.wikipedia.org/wiki/Levenshtein_distance`

same kind of transformations, not at the character level, but at the token level. In terms of coding, this adaptation can be done very easily in Python by simply implementing a levenshtein distance function and then feeding it lists of words instead of strings.

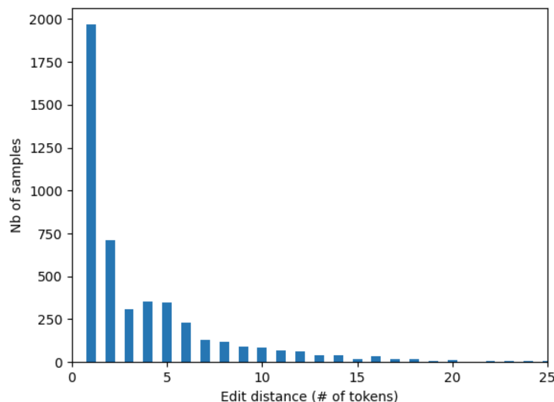


Figure 3.6: Distribution of token-level edit distances between the buggy lines and their fixes.

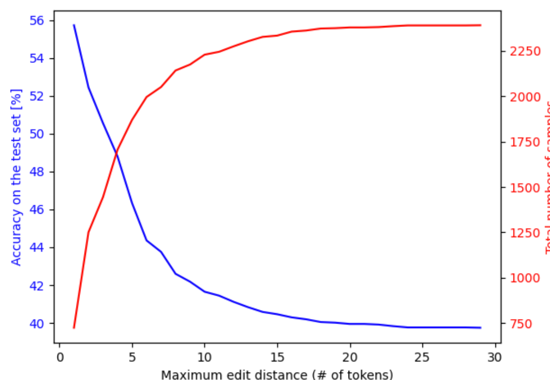


Figure 3.7: Effect of the edit distance on the accuracy of the Golden model on the test set.

Figure 3.6 shows the distribution of such edit distances in the test dataset. With this information, the same experiment as for the buggy line lengths can be conducted for the edit distances. This is shown on figure 3.7. The result is a monotonic curve of the accuracy, very similar to the previous experiment's results. This indicates that the model is having a harder time predicting fixes that are far from their buggy lines.

Finally, figure 3.8 shows the results of an experiment measuring the effect on the accuracy of the combination of the buggy line length and the edit distance. This graph was produced by computing the accuracy on the test set while performing a grid search of the two parameters' previous ranges. Even though the effects of the two metrics are not completely independent, these metrics are not redundant either. These experiments therefore identify two different factors of the difficulty of some of the test samples.

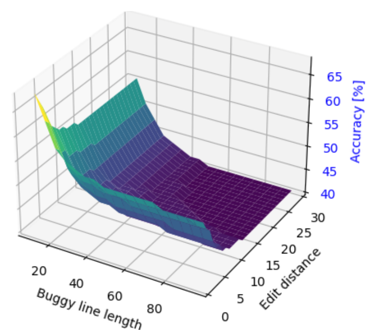


Figure 3.8: Effect of the edit distance and the buggy line length on the accuracy of the Golden model on the test set.

## 3.3 Conclusion

In this chapter, we analysed the limitations of the `SequenceR` hence identifying that the most critical one comes from true OOV problems. Solving this situation would require major changes to the `SequenceR` model. However, the same type of model could still get better performances, which is what is explored in the next chapter.

# Chapter 4

# Improvements

This chapter goes through the different contributions of this master's thesis to the `SequenceR`. The main goal of this work is to see if the model could benefit from additional information about its input data. All the implementation work described in this chapter can be found on Github[1] or on Docker hub[2].

## 4.1 Word features

The main way considered to make the input richer makes use of the word features of the OpenNMT-py framework. This section proposes and defines a list of such word features. These features are all implemented in a Python script.

### 4.1.1 Features definitions

**Line number** The first word feature that was created aims at labeling each token with the number of the line on which it stands. The motivation is that code is organised by statements, each on a different line. However, that information is lost when the abstract buggy context is put on a single line. Since the available dataset was already pre-processed to be a dataset of tokenized abstract buggy context, that feature was implemented by parsing each abstract buggy context and increasing the line count after every '{', '}' and ';' except for the semicolons placed inside a "for" construction or inside a buggy line.

**Number in line** The second word feature is making use of the `Line number` implementation to label each word with its index in the line. This means that, for each line, the first word has index 0, then the second has index 1, etc. This word feature is a little bit similar to the first one, but it could provide different information such as the typical length of a line in a particular code or if any words have a tendency of holding a particular place in the line.

[1] `https://github.com/gildasmulders/SequenceR`
[2] `https://hub.docker.com/repository/docker/gildasmulders/sequencer`

**Encapsulation count** Next, encapsulation count gets a little closer to the tree structure of code by numbering the encapsulation count of each word. For instance the class definition is at level 0, then methods definitions are at level 1, then the buggy method's body is at level 2, etc. This was implemented by counting the opening and closing brackets '{' and '}'.

**Tag** The tag feature distinguishes 7 types of tokens and annotates them with a different letter. Here are the different possibilities:

- `Keywords - k` Words are tagged with the letter 'k' if they belong to the set of reserved words of the Java Language. That list includes 'abstract', 'assert', 'boolean', 'break' and many more.
- `Special symbols - s` The special symbols category refers to what could be seen as punctuation for the Java programming language such as '{', ';', '(', '[' and so on.
- `Operators - o` Operators include the basic '∗', '/', '+', '−', but also '%', all the comparison operators ('==', '<', etc), boolean and bitwise operators.
- `Assignments - a` Assignment operators were originally part of the previous operators tag, but were separated due to their special effect. This tag is also determined by a list including '=', '+ =', '& =', '>>=', etc.
- `Values - v` Values are either strings, defined as any token both starting and ending with " or ′, or numbers. In Java, numbers can end with a L for longs, a F for float or a D for double, so these are first removed if necessary and then the number token is cast into a Python number. If it is not a number, an error is raised and caught.
- `Delimiters - d` The 'd' annotation is only given to two tokens in each abstract buggy context, namely the <START_BUG> and <END_BUG> tokens delimiting the buggy line.
- `Identifiers - i` Finally, all tokens not attributed any of the previous tags are considered to be identifiers.

The aim of this feature is to give a hint about the different roles of these types of tokens to the model. Combined with the `Number in line` feature, it could also potentially allow the model to learn patterns of how different types of tokens are used in a line, therefore easily understanding constructs such as "(Identifier,0) (Assignment,1) (Identifier,2) (Operator,3) (Value,4) (Special symbol,5)".

**Frequency rank** This feature is less related to the natural structure of code. The frequency rank works the following way. First, we count each word's number of occurrences in the abstract buggy context. Then, we sort them by increasing number of occurrences. Finally, we annotate each word so that the most frequent gets the highest number and the less frequent one is assigned 0. The motivation

comes from the way caching works, that is, the more a token was used in the past, the most likely it is to be used again, so we give more importance to frequent tokens. This could be particularly useful for OOV words, since the copy mechanism works from a range of encoded <unk> tokens distinguished by their contexts only.

**Unique id** The unique id feature is motivated by the loss of information about the 20% of the abstract buggy contexts which is made of `"<unk>"` tokens. It aims at fixing that lack of information by giving each token a unique id and ordering them by similarity. This is done by taking the smallest word of the abstract buggy context as token 0 and then assigning the following numbers so that each word has the smallest levenshtein distance from the previous word. This can be seen as a greedy search of the optimal total levenshtein distance. It could be interesting to make an exhaustive search instead of a greedy one, but that could be very time-consuming and a few tests of the greedy search showed that similar words did end up close to each other.

**Distance from buggy line** Finally, the last feature is a variation of the `Line number` where lines are numbered around the buggy line. The buggy line tokens are annotated with 0, the previous lines in the code are annotated with increasing values and the next ones with decreasing, negative, values. The idea is to represent the distance in number of lines between each token and the buggy line. The methods definitions are given a value of 1 so that they are not impacted by the length of the buggy method. The motivation behind this feature can also be seen as a caching mechanism, namely, what was used most recently is likely to be used again.

### 4.1.2 Numerical features in OpenNMT-py

Apart from the tag feature, all the proposed features are numerical, therefore not requiring an embedding to be used by the model. Unfortunately, OpenNMT-py does not support such numerical features, it just takes them as strings and associates each of them to randomly initialised trainable vectors. This is undesirable, because that would just give the model more things to learn rather than helping it with clear information. To make this work, the solution was to modify the locally installed implementation of OpenNMT-py to add this functionality.
From the user's point of view, this additional functionality simply works by feeding one more optional argument with the numerical features indexes when calling the `preprocess.py` and `train.py` scripts. For instance if each word is annotated the following way: token|uniqueid|tag|linenumber, then the numerical features indexes are 0 and 2.
From an implementation point of view, a vocabulary of numbers going from $-1000$ to 1000 is built for numerical features at the pre-processing step. Two special tokens are added to that vocabulary: <unk> in case the model encounters a feature with a value not listed in this vocabulary and <blank> for padding. This range of numbers is enough because the abstract buggy context length is limited to 1000 tokens. Then, when the `train.py` script starts by creating embeddings for words and

features, numerical feature embeddings are created differently. Each 'word' of the vocabulary corresponding to a number $x$ between $-1000$ and 1000 is associated to a vector of size 2: $[1, x]$. Then, the 2 special symbols are represented by $[0, 1]$ for <unk> and $[0, 0]$ is reserved for <blank>. These embeddings are then frozen, so that the model cannot update them during its training.

### 4.1.3 Using the original dataset

The information we can get from a dataset of tokenized truncated abstract buggy contexts is limited. For instance, it is impossible to build the Abstract Syntax Tree of a code from this format. To do so, the original untouched datasets are needed. The idea is then to extract features from the AST built from the full codes. These features could then be used to annotate the corresponding abstract buggy contexts. This is an interesting way to get features because the AST can provide information that is much more accurate than what can be inferred from reading the code as a sequence.

**Recovering the CodRep dataset** The CodRep dataset by Chen and Monperrus (2018) can be found on Github[3]. However, the Bugs2Fix dataset from Tufano et al. (2019) is harder to find. This significantly decreases the number of samples to work with.

The codRep dataset requires some work to build a bijection between the abstract buggy contexts and the codRep tasks. This work can be found on my Github codRep fork[4]. Unfortunately, the whole transformation process from codRep to the SequenceR dataset is not very documented, so not all the samples were recovered. The implemented data recovery process can be described as follows. First, each task file is split in two to have the code in one file and the fixed line in another one. Then, each code goes through the process of abstract buggy context creation, tokenization and truncation described in section 2.2.1. To avoid splitting strings in several tokens, each space located inside a string is replaced by the special "<seq2seq4repair_space>" token before tokenization. Then, these abstract buggy contexts can be

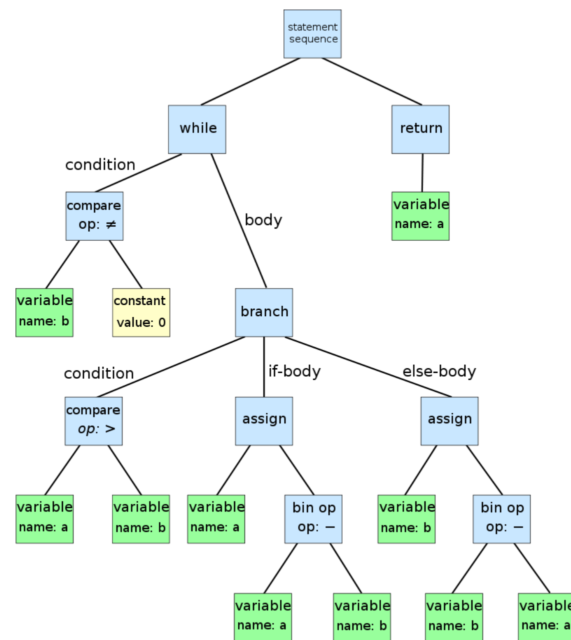


Figure 4.1: Simple AST representation, from `https://en.wikipedia.org/wiki/Abstract_syntax_tree`.

[3] `https://github.com/KTH/CodRep-competition`
[4] `https://github.com/gildasmulders/CodRep-competition`

compared to the SequenceR datasets. The results are that 38.67% of the training set, 37.92% of the validation set and 73.4% of the test set were recovered.

**Building AST features** Using the recovered samples, the first approach considered was to use the Spoon library to build ASTs from the code. This proved to be complicated since ASTs are abstract in the sense that they are defined by a lot of meta-elements which are actually pretty hard to link to the actual tokens written in the code. This complication also indicates that translating the `SequenceR` into a Geometric Deep Learning technique would not be straightforward. A solution was to code a simpler version of the AST inspired by the intuitive understanding of ASTs illustrated by figure 4.1. The actual implementation is simpler than the one depicted by figure 4.1 since it only identifies the classes, methods and control flow structures without diving inside each statement. To do so, it relies on an approach similar to the one adopted for the implementation of the previous features, parsing the code as a string and reacting to semicolons, brackets, etc. The fact that the full code is available is used to extract a more accurate encapsulation count and to recognize methods and their arguments as well as control flow statements and their conditions.

## 4.2 Word embeddings: Word2Vec

This approach is completely different from all the previous word features since it acts on the embeddings of the tokens themselves. It makes use of OpenNMT's possibility to use pretrained embeddings initially instead of starting from random vectors. These embeddings are trained on the whole training set using a Word2Vec model. The motivation behind that is that training a `SequenceR` model multiple times with the exact same parameters can lead to results that vary a lot. The idea is that this variation could be reduced with a non-random embeddings initialization. Like most seq2seq concepts, Word2Vec is initially meant to be used on Natural Language, but it was shown by Budhiraja (2020) that it can give interesting results when applied on source code as well.
Concretely, a Python library called gensim[5] implements both the Skip-gram and CBOW models making it very easy to train embeddings. Skip-grams are known to slightly out-perform the CBOW model, which is why they are used in this master's thesis.

## 4.3 Conclusion

To sum up, we defined 7 word features and modified the OpenNMT implementation to better deal with numerical ones. We also attempted to recover as much untouched data as possible to build a simplified AST from full codes. Finally, we trained Word2Vec embeddings on the source codes of the `SequenceR` training set. The results of these enhancements are discussed in the next chapter.

[5] Řehůřek and Sojka (2010)

# Chapter 5

# Testing

This chapter explains the testing process of the different improvements mentioned in the previous chapter. It then discusses the results of these tests.

## 5.1 Testing process

Once features are added to the data, we cannot simply evaluate the performances of the Golden model by feeding it this annotated data. Indeed, the Golden model is not trained to take advantage of this additional information. Therefore, each different feature or combination of features requires training a new model. With my computer[1], training one model takes about 3 to 4 hours. Then, generating 50 predictions for each of the 4711 samples of the test set with such a model takes around an hour and a half. This process is very long and has to be repeated a lot of times for the different features to be tested, which is why scripts were created to automate this process. Following the example of the authors of the `SequenceR`, these scripts were written in Shell language. Using the most important of these scripts, a user can annotate data, train and test a model from the SequenceR datasets simply by specifying the desired features or whether to use word2vec from the command-line. Calling this script without any parameters therefore amounts to training a model similar to the Golden model. Here is the list of optional arguments accepted by this script and their meaning.

- **–line_index** for the `Line number`.
- **–number** for the `Number in line`.
- **–indent** for the `Encapsulation count` feature.
- **–tag** for the `Tag` word feature.
- **–kmost** for the `Frequency rank`.
- **–uniqueid** for the `Unique id`.

[1]Dell XPS_15 with 16 Gb of RAM and a NVIDIA Geforce 1650 Ti GPU with 4Gb of memory

- **–distbug** for the `Distance from buggy line`.
- **–word2vec** to initialize word embeddings with previously computed word2vec embeddings.
- **–fix_embedding** to freeze the word embeddings.
- **–steps x** specifying the number of steps during which the model should be trained.
- **–checkpoint y** so that a model is saved every y steps.
- **–rm** to remove the model after testing it.

The simple AST feature is not testable with this script as it does not rely on the same datasets.
The final output of this script is written in a file named after the model's set of features and simply consists in the number of samples of the test set that are perfectly predicted by the new model. Eventually, the CECI clusters mentioned in the acknowledgement were used instead of my computer for convenience. However, due to the limited amount of GPU-equipped machines and the important traffic on these clusters, this approach did not improve the run times.

## 5.2 Results

Table 5.1 shows the results of these tests for the different improvements proposed. For each feature, the model was saved and tested after 10000 and 20000 training steps. Compared to the 950 perfect predictions of the Golden model, these results are worse. However, these numbers were obtained after training 1 or 2 models for each different features compared to the ten runs needed before finding the Golden model. These results can therefore be put in perspective by looking at the mean result of the 10 models built with the Golden parameters, which is 859. These results therefore seem very comparable. While they are not included in this table, experiments were also made to combine different features. This produced similar result with a best score of 911 perfect predictions for a model combining the tag and number features and trained for 20000 steps. In conclusion, it seems that neither word2vec embeddings nor word features have a big impact on the models.

| Feature\Steps | 10000 | 20000 |
|---|---|---|
| line_index | 843 | 717 |
| number | 898 | 812 |
| indent | 882 | 781 |
| tag | 866 | 879 |
| kmost | 817 | 712 |
| uniqueid | 846 | 791 |
| distbug | 902 | 864 |
| word2vec | 852 | 775 |

Table 5.1: Number of perfectly predicted test samples (out of 4711) with different features for 10000 and 20000 steps of training.

On the other hand, the simple AST features yield very bad results compared to the Golden model. The Golden model perfectly predicts 640 samples out of the 3458

recovered test samples while the simple AST results range from 300 to 450 perfect predictions. This is probably due to the fact that the simple AST models can only be trained from a subset of less than 40% of the initial training set, which is not enough data.

## 5.3 Conclusion

To conclude, testing each feature is a long process which we automated with Shell scripts. The final results show that the proposed improvements do not have a significant impact on the model. Feature selection is a hard problem, which can in fact be seen as a field of its own (Kumar (2014)). Geometric Deep Learning is a new approach of this field which might be relevant to this machine-learning based application that already requires a big amount of data anyway.

# Chapter 6

# Difficulties Encountered

This master's thesis represents a lot of challenges that can all be seen as opportunities to learn. From installation problems to unknown technologies and to more conceptual problems, this chapter explains the main difficulties encountered during this master thesis work.

## 6.1 Installation

One of the very first challenges is to make the SequenceR work. Indeed, simply cloning the SequenceR github repository and then following the installation instructions does not work. A lot of bugs can arise from this approach for the unexplicit reason that there are multiple mismatches in the versions of the libraries used by this project. A solution is then to get the image of the SequenceR stored on Docker Hub at `https://hub.docker.com/r/zimin/sequencer`. This image does not directly produce a fully working container, but it is easier to fix. One of the key problems of the SequenceR installation is the fact that it relies on an obsolete version of OpenNMT-py. The solution involving a docker image therefore works because the image already contains a local installation of OpenNMT-py at its right version. Nevertheless, the current version of OpenNMT-py has been completely changed, which means that the documentation that can be found on OpenNMT-py online is often irrelevant to understand or modify this old version. At this step, adding code and training models can be a lengthy process since the docker container is accessed through a simple terminal. So it is either coding from nano or vim or going through the following process.

1. Write some code to add a feature to data for instance.
2. Build an image based on the SequenceR working image and copying the local code changes.
3. Run the image to obtain a usable container.
4. Train a model from the container.

One helpful tool when using Visual Studio Code is the docker extension, that allows to use all Visual Studio Code editing facilities 'remotely' to edit code that is located on a docker container. Yet another, better way to work is to copy the contents of the working container on the local machine. Then, a Python virtual environment has to be built to match all the old OpenNMT-py requirements as the docker container does. The list of requirements of this Python environment can be found in the `virtualenv_requirements.txt` file on my github repository[1] for future use. The next challenge is to make GPUs work, since they are needed in order to train models similar to the SequenceR's. Setting up the correct NVIDIA drivers on linux can be a tricky process that can even prevent a computer from booting.

## 6.2 Technologies used

This work relies on a lot of different tools. From a theoretical point of view, it requires a good understanding of a range of machine learning and NLP concepts described in chapter 1. This section lists the practical tools required by this work.

**Platforms** In term of platforms, models were trained using Docker containers, Python virtual environments and finally, CECI clusters. The last platform requires installing dependencies from source and creating SLURM scripts to submit tasks to a job scheduler. Visual studio code extensions are also helpful for working on CECI clusters since they allow to edit code through a ssh connection.

**Languages** The original SequenceR repository makes use of various languages. Some scripts are written in Python, others in Java, in Shell and in Perl.

**Libraries** Finally, OpenNMT-py leverages the Pytorch library, the abstract buggy context creation's AST traversal relies on the Spoon library and Word2vec models are trained using the gensim library. We also tried to use the `INTiMALS` AST models proposed by Nucci et al. (2019), but these required compiled code to build the AST, which was not always possible to get due to the fact that this application works with buggy codes.

## 6.3 Problems of substance

On another level, there are a number of challenges concerning the nature of the work itself. First, we noticed in chapter 3 that the data was not ideal. Deep Learning approaches allow to learn directly from data, alleviating us from the burden of defining application-specific rules manually. However, this task is not necessarily a lot easier, since the challenge then becomes to find good data and to prepare it adequately. Another difficulty relates to the process of research. Even though the features proposed in chapter 4 were not all particularly hard to implement, the

[1] `https://github.com/gildasmulders/SequenceR`

difficulty was to find features that made sense. The lack of convincing results of these features also made this creative process difficult in the sense that there was no indication about which were the most relevant types of features to try.

## 6.4 Conclusion

This master's thesis is a dive into an application which makes use of several concepts and technologies. Making a contribution to this application therefore requires to first acquire this big set of knowledge. After that is done, the next difficulties come from the limitations imposed by the data available and from the need for creativity.

# Conclusion

## Lessons learned

Deep Learning is a field of machine learning that thrives on large amounts of data. It can be used in the framework of Sequence to Sequence (seq2seq) learning to learn to translate sequences from one domain to another, such as from English to French. The `SequenceR` by Chen et al. (2019) relies on that concept to perform Automated Program Repair (APR), which is the field aiming at automatically locating and fixing bugs in source codes. More precisely, the `SequenceR` method focuses on 'translating' a given buggy code into a fixed code by processing the whole buggy code as a sequence of words before generating a fix.
This master's thesis aims at studying the limitations of the `SequenceR` models and at measuring the impact on such models of a richer representation of code. The motivation is that the natural structure of code is better accounted for by Abstract Syntax Trees (AST) than by sequences. The limitations of the `SequenceR` model are investigated through qualitative and quantitative analyses of the datasets and Golden model predictions whereas the impact of a richer representation of code is measured by proposing different features and testing the effect on the models of annotating code with these features.

The experiments conducted as part of this master's thesis highlighted the challenging nature of improving the `SequenceR` method. Indeed, adding features to data to provide a richer representation of code to `SequenceR`-like models did not really seem to impact the performances. Moreover, the main discovery of this master's thesis is that, far more than the code representation, the most limiting factor of this APR technique is still the Out Of Vocabulary (OOV) problem. Nevertheless, this work also suggests that the state of the art methods could benefit from a better understanding of the codes they are altering to better order the kind of fixes they are able to generate. Even though the results of this master's thesis do not confirm the need for a richer representation of data, they cannot refute its potential effect. This master's thesis proved that the tried features were not useful. However, other features and ways to improve the data representation could be tried. A dive into the APR state of the art also revealed the tight connection between APR and AST representations, which are key to heuristic program repair for instance. The interest to use AST representations in Deep learning approaches is also definitely out there, for instance in Tarlow et al. (2020), which uses a string

representation of the AST of compiled code to perform Automated Program Repair on projects builds.

# Further works

Because of this tight link between APR and ASTs, Graph learning still appears as an interesting next step for machined-learning based APR techniques. Meanwhile, the future of APR should focus on the OOV problem and explore different solutions such as detailed below.

- As programmers, when we try to get an understanding of an existing code base, we look at the whole project and its different files. A solution to provide more candidates to the copy mechanism would be to adapt the APR vision to take place at a project-level rather than a one file level.
- A solution to generate Out Of Vocabulary words would be to adopt a character-level sequence to sequence model. This would however be challenging as such models basically have to learn how to spell in addition to everything token-level seq2seq models already have to learn, but it could be a fallback mechanism when the model predicts an OOV word.
- Finally the task of APR could be relaxed as mentioned in this paper to provide a tool that proposes partial patches in the scope of its vocabulary.

Automated Program Repair is a field that calls for innovations. It might look like a field that requires a lot more work before easing programmers' work, but reality is more complex. For instance, Deepcode, by Kim et al. (2018), is a Deep Learning APR technique that can be used to scan Github projects for vulnerabilities and propose fixes. This is therefore not a fantasy; this handy tool exists and just needs more people's interest to become more and more efficient.

# Bibliography


Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

Ayushi, c. (2021). Optimizers in Deep Learning. `https://2809ayushic.medium.com/optimizers-in-deep-learning-31db684c73cf`.

Bahdanau, D., Cho, K., and Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. (2021). Geometric deep learning: Grids, groups, graphs, geodesics, and gauges.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: Going beyond euclidean data.

Brownlee, J. (2017). How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras. `https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/`.

Budhiraja, A. (2020). Word2Vec on source code: Semantic meaning of code and it's beautiful implications. `https://medium.com/@amarbudhiraja`.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. (2019). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transaction on Software Engineering*.

Chen, Z. and Monperrus, M. (2018). The codrep machine learning on source code competition. Technical Report 1807.03200, arXiv.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation.

Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`.

Fey, M. and Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Goues, C. L., Pradel, M., and Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, 62(12):56–65.

Gu, J., Lu, Z., Li, H., and Li, V. O. K. (2016). Incorporating copying mechanism in sequence-to-sequence learning.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

Hosseini, R. and Brusilovsky, P. (2013). Javaparser: A fine-grain concept indexing tool for java problems.

Kim, H., Jiang, Y., Kannan, S., Oh, S., and Viswanath, P. (2018). Deepcode: Feedback codes via deep learning. *CoRR*, abs/1807.00801.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. (2017). OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Kumar, V. (2014). Feature Selection: A literature Review.

Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707.

Loye, G. (2019). Attention Mechanism. `https://blog.floydhub.com/attention-mechanism/`.

Luong, M.-T., Pham, H., and Manning, C. (2015). Effective approaches to attention-based neural machine translation.

Mikolov, T., Corrado, G., Chen, K., and Dean, J. (2013). Efficient estimation of word representations in vector space.

Monperrus, M. (2018). The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr.

Nucci, D. D., Pham, H. S., Fabry, J., Mens, K., Molderez, T., Nijssen, S., Roover, C. D., and Zaytsev, V. (2019). Language-Parametric Modular Framework for Mining Idiomatic Code Patterns. In *Proceedings of the 12th Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE)*, page 6, Bolzano, Italy.

Olah, C. (2015). Understanding lstm networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179.

Pitie, F. (2020). Deep learning and its applications. `https://frcs.github.io/4C16-LectureNotes/`. Material of the E4C16 module at Trinity College, Dublin.

Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks.

Tarlow, D., Moitra, S., Rice, A., Chen, Z., Manzagol, P.-A., Sutton, C., and Aftandilian, E. (2020). Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 19–20, New York, NY, USA. Association for Computing Machinery.

Taylor, M. (2019). Geometric deep learning — Convolutional Neural Networks on Graphs and Manifolds.

Tong, F. (2021). What is Geometric Deep Learning? `https://flawnsontong.medium.com/what-is-geometric-deep-learning-b2adb662d91d`.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *arXiv:1812.08693 [cs]*. arXiv: 1812.08693.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

Walsh, E. P. (2020). Incorporating a copy mechanism into sequence-to-sequence models. `https://medium.com/@epwalsh10/incorporating-a-copy-mechanism-into-sequence-to-sequence-models-40917280b89d`.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2021). Dive into Deep Learning.

# Appendix

Here are the full examples of chapter 3. For each of these samples, you can find the full abstract buggy context in black, then the buggy line in red, and finally, the expected fix for that line in green.

Listing A6.1: Method change failure.

```
@Override public void glPixelStorei ( int pname , int param ) { } @Override public void glPolygonOffset ( float factor , float units )
{ } @Override public void glReadPixels ( int x , int y , int width , int height , int format , int type , Buffer pixels ) { } @Override
public void glReleaseShaderCompiler ( ) { } @Override public void glRenderbufferStorage ( int target , int internalformat , int width
, int height ) { } @Override public void glSampleCoverage ( float value , boolean invert ) { } @Override public void glScissor ( int x ,
int y , int width , int height ) { } @Override public void glShaderBinary ( int n , IntBuffer shaders , int binaryformat , Buffer binary
, int length ) { } @Override public void glShaderSource ( int shader , String string ) { } @Override public void glStencilFunc ( int
func , int ref , int mask ) { } @Override public void glStencilFuncSeparate ( int face , int func , int ref , int mask ) { } @Override
public void glStencilMask ( int mask ) { } @Override public void glStencilMaskSeparate ( int face , int mask ) { } @Override public
void glStencilOp ( int fail , int zfail , int zpass ) { } @Override public void glStencilOpSeparate ( int face , int fail , int zfail , int
zpass ) { } @Override public void glTexImage2D ( int target , int level , int internalformat , int width , int height , int border , int
format , int type , Buffer pixels ) { } @Override public void glTexParameterf ( int target , int pname , float param ) { } @Override
public void glTexParameterfv ( int target , int pname , FloatBuffer params ) { } @Override public void glTexParameteri ( int target
, int pname , int param ) { } @Override public void glTexParameteriv ( int target , int pname , IntBuffer params ) { } @Override
public void glTexSubImage2D ( int target , int level , int xoffset , int yoffset , int width , int height , int format , int type , Buffer
pixels ) { } @Override public void glUniform1f ( int location , float x ) { } @Override public void glUniform1fv ( int location , int
count , FloatBuffer v ) { } @Override public void glUniform1i ( int location , int x ) { } @Override public void glUniform1iv ( int
location , int count , IntBuffer v ) { } @Override public void glUniform2f ( int location , float x , float y ) { } @Override public
void glUniform2fv ( int location , int count , FloatBuffer v ) { } @Override public void glUniform2i ( int location , int x , int y ) {
} @Override public void glUniform2iv ( int location , int count , IntBuffer v ) { } @Override public void glUniform3f ( int location
, float x , float y , float z ) { } @Override public void glUniform3fv ( int location , int count , FloatBuffer v ) { } @Override public
void glUniform3i ( int location , int x , int y , int z ) { } @Override public void glUniform3iv ( int location , int count , IntBuffer v
) { } @Override public void glUniform4f ( int location , float x , float y , float z , float w ) { } @Override public void glUniform4fv
( int location , int count , FloatBuffer v ) { } @Override public void glUniform4i ( int location , int x , int y , int z , int w ) {
} @Override public void glUniform4iv ( int location , int count , IntBuffer v ) { } @Override public void glUniformMatrix2fv ( int
location , int count , boolean transpose , FloatBuffer value ) { } @Override public void glUniformMatrix3fv ( int location , int
count , boolean transpose , FloatBuffer value ) { } @Override public void glUniformMatrix4fv ( int location , int count , boolean
transpose , FloatBuffer value ) { <START_BUG> gl . glUniformMatrix2x4fv ( location , count , transpose , value ) ; <END_BUG>
} @Override public void glUseProgram ( int program ) { } @Override public void glValidateProgram ( int program ) { } @Override
public void glVertexAttrib1f ( int indx , float x ) { } @Override public void glVertexAttrib1fv ( int indx , FloatBuffer values ) {
} @Override public void glVertexAttrib2f ( int indx , float x , float y ) { } @Override public void glVertexAttrib2fv ( int indx ,
FloatBuffer values ) { } @Override public void glVertexAttrib3f ( int indx , float x , float y , float z ) { } @Override public void
glVertexAttrib3fv ( int indx , FloatBuffer values ) { } @Override public void glVertexAttrib4f ( int indx , float x , float y , float z ,
float w ) { } @Override public void glVertexAttrib4fv ( int indx , FloatBuffer values ) { } @Override public void glVertexAttribPointer
( int indx , int size , int type , boolean normalized , int stride , Buffer ptr ) { } @Override public void glViewport ( int x , int y , int
width , int height ) { } @Override public void glDrawElements ( int mode , int count , int type , int indices ) { } @Override public
void glVertexAttribPointer ( int indx , int size , int type , boolean normalized , int stride , int ptr ) { } }
gl . glUniformMatrix2x4fv ( location , count , transpose , value ) ;
gl . glUniformMatrix4fv ( location , count , transpose , value ) ;
```

Listing A6.2: Operator change failures.

```
public class Bits { long [ ] bits = new long [ ] { 0 } ; public boolean get ( int index ) { } public void set ( int index ) { } public void flip ( int index ) { } private void checkCapacity ( int len ) { <START_BUG> if ( len > ( bits . length ) ) { <END_BUG> long [ ] newBits = new long [ len + 1 ] ; System . arraycopy ( bits , 0 , newBits , 0 , bits . length ) ; bits = newBits ; } } public void clear ( int index ) { } public void clear ( ) { } public int numBits ( ) { } }
if ( len > ( bits . length ) ) {
if ( len >= ( bits . length ) ) {
public class FloatFieldMapper extends NumberFieldMapper < Float > { public static final String CONTENT_TYPE = "float" ; public static class Defaults extends NumberFieldMapper . Defaults { public static final FieldType FLOAT_FIELD_TYPE = new FieldType ( NumberFieldMapper . Defaults . NUMBER_FIELD_TYPE ) ; public static final Float NULL_VALUE = null ; } public static class Builder extends NumberFieldMapper . Builder < FloatFieldMapper . Builder , FloatFieldMapper > { protected Float nullValue = FloatFieldMapper . Defaults . NULL_VALUE ; public Builder ( String name ) { } public FloatFieldMapper . Builder nullValue ( float nullValue ) { } @Override public FloatFieldMapper build ( BuilderContext context ) { <START_BUG> fieldType . setOmitNorms ( ( ( fieldType . omitNorms ( ) ) || ( ( boost ) != 1.0F ) ) ) ; <END_BUG> FloatFieldMapper fieldMapper = new FloatFieldMapper ( buildNames ( context ) , precisionStep , fuzzyFactor , boost , fieldType , nullValue , ignoreMalformed ( context ) ) ; fieldMapper . includeInAll ( includeInAll ) ; return fieldMapper ; } } public static class TypeParser implements Mapper . TypeParser { @Override public Mapper . Builder parse ( String name , Map < String , Object > node , ParserContext parserContext ) throws MapperParsingException { } } private Float nullValue ; private String nullValueAsString ; protected FloatFieldMapper ( Names names , int precisionStep , String fuzzyFactor , float boost , FieldType fieldType , Float nullValue , Explicit < Boolean > ignoreMalformed ) { } @Override protected int maxPrecisionStep ( ) { } @Override public Float value ( Field field ) { } @Override public Float valueFromString ( String value ) { } @Override public String indexedValue ( String value ) { } @Override public Query fuzzyQuery ( String value , String minSim , int prefixLength , int maxExpansions , boolean transpositions ) { } @Override public Query fuzzyQuery ( String value , double minSim , int prefixLength , int maxExpansions , boolean transpositions ) { } @Override public Query fieldQuery ( String value , @Nullable QueryParseContext context ) { } @Override public Query rangeQuery ( String lowerTerm , String upperTerm , boolean includeLower , boolean includeUpper , @Nullable QueryParseContext context ) { } @Override public Filter fieldFilter ( String value , @Nullable QueryParseContext context ) { } @Override public Filter rangeFilter ( String lowerTerm , String upperTerm , boolean includeLower , boolean includeUpper , @Nullable QueryParseContext context ) { } @Override public Filter rangeFilter ( FieldDataCache fieldDataCache , String lowerTerm , String upperTerm , boolean includeLower , boolean includeUpper , @Nullable QueryParseContext context ) { } @Override public Filter nullValueFilter ( ) { } @Override protected boolean customBoost ( ) { } @Override protected Field innerParseCreateField ( ParseContext context ) throws IOException { } @Override public FieldDataType fieldDataType ( ) { } @Override protected String contentType ( ) { } @Override public void merge ( Mapper mergeWith , MergeContext mergeContext ) throws MergeMappingException { } @Override protected void doXContentBody ( XContentBuilder builder ) throws IOException { } public static class CustomFloatNumericField extends CustomNumericField { private final float number ; private final NumberFieldMapper mapper ; public CustomFloatNumericField ( NumberFieldMapper mapper , float number , FieldType fieldType ) { } @Override public TokenStream tokenStream ( Analyzer analyzer ) throws IOException { } @Override public String numericAsString ( ) { } } }
fieldType . setOmitNorms ( ( ( fieldType . omitNorms ( ) ) || ( ( boost ) != 1.0F ) ) ) ;
fieldType . setOmitNorms ( ( ( fieldType . omitNorms ( ) ) && ( ( boost ) == 1.0F ) ) ) ;
```

Listing A6.3: Target using an unkwown string value.

```
public class TextureDataTest extends GdxTest { private SpriteBatch spriteBatch ; private Sprite sprite ; public void create ( ) { spriteBatch = new SpriteBatch ( ) ; sprite = new Sprite ( new Texture ( new TextureData ( ) { int width = 0 ; int height = 0 ; public void load ( ) { <START_BUG> Pixmap pixmap = new Pixmap ( files . internal ( "data/badlogicsmall.jpg" ) ) ; <END_BUG> gl . glTexImage2D ( GL_TEXTURE_2D , 0 , pixmap . getGLInternalFormat ( ) , pixmap . getWidth ( ) , pixmap . getHeight ( ) , 0 , pixmap . getGLFormat ( ) , pixmap . getGLType ( ) , pixmap . getPixels ( ) ) ; width = pixmap . getWidth ( ) ; height = pixmap . getHeight ( ) ; pixmap . dispose ( ) ; } public int getWidth ( ) { return width ; } public int getHeight ( ) { return height ; } } ) ) ; } public void render ( ) { } public boolean needsGL20 ( ) { } }
Pixmap pixmap = new Pixmap ( files . internal ( "data/badlogicsmall.jpg" ) ) ;
Pixmap pixmap = new Pixmap ( files . internal ( "data/stone2.png" ) ) ;
```

Listing A6.4: Target using an unknown object.

```
public class LwjglDebugStarter { public static void main ( String [ ] argv ) { new SharedLibraryLoader ( "../../extensions/gdx-audio/libs/gdx-audio-natives.jar" ) . load ( "gdx-audio" ) ; new SharedLibraryLoader ( "../../extensions/gdx-image/libs/gdx-image-natives.jar" ) . load ( "gdx-image" ) ; new SharedLibraryLoader ( "../../extensions/gdx-freetype/libs/gdx-freetype-natives.jar" ) . load ( "gdx-freetype" ) ; new SharedLibraryLoader ( "../../extensions/gdx-controllers/gdx-controllers-desktop/libs/gdx-controllers-desktop-natives.jar" ) . load ( "gdx-controllers-desktop" ) ; new SharedLibraryLoader ( "../../gdx/libs/gdx-natives.jar" ) . load ( "gdx" ) ; <START_BUG> GdxTest test = new BatchRenderTest ( ) ; <END_BUG> LwjglApplicationConfiguration config = new LwjglApplicationConfiguration ( ) ; config . useGL20 = test . needsGL20 ( ) ; new com . badlogic . gdx . backends . lwjgl . LwjglApplication ( test , config ) ; } }
GdxTest test = new BatchRenderTest ( ) ;
GdxTest test = new JsonModelLoaderTest ( ) ;
```

Listing A6.5: Target using unknown methods or attributes.

```
public class TransportBenchmark { static enum Type { LOCAL ( ) { @Override public Transport newTransport ( Settings settings ,
ThreadPool threadPool ) { } } , NETTY ( ) { @Override public Transport newTransport ( Settings settings , ThreadPool threadPool
) { } } ; public abstract Transport newTransport ( Settings settings , ThreadPool threadPool ) { } } public static void main ( String
[ ] args ) { <START_BUG> final String executor = Names . CACHED ; <END_BUG> final boolean waitForRequest = true ;
final ByteSizeValue payloadSize = new ByteSizeValue ( 100 , ByteSizeUnit . BYTES ) ; final int NUMBER_OF_CLIENTS = 1 ;
final int NUMBER_OF_ITERATIONS = 100000 ; final byte [ ] payload = new byte [ ( ( int ) ( payloadSize . bytes ( ) ) ) ] ; final
AtomicLong idGenerator = new AtomicLong ( ) ; final TransportBenchmark . Type type = TransportBenchmark . Type . NETTY ;
Settings settings = ImmutableSettings . settingsBuilder ( ) . build ( ) ; final ThreadPool serverThreadPool = new ThreadPool ( ) ;
final TransportService serverTransportService = new TransportService ( type . newTransport ( settings , serverThreadPool ) , server-
ThreadPool ) . start ( ) ; final ThreadPool clientThreadPool = new ThreadPool ( ) ; final TransportService clientTransportService
= new TransportService ( type . newTransport ( settings , clientThreadPool ) , clientThreadPool ) . start ( ) ; final DiscoveryNode
node = new DiscoveryNode ( "server" , serverTransportService . boundAddress ( ) . publishAddress ( ) ) ; serverTransportService .
registerHandler ( "benchmark" , new BaseTransportRequestHandler < BenchmarkMessage > ( ) { @Override public BenchmarkMes-
sage newInstance ( ) { return new BenchmarkMessage ( ) ; } @Override public String executor ( ) { return executor ; } @Override
public void messageReceived ( BenchmarkMessage request , TransportChannel channel ) throws Exception { channel . sendResponse
( request ) ; } } ) ; clientTransportService . connectToNode ( node ) ; for ( int i = 0 ; i < 10000 ; i ++ ) { BenchmarkMessage
message = new BenchmarkMessage ( 1 , payload ) ; clientTransportService . submitRequest ( node , "benchmark" , message , new
BaseTransportResponseHandler < BenchmarkMessage > ( ) { @Override public BenchmarkMessage newInstance ( ) { return new
BenchmarkMessage ( ) ; } @Override public String executor ( ) { return Names . SAME ; } @Override public void handleResponse (
BenchmarkMessage response ) { } @Override public void handleException ( TransportException exp ) { exp . printStackTrace ( ) ; } }
) . txGet ( ) ; } Thread [ ] clients = new Thread [ NUMBER_OF_CLIENTS ] ; final CountDownLatch latch = new CountDownLatch
( ( NUMBER_OF_CLIENTS * NUMBER_OF_ITERATIONS ) ) ; for ( int i = 0 ; i < NUMBER_OF_CLIENTS ; i ++ ) { clients
[ i ] = new Thread ( new Runnable ( ) { @Override public void run ( ) { for ( int j = 0 ; j < NUMBER_OF_ITERATIONS ; j ++
) { final long id = idGenerator . incrementAndGet ( ) ; BenchmarkMessage message = new BenchmarkMessage ( id , payload ) ;
BaseTransportResponseHandler < BenchmarkMessage > handler = new BaseTransportResponseHandler < BenchmarkMessage > ( )
{ @Override public BenchmarkMessage newInstance ( ) { return new BenchmarkMessage ( ) ; } @Override public String executor ( )
{ return executor ; } @Override public void handleResponse ( BenchmarkMessage response ) { if ( ( response . id ) != id ) { System
. out . println ( ( ( ( ( "NO<seq2seq4repair_space>ID<seq2seq4repair_space>MATCH<seq2seq4repair_space>[" + ( response .
id ) ) + "]<seq2seq4repair_space>and<seq2seq4repair_space>[" ) + id ) + "]" ) ) ; } latch . countDown ( ) ; } @Override public
void handleException ( TransportException exp ) { exp . printStackTrace ( ) ; latch . countDown ( ) ; } } ; if ( waitForRequest )
{ clientTransportService . submitRequest ( node , "benchmark" , message , handler ) . txGet ( ) ; } else { clientTransportService .
sendRequest ( node , "benchmark" , message , handler ) ; } } } } ) ; } StopWatch stopWatch = new StopWatch ( ) . start ( ) ; for (
int i = 0 ; i < NUMBER_OF_CLIENTS ; i ++ ) { clients [ i ] . start ( ) ; } try { latch . await ( ) ; } catch ( InterruptedException
e ) { e . printStackTrace ( ) ; } stopWatch . stop ( ) ; System . out . println ( ( ( ( ( ( ( ( ( ( "Ran<seq2seq4repair_space>["
+ NUMBER_OF_CLIENTS ) + "],<seq2seq4repair_space>each<seq2seq4repair_space>with<seq2seq4repair_space>[" ) + NUM-
BER_OF_ITERATIONS ) + "]<seq2seq4repair_space>iterations,<seq2seq4repair_space>payload<seq2seq4repair_space>[" ) +
payloadSize ) + "]:<seq2seq4repair_space>took<seq2seq4repair_space>[" ) + ( stopWatch . totalTime ( ) ) ) +
"],<seq2seq4repair_space>TPS:<seq2seq4repair_space>" ) + ( ( NUMBER_OF_CLIENTS * NUMBER_OF_ITERATIONS ) /
( stopWatch . totalTime ( ) . secondsFrac ( ) ) ) ) ) ; clientTransportService . close ( ) ; clientThreadPool . shutdownNow ( ) ;
serverTransportService . close ( ) ; serverThreadPool . shutdownNow ( ) ; } }
```

```
final String executor = Names . CACHED ;
final String executor = Names . GENERIC ;
```

```
public class AnalyzeActionTests extends AbstractNodesTests { private Client client ; @BeforeClass public void createNodes ( ) throws
Exception { } @AfterClass public void closeNodes ( ) { } protected Client getClient ( ) { } @Test public void simpleAnalyzerTests (
) throws Exception { } @Test public void analyzerWithFieldOrTypeTests ( ) throws Exception { try { client . admin ( ) . indices (
) . prepareDelete ( "test" ) . execute ( ) . actionGet ( ) ; } catch ( Exception e ) { } client . admin ( ) . indices ( ) . prepareCreate
( "test" ) . execute ( ) . actionGet ( ) ; client . admin ( ) . cluster ( ) . prepareHealth ( ) . setWaitForGreenStatus ( ) . execute ( ) .
actionGet ( ) ; client . admin ( ) . indices ( ) . preparePutMapping ( "test" ) . setType ( "document" ) . setSource ( ( "\n" + ( ( ( (
( ( ( ( "<seq2seq4repair_space>d̈ocument:̈\n" + "<seq2seq4repair_space>p̈roperties:̈\n" ) + "<seq2seq4repair_space>s̈imple:̈\n"
) + "<seq2seq4repair_space>ẗype:̈s̈tring,̈\n" ) + "<seq2seq4repair_space>änalyzer:̈<seq2seq4repair_space>s̈implë\n" ) +
"<seq2seq4repair_space>\n" ) + "<seq2seq4repair_space>\n" ) + "<seq2seq4repair_space>\n" ) + "" ) ) ) . execute ( ) . actionGet
( ) ; for ( int i = 0 ; i < 10 ; i ++ ) { final AnalyzeRequestBuilder requestBuilder = client . admin ( ) . indices ( ) . prepareAnalyze
( "test" , "THIS<seq2seq4repair_space>IS<seq2seq4repair_space>A<seq2seq4repair_space>TEST" ) ; <START_BUG> request-
Builder . field ( "document.simple" ) ; <END_BUG> AnalyzeResponse analyzeResponse = requestBuilder . execute ( ) . actionGet
( ) ; assertThat ( analyzeResponse . tokens ( ) . size ( ) , equalTo ( 4 ) ) ; AnalyzeResponse . AnalyzeToken token = analyzeResponse
. tokens ( ) . get ( 3 ) ; assertThat ( token . term ( ) , equalTo ( "test" ) ) ; assertThat ( token . startOffset ( ) , equalTo ( 10 ) ) ;
assertThat ( token . endOffset ( ) , equalTo ( 14 ) ) ; } } }
```

```
requestBuilder . field ( "document.simple" ) ;
requestBuilder . setField ( "document.simple" ) ;
```

Listing A6.6: Long buggy line.

```
@Test public class SimpleJsonIndexQueryParserTests { private final Index index = new Index ( "test" ) ; @Test public void test-
QueryStringBuilder ( ) throws Exception { } @Test public void testQueryString ( ) throws Exception { } @Test public void test-
QueryStringFields1Builder ( ) throws Exception { } @Test public void testQueryStringFields1 ( ) throws Exception { } @Test public
void testQueryStringFields2Builder ( ) throws Exception { } @Test public void testQueryStringFields2 ( ) throws Exception { } @Test
public void testQueryStringFields3Builder ( ) throws Exception { } @Test public void testQueryStringFields3 ( ) throws Exception
{ } @Test public void testMatchAllBuilder ( ) throws Exception { } @Test public void testMatchAll ( ) throws Exception { } @Test
public void testDisMaxBuilder ( ) throws Exception { } @Test public void testDisMax ( ) throws Exception { } @Test public void
testTermQueryBuilder ( ) throws IOException { } @Test public void testTermQuery ( ) throws IOException { } @Test public void
testFieldQueryBuilder1 ( ) throws IOException { } @Test public void testFieldQuery1 ( ) throws IOException { } @Test public
void testFieldQuery2 ( ) throws IOException { } @Test public void testFieldQuery3 ( ) throws IOException { } @Test public void
testTermWithBoostQueryBuilder ( ) throws IOException { } @Test public void testTermWithBoostQuery ( ) throws IOException {
} @Test public void testPrefixQueryBuilder ( ) throws IOException { } @Test public void testPrefixQuery ( ) throws IOException {
} @Test public void testPrefixFilteredQueryBuilder ( ) throws IOException { } @Test public void testPrefixFilteredQuery ( ) throws
IOException { } @Test public void testPrefixQueryBoostQueryBuilder ( ) throws IOException { } @Test public void testPrefixQuery-
BoostQuery ( ) throws IOException { } @Test public void testWildcardQueryBuilder ( ) throws IOException { } @Test public void
testWildcardQuery ( ) throws IOException { } @Test public void testRangeQueryBuilder ( ) throws IOException { } @Test public
void testRangeQuery ( ) throws IOException { } @Test public void testRangeFilteredQueryBuilder ( ) throws IOException { } @Test
public void testRangeFilteredQuery ( ) throws IOException { } @Test public void testBoolFilteredQuery ( ) throws IOException {
} @Test public void testBoolQueryBuilder ( ) throws IOException { } @Test public void testBoolQuery ( ) throws IOException { }
@Test public void testFilteredQueryBuilder ( ) throws IOException { } @Test public void testFilteredQuery ( ) throws IOException
{ } @Test public void testFilteredQuery2 ( ) throws IOException { } @Test public void testFilteredQuery3 ( ) throws IOException
{ } @Test public void testFilteredQuery4 ( ) throws IOException { } @Test public void testTermsFilterQueryBuilder ( ) throws
Exception { } @Test public void testTermsFilterQuery ( ) throws Exception { } @Test public void testConstantScoreQueryBuilder
( ) throws IOException { } @Test public void testConstantScoreQuery ( ) throws IOException { } @Test public void testSpanTer-
mQueryBuilder ( ) throws IOException { } @Test public void testSpanTermQuery ( ) throws IOException { } @Test public void
testSpanNotQueryBuilder ( ) throws IOException { } @Test public void testSpanNotQuery ( ) throws IOException { } @Test public
void testSpanFirstQueryBuilder ( ) throws IOException { } @Test public void testSpanFirstQuery ( ) throws IOException { } @Test
public void testSpanNearQueryBuilder ( ) throws IOException { } @Test public void testSpanNearQuery ( ) throws IOException { }
@Test public void testSpanOrQueryBuilder ( ) throws IOException { } @Test public void testSpanOrQuery ( ) throws IOException {
} @Test public void testQueryFilterBuilder ( ) throws Exception { } @Test public void testQueryFilter ( ) throws Exception { } @Test
public void testMoreLikeThisBuilder ( ) throws Exception { } @Test public void testMoreLikeThis ( ) throws Exception { } @Test
public void testMoreLikeThisFieldBuilder ( ) throws Exception { } @Test public void testMoreLikeThisField ( ) throws Exception
{ } private JsonIndexQueryParser newQueryParser ( ) throws IOException { <START_BUG> return new JsonIndexQueryParser
( new Index ( "test" ) , EMPTY_SETTINGS , newMapperService ( ) , new org . elasticsearch . index . cache . filter . none .
NoneFilterCache ( index , EMPTY_SETTINGS ) , new org . elasticsearch . index . analysis . AnalysisService ( index ) , null , null
, "test" , null ) ; <END_BUG> } private MapperService newMapperService ( ) throws IOException { } }
return new JsonIndexQueryParser ( new Index ( "test" ) , EMPTY_SETTINGS , newMapperService
( ) , new org . elasticsearch . index . cache . filter . none . NoneFilterCache ( index ,
EMPTY_SETTINGS ) , new org . elasticsearch . index . analysis . AnalysisService ( index ) , null
, null , "test" , null ) ;
return new JsonIndexQueryParser ( new Index ( "test" ) , EMPTY_SETTINGS , newMapperService
( ) , new org . elasticsearch . index . cache . filter . none . NoneFilterCache ( index ,
EMPTY_SETTINGS ) , new org . elasticsearch . index . analysis . AnalysisService ( index ) , null
, null , null , "test" , null ) ;
```

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**