University of Applied Sciences RheinMain
Faculty of Design, Computer Science and Media
Computer Science

Master Thesis

to obtain the academic degree of

Master of Science (M. Sc.)

A syntax-aware Transformer Model for Unit Test Completion

Submitted by	Jonas Depoix
Submission date	January 22, 2021
Supervisor	Prof. Dr. Adrian Ulges
Co-supervisor	Prof. Dr. Dirk Krechel

Erklärung gemäß ABPO

Ich erkläre hiermit, dass ich

- die vorliegende Abschlussarbeit selbstständig angefertigt,
- keine anderen als die angegebenen Quellen benutzt,
- die wörtlich oder dem Inhalt nach aus fremden Arbeiten entnommenen Stellen, bildlichen Darstellungen und dergleichen als solche genau kenntlich gemacht und
- keine unerlaubte fremde Hilfe in Anspruch genommen habe.

Wiesbaden, 22. Januar 2021 _____

Jonas Depoix

Erklärung zur Verwendung der Master Thesis

Hiermit erkläre ich mein Einverständnis mit den im folgenden aufgeführten Verbreitungsformen dieser Abschlussarbeit:

Verbreitungsform	**Ja**	**Nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger	×	
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	×	
Veröffentlichung des Titels der Arbeit im Internet	×	
Veröffentlichung der Arbeit im Internet	×	

Wiesbaden, 22. Januar 2021 _____

Jonas Depoix

Abstract

Although writing software tests is an important part of software development, it is often neglected as it is time consuming and considered rather tedious by most developers. This has motivated various endeavors towards automatically generating software tests. Most notably, approaches like *EvoSuite* and *Randoop* showed their ability to generate tests with decent code coverage using static code analysis and heuristics. However, recent studies have found the readability of those generated tests to be subpar. They also discovered that these approaches frequently fail to properly validate if the executed code behaved as expected, revealing that they struggle to generate meaningful assertion statements.

The recent introduction of an attention focused neural network architecture called Transformer has shown groundbreaking success in various Natural Language Processing tasks. This thesis presents a novel approach employing such a Transformer model to generate source code for software tests. As previous research towards machine learning driven test generation is limited and mostly yielded poor results, this thesis tackles the more approachable task of test completion, where only a part of a test is generated given the other part. More specifically, the part of the test is generated which asserts that the method under test fulfills its requirements.

With 2.2 million datapoints for test generation and 1.3 million for test completion this thesis contributes the biggest dataset for those tasks available to date. The model trained on this dataset innovates on the previous test generation approaches by encoding the processed source code using Abstract Syntax Trees (AST). In experiments the model has been shown to leverage the syntactic structure embedded in ASTs, thereby improving its performance by 1.732 BLEU points. Another novelty introduced by the proposed approach is the augmentation of the model's input with the declarations of methods used in the context of the test or method under test. These are identified through AST analysis and were shown to significantly improve the model's performance by 10.375 BLEU points.

Using this approach, the model achieves a BLEU score of 38.192, while only generating unparsable code in 1.92% of cases. This indicates a good understanding of source code's syntactic structure and a promising ability to complete tests with meaningful predictions. However, experiments have indicated that the model struggles to generalize knowledge it has gathered from tests within the same project to tests in other projects, which remains as a key challenge. Nonetheless, these results show this approach to have great potential future work could build on to take a step towards succeeding at machine learning driven test generation.

Contents

Chapter 1

Introduction

With the growing size and complexity of modern software, writing software tests is becoming an increasingly important part of its development process. While they allow developers to validate that their code is working as intended during development, their most valuable property is their ability to verify that the code is still able to fulfill the same requirements after new features were added or existing code has been refactored. A common problem when working with untested codebases is that developers shy away from touching older parts of the codebase, as they have no way of testing if their changes will break other parts of the application. This leads to them adding additional code on top of a potentially already messy codebase, instead of changing parts in desperate need of a refactoring, thereby worsening the code quality increasingly [1]. This makes software tests a safety net allowing developers to refactor and extend code with greater confidence [2], which enables faster and more reliable development iterations.

However, writing tests is a time consuming and therefore expensive task, which is considered to be rather tedious by most developers. Also, as the before mentioned properties mostly benefit future development, the value of writing tests is often not as present in the moment of writing the code which should be tested. Therefore, this task is often neglected due to pressing deadlines, or quite frankly laziness, even though 88% of software developers surveyed in [3] stated that they think they would be better off writing tests, whenever they skip doing so.

Naturally, this has motivated various endeavors to decrease the cost of writing software tests through tooling and automation. Most notably, academia brought forward several approaches using static code analysis and heuristics to generate entire test suites for a given class [4, 5, 6, 7, 8]. While these approaches are able to achieve decent code coverage, recent studies [9, 10] have revealed considerable limitations holding them back from being relevant for practical use. Besides being

computationally expensive, the readability of the generated tests leaves much to be desired. More severely however, [9] and [10] revealed that although the generated tests covered relevant execution branches, many of the generated assertions failed to properly test if the executed code behaved as expected. This does not come as too much of a surprise considering that the employed heuristics analyze individual instructions without developing an understanding of the requirements the code under test fulfills.

The recent introduction of a novel machine learning architecture called Transformer by Vaswani et al. [11] set a new standard in machine translation at a lower training cost than previous approaches [12, 13, 14, 15]. It also has set the stage for a plethora of new models based on the Transformer architecture, outperforming previous state-of-the-art models in various Natural Language Processing (NLP) tasks [16, 17, 18, 19, 20, 21]. These Transformer based models have shown an impressive ability to understand natural language in tasks such as text summarization and generation [16, 18, 19, 20], which raises the question if a similar approach could be utilized to generate software tests. While employing machine learning architectures originating from NLP on software-engineering tasks is not a novel concept [22, 23, 24, 25, 26, 27], there has been very limited research on doing so to generate software tests [28, 29, 30] with mostly poor results [28, 29].

Therefore, this thesis presents a model building on the success of the Transformer architecture to generate source code for software tests. However, since the before mentioned previous attempts at machine learning driven test generation did not yield any promising results [28, 29], it tackles the task of test completion instead. In test generation a entire test or test suite of multiple tests is generated given the method that should be tested (denoted as target method). Test completion is a subtask of test generation where a part of the test is given in addition to the target method, while the task is to generate the remaining part of the test. As this makes test completion a slightly less complex problem, this thesis will tackle it as a more approachable first step towards machine learning driven test generation.

More specifically, the approach presented in this thesis generates the part of the test asserting that the target method fulfills its requirements. This arguably is the test's part which requires the most understanding of the target methods requirements to generate. Since this also is the part which the previously mentioned heuristic approaches struggle the most at, filling this gap provides the most valuable addition to existing solutions.

1.1 Problem analysis

The following sections analyzes the problem domain, elaborating on terms and definitions relevant to this work, while highlighting what makes test completion a difficult task for machine learning models. Section 1.1.1 covers the different types of tests, clarifying which are in the focus of this research. Afterwards Section 1.1.2 analyzes the complexity underlying tests, carving out the challenges a machine learning model has to overcome to succeed at test completion. Building on this Section 1.1.3 introduces different types of complexity which tests can be classified with. Finally, the *Given/When/Then* pattern found in most tests is explained in Section 1.1.4. In this thesis this pattern is utilized as an archetype for segmenting tests into sections. This segmentation is then used to train the model to predict the *Then* section, given the others.

1.1.1 Test types

While automated software testing is a very complex and nuanced topic, with many different types of tests and testing methodologies, most fall in one of the following three categories [31, p. 307].

Unit tests: test a single use-case of one isolated unit. In an object-oriented context this unit usually is a method, or a function in a functional context. Due to the small unit under test, unit tests are good at isolating failures, whilst allowing for fast and cheap execution. However, they often require mocking components the tested unit relies on. This can decrease the meaningfulness of their results, as the artificial execution context created by the mocks can hide errors only occurring during interaction between those components. Therefore, while unit test may not be suitable for ensuring a unit is fulfilling its purpose in a system, they are often considered a contract defining requirements a unit has to comply to.

Integration tests: test the integration between different units or components, filling the blank spot left behind by unit tests. This includes components outside of the application, like databases or external APIs. Therefore, integration tests require running those external components during execution, making them slower and more expensive than Unit tests.

End-to-end (E2E) tests: test the entire system from one end to the other. In most cases this is implemented as a UI test, since the entrypoint for most systems is a UI. Therefore, the test is formulated without any knowledge of the system's internals. This could be something like pressing a button and testing for the expected result, disregarding what is leading to this result. While this allows for testing a system

from a user's point of view, E2E tests can't reveal which specific component is failing inside the application. Also, E2E tests require running all applications and components which are part of the tested system, making them particularly slow and expensive. Especially headlessly running the UI to execute the tests can slow down the testing process considerably.

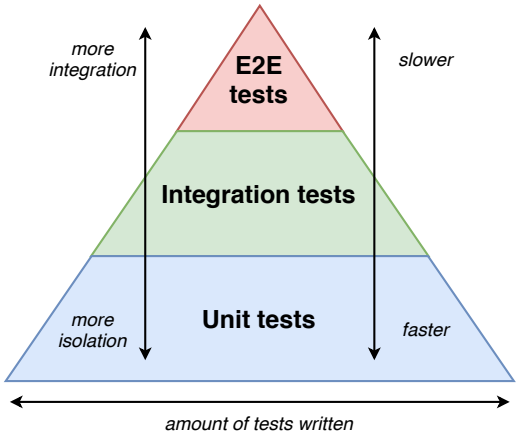


Figure 1.1: Illustration of the testing pyramid introduced by Mike Cohn in [31, p. 307]. It suggests the composition of a good test suite, relying mainly on unit tests, which are fast an cheap. Integration and E2E tests make up a smaller part of the suite, testing aspects not covered by Unit tests.

Since these type of tests cover different aspects of a system, a combination of all types is required to test reliably. However, as unit tests are the fastest to implement and execute, they usually make up the bulk of a test suite, with integration and E2E tests covering the rest. In practice a rule introduced by Google employees in [32] has prevailed. It suggests splitting up the amount of tests in the test suite into approximately 70% Unit, 20% Integration and 10% E2E tests. This rule builds upon the testing pyramid introduced by Mike Cohn in [31], which is illustrated in Figure 1.1.

E2E tests are also called Black-Box tests, as they test a system's behavior without looking inside the system. Contrary to that, Unit and Integration tests are considered White-Box tests, since they test for the application's internals to fulfill specific requirements. As Section 1.2 later specifies in more detail, this work researches whether a machine learning model can complete a test by understanding the system it is testing. This makes Black-Box tests not relevant to this research. Also, as illustrated by Figure 1.1, unit tests are the most common type of tests, which makes a machine learning model helping at creating them the most useful. Therefore, whenever the type of a test is not specified in this work, only White-Box tests are referenced (Unit and Integration tests).

1.1.2 Challenges

Completing tests poses multiple challenges to machine learning models. While most of these challenges might seem trivial to experienced software engineers, they expose considerable complexity when looked at closely, making test completion a learning problem far from trivial. This section analyzes the most notable challenges, investigating what is required from a machine learning system to succeed at them. All examples containing source code are written in *Java*, since the dataset employed in this work focuses on *Java* code, as later explained in Section 3.1.

Understanding syntax: first of all, a model must be able to interpret and generate valid source code. To do so, it has to understand the syntactic structure of source code.

Understanding types: beyond the syntax, a model must be able to infer an object's type. To generate compilable code, only methods part of its interface can be called on the object. Also, if an object of a given type is initialized, it can only be assigned to a variable of a compatible type. While the name of an object's type is easy to find, looking at how it is initialized, finding the corresponding class and inferring its interface is challenging for a machine learning model. But even determining the type's name can be more complicated, if a method returns an object not assigned to a variable. For example, in `assertEquals(getUser(), ...);` a model needs to infer the return type of `getUser()` to know what it can be compared to. Also, the behavior of arithmetic operators depends on the types used. In `a + b` the result could be an integer if `a` and `b` are. However, it could also be a string if one or both of them are strings. Therefore, inferring an object's type from the context it appears in, is an important skill for a model completing tests.

Understanding requirements: to formulate meaningful assert statements, a model needs to first understand the requirement the code under test is trying to fulfill. These can be inferred from the logic embedded in the tested source code, however, understanding this logic is a task often even difficult for humans. This can be complicated even further, if a method's logic is outsourced to multiple other methods it calls. To reveal the method's rationale, a model not only needs to understand what that method is doing, but also which methods it calls. It has to infer how those methods are defined and what their purpose is and then put the pieces together.

Even if a model is able to infer what a method is doing throughout a deep call stack, domain knowledge might be required to decode its purpose. If a method just executes a HTTP request to an external API, it can be hard to figure out its purpose without knowing what that API does. However, names of entities like variables, classes and methods can be helpful to infer information about the domain they are operating in. For example, even if the method `createUser()` just execute

a request to an unknown API, its name indicates that this request creates a user. Therefore, understanding entity names can allow a model to make out concepts not embedded in the code's logic.

1.1.3 Complexity types

To classify different tests based on the challenges they pose, three complexity types are introduced. These are built on the concept underlying the Integration Operation Separation Principle (IOSP) introduced by Westphal in [33]. This software engineering pattern introduces the idea that all of a software's complexity can be reduced by splitting it into integration and operation units. An operation unit exclusively contains logic and is not allowed to call other units. Contrary to that, integration units exclusively contain calls to other units, thereby integrating the logic implemented by operation units with each other. This separates the complexity introduced by integrating different units from the complexity underlying a system's logic. Using the idea behind this separation, this work takes the complexity in tests apart by introducing the following complexity types:

Integration complexity: describes the complexity introduced by integrating other units with each other. An example of this is shown in Listing 1.1. The things done by `Operator1` and `Operator2` are fairly simple. However, what makes the difficulty in predicting the outcome of `calc()` is the logic being separated in multiple units, with `Integrator` joining those. For the sake of keeping the example simple, the complexity is still fairly manageable in this case, however there is no limit to the amount of integrated units. The actual logic under test could be separated throughout multiple methods, classes and packages, making it increasingly complex to reconstruct its behavior, even if the complexity of the Operation units being integrated is not that high.

Operation complexity: describes the complexity introduced by an operation unit's logic. Since a operation unit's complexity is more nuanced, Operation complexity will be further divided into Algorithmic complexity and State complexity, as explained in the following.

Algorithmic complexity: as the name suggests, this describes the complexity introduced by the algorithm a operation unit implements. An example of this is shown in Listing 1.2. Here the blank can only be filled out by understanding that the algorithm implemented by `calc` computes the greatest common divisor of `a` and `b`.

State complexity: here the complexity doesn't arise from the operation unit's algorithm itself, but the state that this algorithm depends on. What makes filling out the blank in Listing 1.3 hard is not understanding what `get()` does, but how the lines

```
1  class Operator1 {
2     int calc(int a) {
3        return a * 2;
4     }
5  }
6
7  class Operator2 {
8     int getBaseValue() {
9        return 42;
10    }
11 }
12
13 class Integrator {
14    int execute() {
15       Operator1 o1 = new Operator1();
16       Operator2 o2 = new Operator2();
17       return o1.calc(o2.getBaseValue());
18    }
19 }
20
21 class IntegratorTest {
22    @Test
23    public void testExecute() {
24       Integrator integrator = new Integrator();
25       assertEquals(integrator.execute(), ???);
26    }
27 }
```

Listing 1.1: Example of a test showing high Integration complexity. Filling out the blank in line 25 is only possible by understanding how `Operator1` and `Operator2` are integrated to each other by `Integrator`. Although the complexity of the individual Operation units is low, they become more complicated by being integrated into each other.

```
1  class GcdCalculator {
2     static int calc(int a, int b) {
3        while (a != 0) {
4           int c = a;
5           b = c;
6        }
7        return b;
8     }
9  }
10
11 class GcdTest {
12    @Test
13    public void testCalc() {
14       int result = GcdCalculator.calc(8, 4);
15       assertEquals(result, ???);
16    }
17 }
```

Listing 1.2: Example of a test showing high Algorithmic complexity. Filling out the blank in line 15 is only possible by understanding that `calc` implements a greatest common divisor algorithm.

4-8 mutating the object's state influence its return value.

However, as most code does not strictly follow IOSP, the code testing it can't exclusively be classified by one of those complexity types. Therefore, a test's complexity is usually made up by varying degrees of each complexity type.

1.1.4 *Given/When/Then*

The *Given/When/Then* pattern (GWT) is an archetype for structuring tests. While it was introduced as part of Behavior-Driven Design by Terhorst-North et al. [34], it turned out applicable to almost all tests. GWT suggests that segmenting tests into the following three sections makes them more easily readable and understandable:

***Given*:** defines the preconditions required to achieve the expected result. This usually means setting up the state, by initializing objects needed to execute the test.

***When*:** defines what has to be executed to achieve the expected result, given the preconditions described in the *Given* section. This section only consists of the target method's invocation. Meaning, it executes the code that is being tested.

***Then*:** defines how to test whether the result meets the expectation. Usually this is done using assert-statements provided by a testing library or framework.

```
1  @Test
2  public void testGet() {
3     List<Integer> list = new ArrayList<Integer>();
4     list.add(1);
5     list.add(2);
6     list.remove(1);
7     list.add(3);
8     list.remove(0);
9     assertEquals(list.get(0), ???);
10 }
```

Listing 1.3: Example of a test showing high State complexity. While the behavior of `get()` is not algorithmically complex, it is dependent on how the state of `list` is mutated by the lines 4-8.

This segmentation is illustrated in Listing 1.4 showing a test implemented in *Java* using the *JUnit*[1] testing framework. In this example the *Given* section is highlighted in yellow, *When* in red and *Then* in green. This color coding will be used in all listings throughout this work where GWT is relevant.

```
1  class TestArrayList {
2     @Test
3     public void testIsEmpty() {
4        List<Integer> list = new ArrayList<Integer>();
5        boolean result = list.isEmpty();
6        assertTrue(result);
7     }
8  }
```

Listing 1.4: Simple test targeting `ArrayList.isEmpty()`. Line 4 builds up the state (*Given*), line 5 calls the target method (*When*) and line 6 asserts the result to be as expected (*Then*).

It is worth noting, that this concept is also known as the *Arrange/Act/Assert* (AAA) pattern [35]. However, the underlying idea is equivalent to GWT, with *Arrange* replacing *Given*, *Act When* and *Assert Then*. This work will use the GWT terminology.

[1] `https://junit.org`

1.2 Research goals

The goal of this work is to explore the potential of machine learning driven test completion. More specifically, the GWT pattern introduced in the previous section is used to segment a test and a machine learning model is trained to predict its *Then* section, given the *Given* and *When*.

Since there is no sufficiently sized dataset available said model could be trained on, this work first aims at creating such a dataset. This dataset maps test methods to their target methods, while segmenting them using the GWT pattern. By contributing this dataset to the research community, the results of this research can be reevaluated with better models the future may bring.

The other major contribution of this work is the implementation of the before mentioned test completion model, based on the Transformer architecture, which has recently been employed with great success in the field of Natural Language Processing [11]. While there has been previous research on machine learning driven test generation [28, 29, 30], this is the first attempt at applying machine learning to test completion, to the best of the author's knowledge. Contrary to the previous test generation approaches [28, 29, 30] the model proposed in this work uses Abstract Syntax Trees (Section 2.4) to encode the source code inputted into the model, which has showed promising results for related models trained on other software-engineering tasks [36, 26, 37, 25].

As explained in Section 1.1.2, information needed to complete a test is often not embedded in the test itself, but its context (other methods it is using). Therefore, this work innovates on previous test generation approaches [28, 29, 30] by adding declarations of methods to the model's input, which are called by the test and target methods.

To evaluate the potential of machine learning driven test completion, experiments are conducted to quantitatively assess the model's ability to generate meaningful *Then* sections. Also a qualitative evaluation will explore if the model is able to handle the challenges of test completion described in Section 1.1.2 and how it performs on different tests depending on their complexity types. Furthermore, it will be evaluated if the model is able to generate parsable source code and leverage the syntactic information embedded in Abstract Syntax Trees. Another part of this research is to analyze if the model is able to understand how the behavior of a test depends on methods in its context. More specifically, if the model's performance can be improved by adding context information to its input, or if it will suffer as the model is unable to cope with the larger input.

Thus, the goal of this work is to answer the following **research questions** in regards

to the previously mentioned dataset and machine learning model:

RQ1: Is the quality of the test completion dataset, which has been automatically generated from a unlabeled codebase, sufficient to train a machine learning model on?

RQ2: What is the model's ability to generate parsable source code?

RQ3: What is the model's ability to overcome the challenges of test completion and generate meaningful *Then* sections?

RQ4: What impact does using Abstract Syntax Tree in the model's input have on its performance?

RQ5: Is the model able to understand the context a test is in, if provided with context information and what impact does this additional information have on the model's performance?

1.3 Outline

This work is structured into five chapters: Chapter 2 will provide background information on Natural Language Processing approaches most relevant to this thesis and give a brief overview of related work. Based on this information, Chapter 3 presents the approach chosen by this work to investigate the research questions posed in the previous section. This approach includes creating a dataset labeled for test completion and training a machine learning model on it. Using this model, Chapter 4 conducts extensive experiments to evaluate its abilities and disabilities, thereby evaluating how the chosen approach answers the posed research questions. Finally, Chapter 5 concludes on the results presented in Chapter 4 and discusses how the findings of this thesis could be employed by future work.

Chapter 2

Background

Although test completion is a software-engineering task, the approach proposed by this work (Chapter 3) mainly employs machine learning concepts which originated from the field of Natural Language Processing (NLP). Therefore, this section gives an overview of the recent advances in NLP most relevant to this thesis.

A central concept of the proposed approach is that source code is represented by a sequence of tokens, as common in NLP. Such a sequence can then be processed by so-called sequence-to-sequence models. Section 2.1 covers the general idea behind these sequence-to-sequence models and the encoder-decoder architecture they usually deploy. Building on this architecture, the Transformer model is explained in Section 2.2, which arguably is one of the most influential NLP models proposed in recent years. Finally, Section 2.3 explains how byte-pair encoding can be used for subword tokenization, followed by Section 2.4 giving a brief overview of related work.

2.1 Sequence-to-Sequence Models

As the name suggests, sequence-to-sequence models describe a class of machine learning architectures which take a sequence $x = (x_1, \dots, x_n)$ as input and output another sequence $y = (y_1, \dots, y_m)$, where n denotes the length of x and m that of y. One common use case for this type of models is Neural Machine Translation (MNT). Here x is a sequence of words in the source language, while y is the corresponding sequence of translated words in the target language.

To make these words processable by machine learning models they are commonly mapped into a vector space of a fixed-length d, using word embeddings pre-trained by Word2Vec [38] or GloVe [39] for example. While different in detail, these tech-

niques share the general concept of representing a word as a vector $w \in \mathbb{R}^d$, with a minimal euclidean distance to other vectors representing words which commonly occur in a similar context. For example, the word embedding vector for the word *King* most likely has a minimal distance to the vector representing the word *Queen*, as those words are likely to occur in a sentence together. Thus, $x \in \mathbb{R}^{d \times n}$ and $y \in \mathbb{R}^{d \times m}$.

While the term "sequence-to-sequence" was coined by Sutskever et al. in [12], the concept builds on the Encoder-Decoder architecture previously proposed by Cho et al. [13]. The general idea behind this architecture is that an encoder takes x as an input, compressing (encoding) it into a context vector c, which embeds the information relevant to x. This vector is then forwarded to the decoder, decoding the embedded information into the target language based on the information encoded in c.

To be able to do so, the decoder is trained to predict the next word $y_{t'}$ for the time step t' given c and the previously predicted words $(y_1, \dots, y_{t'-1})$. Therefore, the decoder defines a probability over y as the joint probability

$$P(y) = \prod_{t'=1}^{m} P(y_{t'} \mid (y_1, \dots, y_{t'-1}),\ c).$$

The initial approach presented in [13] employs a variant of a Recurrent Neural Network (RNN) [40] to implement the encoder and decoder. In this case, x_t is inputted into the RNN for $t \in \{1, \dots, n\}$ encoding time steps. The hidden state resulting from the last time step h_n is then used as the context vector forwarded to the decoder, thus $c = h_n$. Subsequently, the RNN is used to predict $y_{t'}$ in the decoding time step $t' \in \{1, \dots, m\}$, with its input being the prediction of the previous time step $y_{t'-1}$ together with c and the previous hidden state $h'_{t'-1}$. At the first decoding time step $t' = 1$ the input usually is a start-of-sentence token (like *SOS*), representing the start of the target sentence. An example of this is illustrated in Figure 2.1.

Since RNNs are known to suffer from the vanishing gradient problem [41], [13] introduced Gated Recurrent Units (GRU), a RNN variant controlling the information stored in the hidden state. This idea is similar in concept to Long Short-Term Memory (LSTM) networks presented earlier by Hochreiter and Schmidhuber in [42], while being simpler to compute and implement [13]. However, in [12] an LSTM is employed in an encoder-decoder architecture, outperforming the GRU based model proposed in [13] in various NMT tasks.

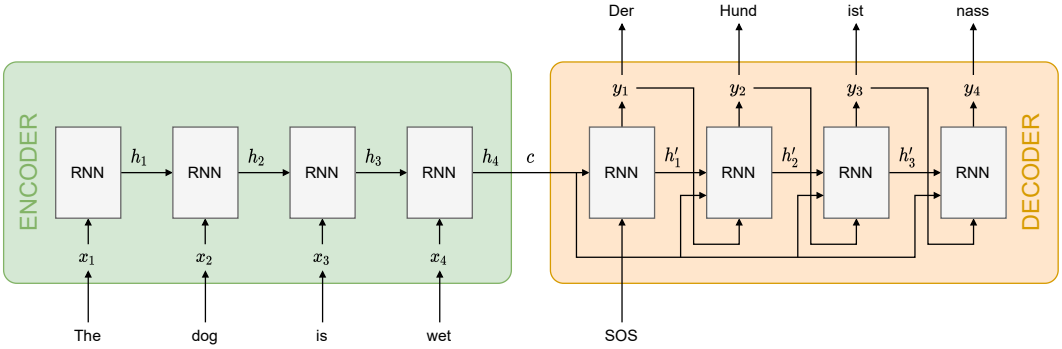


Figure 2.1: Illustration of a RNN encoder-decoder translating the English sentence "The dog is wet" into German. The last hidden state resulting from the encoder (left) is used as the context vector c passed into the decoder (right), thus $c = h_4$. In the decoder the output of the previous time step $y_{t'-1}$ is used as the input for the RNN in the current time step t', together with c and the previous hidden state $h'_{t'-1}$.

2.1.1 Attention

While LSTMs and GRUs are able to combat the vanishing gradient problem whilst improving long-term memory, both suffer from a similar problem when employed in a encoder-decoder architecture. Due to the sequential nature of RNN based models, a long sequence's hidden state at the last time step is likely to have lost information about the words from early time steps. Since the output sequence is generated from the last hidden state outputted by the encoder, recurrence based encoder-decoder models increasingly struggle to recognize global dependencies between input and output sequences the longer they are.

To resolve this problem, Bahdanau et al. introduced the attention mechanism in [14], which was then refined by Luong et al. in [15]. The intuition behind this mechanism is to quantify how important a given word x_t in the input is to another word $y_{t'}$ in the output with the attention score $A_{t't} \in [0, 1]$. This results in the attention matrix $A \in \mathbb{R}^{m \times n}$ with

$$\sum_{t=1}^{n} A_{t't} \;=\; 1$$

for all decoder time steps $t' \in \{1, \dots, m\}$. Figure 2.2 exemplarily visualized the attention matrix generated for an English sentence translated to French [13].

Contrary to only using h_n during decoding, A allows for taking all encoder hidden states into account, as it indicates how much *attention* should be payed to the hidden state h_t while predicting $y_{t'}$. Therefore, a separate context vector $c_{t'}$ is defined for each decoding time step t' as the sum of all encoder hidden states weighted by $A_{t't}$:

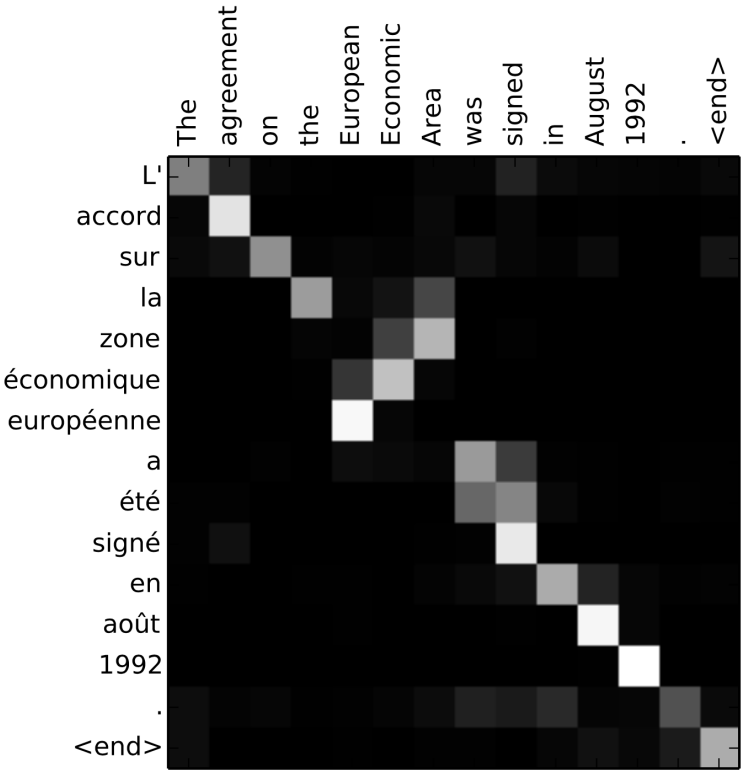


Figure 2.2: Example of the attention matrix A generated for a English source sentence (x-axis) translated to French (y-axis). Each pixel represents the attention score $A_{t't} \in [0, 1]$ for the t-th source and t'-th target word. White depicts the score 1 and black 0. [14]

$$c_{t'} := \sum_{t=1}^{n} A_{t't} h_t.$$

The decoder's output $y_{t'}$ is then predicted in respect to $h'_{t'-1}$ and $c_{t'}$.

While multiple approaches to generating A have been proposed in previous research [14, 15], Section 2.2.2 later explains the scaled dot-product attention, which is most relevant to this work.

2.2 Transformers

Although recurrence based sequence-to-sequence models like [13] and [12] yielded results revolutionary at their time, more recently the Transformer architecture presented by Vaswani et al. in [11] has set the stage for a new generation of natural language models like BERT [16], the GPT models [21, 20, 19] and RoBERTa [17]

taking NLP to a new level.

The novel concept responsible for the success of the Transformer architecture is its abandonment of any form of recurrence or convolutions, solely relying on the attention mechanism to draw global dependencies between, as well as within, input and output [11]. This circumvents many of the disadvantages associated with the sequential nature of RNN based models. As obtaining a hidden state h_t requires processing its predecessor h_{t-1} first, RNNs are nearly impossible to effectively parallelize within training examples [11]. Being able to process inputs in parallel allows the Transformer to be trained in a fraction of the time required for training previous LSTM based approaches [11]. However, due to the self-attention mechanism explained in the following section, the memory used by the transformer architecture is quadratic on the length of the processed sequences. Therefore, although Transformer models can be trained much more efficiently than LSTMs, the size of the input sequence is constrained by the available memory.

2.2.1 Self-attention

The Transformer not needing any form of recurrence is enabled by a variation of the attention mechanism (Section 2.1.1) called self-attention. While preserving the general concept of the attention mechanism, self-attention measures attention between words of the same sentence. In the Transformer architecture this is done for the encoder's and decoder's input, allowing it to catch relations within the input and output sentences. The intuition behind self-attention is illustrated in Figure 2.3 where the high attention of the word "it's" towards "dog" indicates that it is referencing it.

Figure 2.3: Illustration of the self-attention between the word "it's" and other words in the same sentence. That "it's" references the word "dog" is indicated by a high attention towards it.

Also know as intra-attention, self-attention is not a concept novel to the Transformer. It has successfully been employed in various tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations [43, 44, 45, 46, 11]. However, the Transformer is the first architecture entirely relying on self-attention to encode relations in its input and output, instead of using sequentially created hidden states [11].

2.2.2 Scaled Dot-Product Attention

Another contribution by [11] is the introduction of the scaled dot-product attention, a new method of calculating attention. The general idea behind this is to map a *query* and a corresponding *key-value* pair to an output. How relevant a *value* is to the *query* is measured by computing the similarity between the *query* and *key*. This is done by calculating the dot-products of the *query* and all *keys*, which both are of dimension d_k. The result is divided by $\sqrt{d_k}$ to make it independent from d_k. Applying a $softmax$ to the output results in a score for each *key* in the interval $[0, 1]$. However, as the $softmax$'s gradients are small for high input values, the dot-product is scaled by $\frac{1}{\sqrt{d_k}}$ before applying the $softmax$. Weighting the *values* of dimension d_v with this score yields the final output.

In practice, formulating this as a set of matrix operations allows for computing the attention function on multiple *queries* simultaneously. Therefore, the scaled-dot product attention is defined as

$$Attention(Q, K, V) := softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)\ V.$$

with Q, K and V being matrices containing all *queries*, *keys* and *values* respectively [11].

The main advantage of scaled dot-product attention compared to previous approaches to calculating attention [14], is it relying on matrix multiplication. This makes it much faster and space-efficient, as GPUs are highly optimized for matrix multiplication [11].

2.2.3 Multi-head Attention

One of the key contributions in [11] is multi-head attention, which is essential to enabling the Transformer to capture complex relations solely using attention. The key idea here is that attention for the same input is computed in multiple representation subspaces, also known as attention heads. Each attention head captures different relations in the sentence, which are then merged into a final representation, capturing a higher level of complexity than a single attention head could.

To do so, $Q \in \mathbb{R}^{m \times d_{model}}$, $K \in \mathbb{R}^{n \times d_{model}}$ and $V \in \mathbb{R}^{n \times d_{model}}$ are linearly projected into each of the subspaces using different projection matrices jointly learned with the model. The size of the input embedding is denoted as d_{model} while n is the size of the encoder's input and m that of the decoder. The previously explained scaled

dot-product attention is then applied to each of those projections and the outputs are concatenated before applying a last linear projection. In [11] this process is illustrated with Figure 2.4.

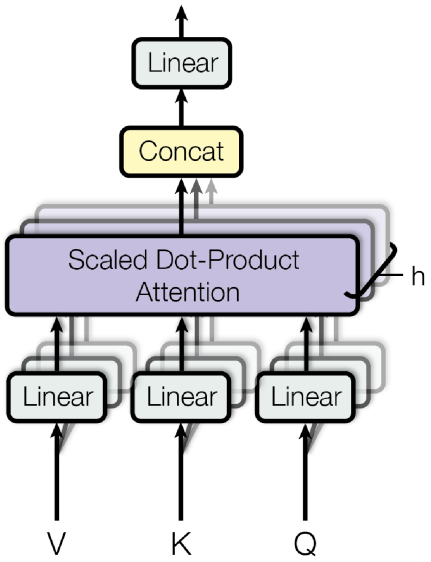


Figure 2.4: Illustration of multi-head attention for h attention heads. V, K and Q are linearly projected before applying scaled dot-product attention. The output of all heads is then concatenated before applying another linear projection. [11]

Therefore, the multi-head attention for h attention heads is defined as

$$MultiHead(Q, K, V) \coloneqq Concat(head_1, \ ... \ , head_h) \cdot W^O$$

with

$$head_i \coloneqq Attention(QW_i^Q, \ KW_i^K, \ VW_i^V)$$

and the projection matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$ for Q, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ for K, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ for V and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ projecting the concatenated output [11].

2.2.4 Architecture

Figure 2.5 illustrates the architecture employed by the Transformer, putting the building blocks described in the Section 2.2.1, 2.2.2 and 2.2.3 together.

At its core the Transformer uses the same encoder-decoder approach described in Section 2.1, The input $x = (x_1, ... \ , x_n)$ of length n is processed by the encoder, outputting its continuous representation $z = (z_1, ... \ , z_n)$. Given z and the tokens

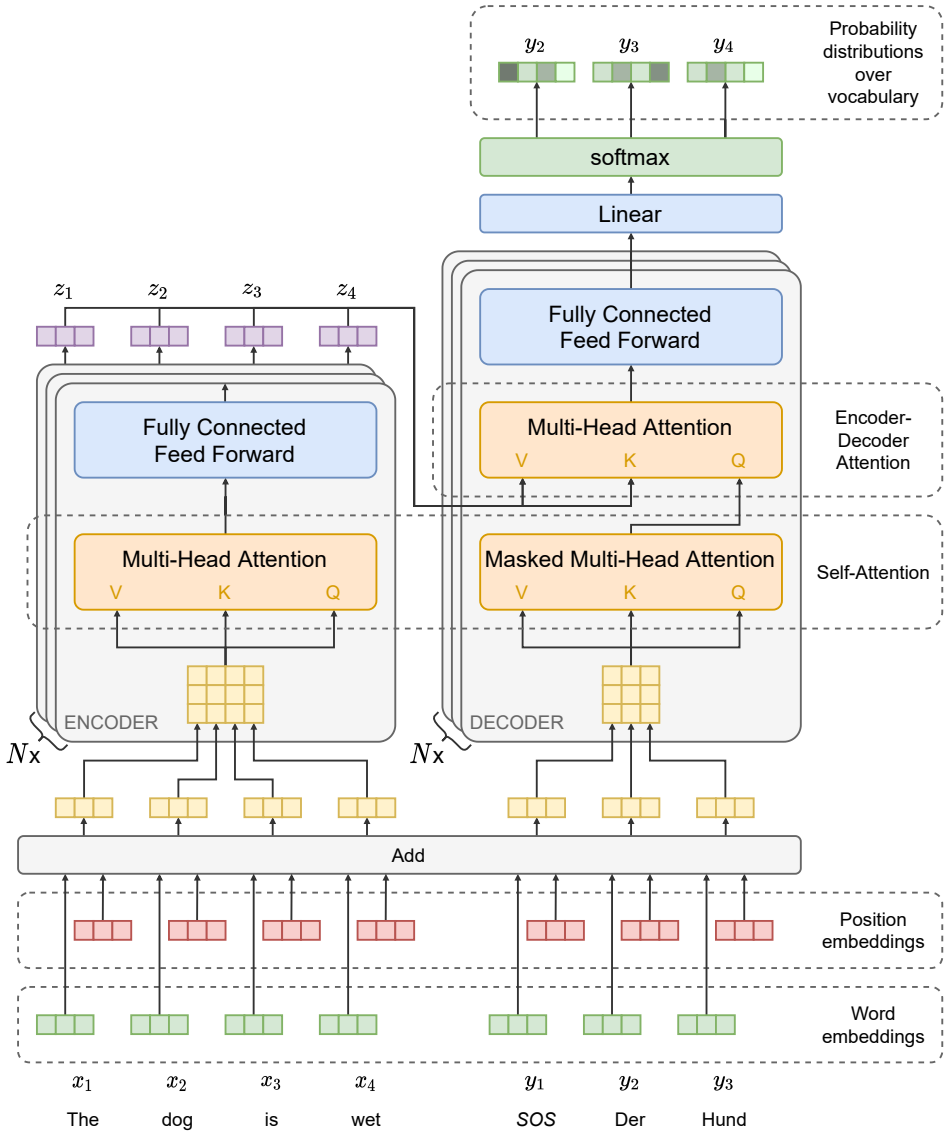


Figure 2.5: Overview of the key components of the Transformer's architecture. Word and positional embeddings are used to retrieve vector representation in $R^{d_{model}}$ of the tokens in x and $y_{1:3}$ (bottom). The encoder applies self-attention followed by a position-wise fully connected feed forward network to the input N times in a row (left). Its final output z is used in all N decoder instances to compute the encoder-decoder attention together with the output of the masked self-attention, which is retrieved from the decoders input (right). After another pass through a fully connected feed forward network, the decoders output is passed through a linear layer and the $softmax$ is used to retrieve a probability distribution over the target vocabulary V for each token in $y_{2:4}$ (top). Note that the residual connections and layer normalizations within the sublayers are not illustrated here for the sake of simplicity. [11]

generated in the previous $t' - 1$ steps $y_{1:t'-1} = (y_1, \dots, y_{t'-1})$, the sub-sequence $y_{2:t'}$ of the target sentence $y = (y_1, \dots, y_,)$ is outputted by the decoder. Contrary to the RNN described in Section 2.1, the entire input sequence x and sequence of previous predictions $y_{1:t'-1}$ is processed in each forward pass. The output of the decoder is of the same length as its input $y_{1:t'-1}$, however it is shifted by one position, hence the output being $y_{2:t'}$.

The encoder and decoder are each stacked N times, forming the encoder- and decoder-blocks. While the first encoder and decoder in these blocks receive x or $y_{1:t'-1}$ respectively as an input, the following encoders/decoders process the output of their predecessor. The output of the last encoder in the encoder block is used as z in the decoder block.

The following subsections highlight the individual components shown in Figure 2.5, describing their role in the architecture.

Encoder/Decoder Input

The tokens in the encoder's and decoder's input are converted to vectors of dimension d_{model} using embeddings learned jointly with the model. However, as the Transformer does not process these embeddings sequentially, contrary to RNNs, it has no notion of a tokens position in the sequence. As the position of a token is especially relevant in NLP, the input embedding is enriched with a positional embedding. These are vectors of the same size as the word embeddings d_{model}, allowing them to be added.

While there are different methods for computing positional embeddings [47], the Transformer in [11] employs sine and cosine functions of different frequencies, making each dimension $i \in \{1, \dots, d_{model}\}$ correspond to a sinusoid, using

$$PE_{(pos,\, 2i)} := sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,\, 2i+1)} := cos(pos/10000^{2i/d_{model}})$$

where pos is the position that is being encoded.

Also, a dropout is applied to the sum of the word and positional embedding before inputting it into the encoder/decoder.

Encoder

The encoder employs two sub-layers, starting with a self-attention layer. Self-attention (Section 2.2.1) is realized using a multi-head attention layer (Section 2.2.3) where V, K and Q all come from the encoders input sequence, thereby assessing attention between tokens of that sequence. That input sequence of the encoder is the model's input sequence after the word and positional embeddings have been applied. The second sub-layer is a position-wise fully connected feedforward network.

As the output of a encoder is fed into the input of the next one in the encoder block, its output dimensions have to be compatible to its input. Therefore, all sub-layers in the model generate outputs of dimension d_{model}. Also, a dropout is employed on the output of all sub-layers before applying layer normalization [48] and adding it to a residual connection [49, 11].

Decoder

The decoder uses three sub-layers. Similarly to the encoder, a self-attention layer is employed as the first sub-layer, using the decoders input $y_{1:t'-1}$ for K, V and Q. However, as self-attention generally allows for attending to all tokens in the input, it has to be modified to prevent tokens from attending to subsequent ones, to preserve the auto-regressive property of the model [11]. This is achieved in the scaled dot-product attention by masking values which attend to a position greater or equal than the currently predicted position t'. Masked values are set to $-\infty$, due to the scaled dot-product attentions use of the $softmax$ function and $softmax(-\infty) = 0$. Section 2.2.5 later explains why this mechanism is crucial to the training efficiency of Transformers.

The second and third layer are almost equivalent to the sub-layers of the encoder, with the attention layer not being a self-attention layer. Here K and V are sourced from z, while Q is the output from the previous self-attention layer. As this allows the output sequence to attend to the context vector z resulting from the encoder, it works more like the traditional attention mechanism typically employed in sequence-to-sequence models (Section 2.1.1). Since this calculates attention between the encoder's output and the decoder it is also known as the encoder-decoder attention [11].

The dimension size, as well as the employed residual connections, dropout and layer normalization described for the encoder also applies to the decoder's sub-layers.

Decoder output

A linear layer is used to map the decoder block's output for each token to a vector of logits in $\mathbb{R}^{|V|}$, with V being the target vocabulary. By applying a $softmax$ to those logits the model outputs a probability distribution over V.

2.2.5 Training

As the Transformer model presented in [11] is a supervised learner, it is trained on a dataset of source and target sequences pairs. Since it outputs a probability distribution [11] employs the cross-entropy loss based on which the model's gradients are calculated. Those gradients are then used by the *Adam* optimizer [50] to update the model's parameters. These parameters include the word embeddings, the projection matrices used in the multi-head attention layers, as well as the weights of the feed forward networks and the linear output layer.

As predicting the token $y_{t'}$ requires the Transformer to know the previous token $y_{t'-1}$, it could be assumed that it ends up suffering from the same limitations sequential models like RNNs do. While this is true when applying the model to unlabeled data, this is not the case during the computationally more critical training phase. As the Transformer always processes the entire sequence of previous tokens $y_{1:t'-1}$ instead of only the last token $y_{t'-1}$, it is able to utilize *teacher forcing* [51] during training. This means that instead of predicting the sequence of previous tokens $y_{1:m-1}$ in $m-1$ preceding steps, the final sequence $y_{2:m}$ is predicted using the shifted target sequence $y_{1:t'-1}$ from the dataset as the decoder's input. While doing so, the masking in the decoder's self-attention layer becomes very crucial. As it prevents a given token from attending to any subsequent tokens, it hinders the model from learning to simply copy the decoder's input.

2.3 Byte-Pair Encoding

A limitation of training word embeddings like Word2Vec [38], GloVe [39], or even those jointly trained with the Transformer [11], is that the vocabulary is defined by the data they are trained on. If a model using such embeddings then encounters an out of vocabulary word, it can only be represented by a unknown token, possibly hurting the model's performance. This poses the challenge of having to choose between training embeddings on a very large vocabulary, thereby increasing their complexity and computational cost, or only including the most frequently used words into the vocabulary, accepting the increased likelihood of unknown tokens.

This problem becomes even more severe when applying NLP approaches to source code, as done in this thesis. Since entities in source code are named by joining describing words, the resulting entity name could be considered a single word. Therefore, even if the two method names *getUser* and *getID* would be part of a vocabulary, the method *getUserID* could be out of vocabulary, although it contains the same subwords. However, including all possible combinations of entity names in a vocabulary is impossible.

In [52] Sennrich et al. presented a solution to this problem, based on *byte-pair encoding* (BPE) proposed by Gage in [53]. While BPE was initially created as a data compression algorithm [53], Sennrich et al. found it to be applicable to subword tokenization [52]. Doing so allows for fitting a vocabulary of subwords on a dataset, which can be used to recreate every word in the dataset, as well as individual words which are not part of it. The algorithm initializes the vocabulary with all characters in the dataset and then replaces the characters in the dataset making up the most frequently occurring character pairs with that character sequence, while adding the sequence to the vocabulary. This process is repeated until there are no pairs left to be replaced, or a previously set vocabulary size is reached. Therefore, often used words end up as a single subword in the vocabulary, while uncommon words can still be constructed from multiple smaller subwords.

Table 2.1 illustrates the BPE algorithm using the previously mentioned example, where the dataset contains the set of method names $S = \{getUser, getID, getUserID\}$. The vocabulary is then initialized as $V_0 = \{\_g, e, t, U, s, r, I, D\}$, where the underscore marks characters at the beginning of a word. Using V_0, the set of subword sequences making up S is defined as $S_{V_0} = \{(\_g, e, t, U, s, e, r), (\_g, e, t, I, D), (\_g, e, t, U, s, e, r, I, D)\}$. As Table 2.1 shows, V_0 is updated in six steps ultimately ending up as $V_6 = \{\_g, e, t, U, s, r, I, D, \_ge, \_get, Us, Use, User, ID\}$.

i	**Replaced pair**	S_{V_i}
1	$(\_g, e) \rightarrow \_ge$	$\{(\_ge, t, U, s, e, r), (\_ge, t, I, D), (\_ge, t, U, s, e, r, I, D)\}$
2	$(\_ge, t) \rightarrow \_get$	$\{(\_get, U, s, e, r), (\_get, I, D), (\_get, U, s, e, r, I, D)\}$
3	$(U, s) \rightarrow Us$	$\{(\_get, Us, e, r), (\_get, I, D), (\_get, Us, e, r, I, D)\}$
4	$(Us, e) \rightarrow Use$	$\{(\_get, Use, r), (\_get, I, D), (\_get, Use, r, I, D)\}$
5	$(Use, r) \rightarrow User$	$\{(\_get, User), (\_get, I, D), (\_get, User, I, D)\}$
6	$(I, D) \rightarrow ID$	$\{(\_get, User), (\_get, ID), (\_get, User, ID)\}$

Table 2.1: Example of how the set of subword sequences S_{V_i} for $S = \{getUser, getID, getUserID\}$ changes, while the vocabulary V_i is updated in each iteration i of the BPE algorithm.

Using BPE, word embeddings can be trained on the vocabulary of subwords, enabling a model employing these embeddings to handle words which are not part of

the dataset's vocabulary. Besides avoiding unknown tokens, dividing words into subwords can also help the model to understand their similarities to other words containing similar subwords [52]. For example, since *getUserID* and *getID* both contain the subwords *get* and *ID*, it could be more obvious to a model that their meanings are related.

2.4 Related work

Although there has been extensive research on applying machine learning approaches which originated from NLP to various software engineering tasks, doing so to generate or complete tests is fairly unexplored. However, many non-machine learning test suite generation tools have been introduced, solely relying on static code analysis and heuristics. Therefore, this section will give an overview of previous research relevant to this thesis, while separating approaches employing machine learning from those which do not.

Non-Machine Learning Approaches

There are two types of approaches towards non-machine learning based test suite generation: *Random Software Testing* [6, 7, 8] and *Search-Based Software Testing* [4, 5, 54]. Most of these approaches have exclusively focused on *Java* source code, due to its popularity [55] and its bytecode being easily analyzable.

***Random Software Testing*:** the implementation of a *Random Software Testing* approach which has shown the most success is *Randoop* presented by Pacheco et al. in [6]. Given a target class *Randoop* pseudo-randomly generates sequences of method/constructor invocations for the target class [6]. By analyzing the bytecode of the code under test, it explores the different execution branches. Through the use of various heuristics the ranges from which random values are sampled are optimized to increase the likelihood of covering those branches [6]. By executing the generated test and extracting the value returned by the target method given the generated parameters, assertion statements are added which assert the target method's return value to be equal to the extracted value. Meaning, the target method's return value is used to formulate what its expected behavior is. Using this approach, different test suite versions are generated and executed while measuring the code coverage they achieve. The test suite which covers the most execution branches of the code under test is chosen as the most suitable one.

***Search-Based Software Testing*:** in this class of approaches test suite generation is considered a local search problem, where a optimal solution in a search space

of candidate solutions (all possible tests or test suites) has to be found. To do so, metaheuristic local search techniques like genetic algorithms [4, 54] or *tabu search* [5, 56] are employed.

The *Search-Based Software Testing* approach which has proven to be most successful [57] is *EvoSuite* by Campos et al. [4]. Using the given target classes bytecode, *EvoSuite* initializes a fixed-size population of candidate solutions by pseudo-randomly generating test suites, similarly to how *Randoop* does. This population is evolved by mutating the test suites in various ways, like changing parameters and adding/removing statements. Once again various heuristics are applied to the target method to select suitable mutations [4]. The fitness of the candidate solutions is measured by their code coverage. After each evolution candidates with a poor fitness are dropped from the population, while promising candidates covering different branches can be combined. This process is repeated until the optimal solution has been found, or the amount of evolutions exceeds a previously set limit.

After the search is completed, the test suite is executed multiple times while iteratively removing individual statements from the tests. By analyzing if removing a statement changes the test's coverage, statements can be dropped which have no impact on it.

Since the selected solution is only optimized for code coverage at this point, it does not contain any assert statements. Similarly to *Randoop*, assertions are generated from the return values of the target methods and other public methods of the target classes interface which return something. As all methods in the target classes interface are included, this results in a large amount of assert statements with most of them testing irrelevant state. To find identify relevant assertions, random mutations are added to the target method which introduce a fault (like changing >= to <=). By doing so, assertions which fail after the introduction of a mutation are proven to be relevant to testing the target method's behavior. This is repeated for multiple mutations and irrelevant assertions are removed from the test suite.

Disadvantages: while approaches like *Randoop* and *EvoSuite* have proven their ability to generate test suites with decent coverage [57], they have considerable disadvantages holding them back from being relevant for practical use. First of all, the way test suites are generated by theses approaches is fairly inefficient. Especially *EvoSuite* requires repeatedly executing generated tests throughout many evolutions, which makes generating test suites for multiple classes very computationally expensive.

However, more severe disadvantages were revealed in [9] where Almasi et al. surveyed developers on the quality of test suites generated by *EvoSuite* and *Randoop*, among manually created test suites. The vast majority of the respondents easily

identified the machine generated tests and stated that their readability leaves much to be desired. But most importantly, many developers reported that the systems failed to generate meaningful assert statements. This observation has also been confirmed by the results presented in [10], where Shamshiri et al. conducted a study on *EvoSuite*, *Randoop* and the commercial *Search-Based Software Testing* solution *AgitarOne* [54]. In this study they assessed if test suites generated by them are able to reveal bugs in the tested software. They found that none of tested approaches were able to find more than 40.6% of the bugs, although 63.3% of the undetected bugs were covered by the generated tests [10]. This highlights that although the generated tests cover relevant execution branches, they often fail to assert whether those behave as expected.

Machine-Learning approaches

As mentioned before, previous work has conducted extensive research on employing machine learning approaches at software-engineering tasks. The following list will give a brief overview of related work, demonstrating that the NLP techniques described in the previous sections are not only suitable for natural language but also source code:

- **Code completion/suggestion:** Alon et al. [23], Allamanis et al. [58], Parr et al. [59].
- **Finding/fixing bugs in source code:** Bhatia et al. [60], Devlin et al. [61], Ray et al. [62].
- **Detecting code similarities:** Tufano et al. [24], Zorin et al. [27], White et al. [63].
- **Generating source code from natural language:** Yin et al. [36], Kusupati et al. [64].
- **Generating comments from source code:** Zheng et al. [65], Hu et al. [25].
- **Translating source code between programming languages:** Chen et al. [26], Lachaux et al. [22].

While these works are very different in detail, there are two takeaways most relevant to this thesis. Firstly, they show that sequence-to-sequence models can be used on source code with great success. Secondly, while initial approaches have converted source code to sequences by tokenization, as commonly done in NLP, more recent

research has shown that using Abstract Syntax Trees as the model's input can improve its performance [36, 26, 37, 25].

The Abstract Syntax Tree (AST) of a piece of code is a tree representation of its abstract syntactic structure. An example of this is illustrated in Figure 2.6. While each node in the tree represents a construct like control structures occurring in the source code, the tree's leaves represent values. Therefore, AST leaves are referred to as value nodes in this thesis. As an AST encodes the syntactic structure of the corresponding code, it can provide machine learning systems with additional information to learn from.

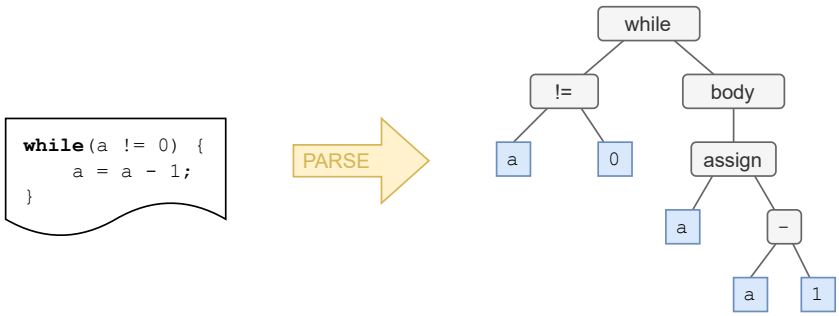


Figure 2.6: Simplified version of an Abstract Syntax Tree (AST) on the right, generated from the code on the left. Value nodes in the AST are highlighted in blue.

More specifically than the previously mentioned related works, the author is not aware of any attempts at applying machine learning to test completion. However, there have been a few on the more complex task of test generation.

Saes [28]: the first attempt at test generation using machine learning was proposed by Saes in [28]. To be able to train a machine learning model on this task he contributed a dataset mapping methods to other methods testing them. The data used to create this dataset was retrieved from GitHub. However, due to the rate limitations of the GitHub API [66], only 3,385 projects were included. Using bytecode and AST analysis those projects were analyzed, while employing heuristics to identify test methods and their corresponding targets [28]. This yielded a dataset containing 52,703 datapoints [28].

The model employed is a sequence-to-sequence encoder-decoder (Section 2.1) using an LSTM as suggested in [12]. The input sequence represents the method under test and the output sequence the test method. Saes presented different methods of encoding those methods as sequences, but experiments have shown tokenizing the code into individual tokens to be most successful. He also suggests representing the methods by their ASTs, while using the SBT method proposed by Hu et al. in [25] to convert that AST into a sequence of tokens. However, in the conducted experiments the increased input size of the AST sequence made training the LSTM

computationally infeasible [28]. While limiting the length of the input sequences to 100 enabled the best performing model to generate parsable code in 86.69% of cases, no model was able to generate any meaningful tests.

White & Krinke [29]: shortly after [28] White & Krinke released a model called *TestNMT* in [29]. The general idea here is to use NMT techniques to "translate" a method to another method testing it. They created a similar dataset as Saes [28] using comparable heuristics, however, they achieved a considerably larger dataset with about 744,000 datapoints.

They also employed a sequence-to-sequence model comparable to [28], but they used a RNN with an attention mechanism (Section 2.1.1) instead of an LSTM. Before being inputted into the model the target method is tokenized by splitting it on spaces and uppercase characters (because of camel-cased variable names) before all tokens are converted to lower-case. However, during evaluation the model trained on the created dataset only achieved a BLEU score of 1.3 [29]. This BLEU score indicates that the model is not able to generate any meaningful tests. They also did not evaluate if the sequences outputted by the model represented parsable code.

Tufano et al. [30]: parallel to this thesis Tufano et al. released a model for test generation called *AthenaTest* in [30]. They also created a new dataset mapping methods to others testing them, which they published under the name *Methods2Test*[1]. This dataset contains about 630,000 datapoints created from more than 70,000 GitHub projects, using a similar heuristic based approach as [29] and [28].

The employed model *AthenaTest* utilizes the *BART* architecture presented by Lewis et al. in [18] [30]. *BART* is a denoising autoencoder [67, p. 507] based on the Transformer architecture proposed in [11] (Section 2.2) [18]. The model is first pretrained on a 160 GB dataset [2] of English text extracted from books, Wikipedia, and news articles for 40 epochs [30]. In a second pretraining step, it is trained on 25 GB of source code from 26,000 GitHub projects for 10 epochs, before being finetuned on the *Methods2Test* dataset [30]. As the datasets used for pretraining are unlabeled, parts of the data are masked and the model is trained to predict that masked part.

In an ablation study the authors evaluated the impact the different pretraining steps have on the model's performance. Wile the model using both pretraining steps performed the best, they interestingly found that the model only pretrained on the English corpus outperformed that pretrained only on source code. With the best performing model achieving a BLEU score of 34.87 on their test data [30], their approach is the first to show any notable success at test generation using machine learning. However, they reported that the model often was having difficulties to correctly initialize the object under test, as it was lacking the context information to

[1] `https://github.com/microsoft/methods2test`

do so.

They also assessed the coverage their model achieved while generating tests for a few selected target methods and compared it to that of *EvoSuite* and GPT-3 [19] finetuned on the same task. While GPT-3 only was able to generate correct tests for about half of the selected methods, both models achieved comparable coverage to *EvoSuite* [30]. Also, in a developer survey they conducted, most of the respondents found the tests generated by *AthenaTest* to have better readability while more appropriately testing the target method, than those generated by *EvoSuite* [30].

It should be noted that although these promising results show the presented approach to have great potential, the approach and experiments presented in Chapter 3 and 4 were designed without considering these findings, as this paper was released towards the end of this thesis. Nonetheless, this thesis introduces concepts which could also be used to complement *AthenaTest*, like the use of ASTs and encoding context information in the model's input. Therefore, Chapter 5 will discuss how those approaches could be combined in future work, to counteract the shortcomings of both.

Chapter 3

Approach

As explained in Section 1.2, this research explores machine learning driven test completion. To do so, this chapter introduces a machine learning system trained to predict a test's *Then* section, given the code under test and the preceding *Given* section. This system builds on the success of the NLP approaches covered in Section 2 by employing a Transformer (Section 2.2) model.

As the Transformer is a sequence-to-sequence model (Section 2.1), the inputted and generated source code is processed as a sequence of tokens. More specifically, this work innovates on previous test generation approaches [28, 29, 30] by using ASTs to represent the source code, which are encoded as a sequences of AST nodes. Also, previous test generation approaches have only used the target method as the model's input [28, 29, 30]. However, as the target method often calls other methods, the logic which is supposed to be tested could potentially be implemented by one of the called methods which are not part of the model's input. As confirmed by recent research [30], this can make it harder for a machine learning model to predict meaningful tests, since it is missing crucial information. This thesis introduces a novel approach to adding declarations of methods to the model's input, which are called by the test and target methods. Thereby, the model is provided with additional information about other methods relevant to the test and target method, and can potentially use this information to generate more meaningful *Then* sections.

The following sections describe the individual steps of the approach employed by this thesis. First Section 3.1 explains how a dataset is created based on the source code of GitHub projects matching certain criteria, which maps tests to their target methods and segments them using the GWT pattern. How the code inputted into the model is encoded as a sequence of AST nodes and augmented with the before mentioned information about methods in its context is explained in Section 3.2. The same encoding method is applied to the previously created dataset in a preprocess-

ing step and the resulting decoded dataset is used to train the model described in Section 3.3. Once the model has been trained it can be applied to unlabeled data using the prediction pipeline covered by Section 3.4. Figure 3.1 gives an overview of these steps and how they depend on each other.

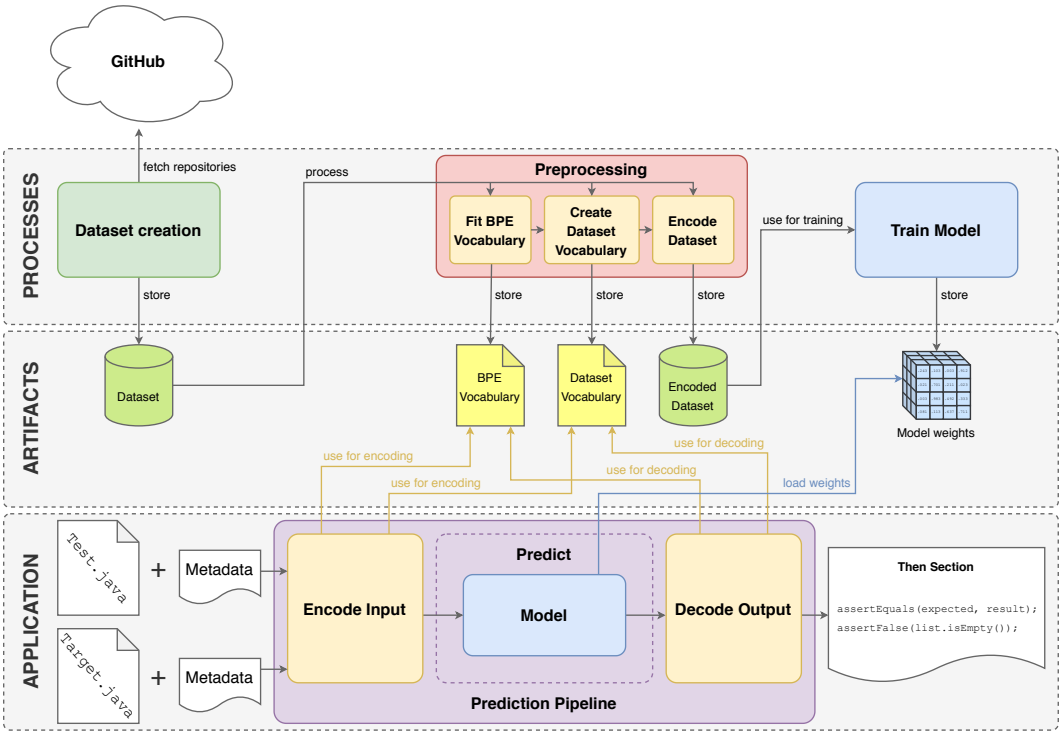


Figure 3.1: Overview of the approach illustrating the processes executed (top), artifacts they generate (middle) and how those are used to apply the model to unlabeled data (bottom). First, a dataset is created based on GitHub repositories, mapping tests to their target methods, while segmenting tests using the GWT pattern (Section 3.1). This dataset is then preprocessed, fitting a byte-pair encoding vocabulary to the value nodes (Section 3.2.4). The so-called dataset vocabulary is created, expanding the BPE vocabulary with all AST nodes (Section 3.2.5). Those vocabularies are used to encode the dataset, which is then employed to train the model, whilst storing the optimized weights. To use the model for test completion (bottom), the test and target classes `.java` files are inputted to a prediction pipeline, alongside metadata labeling the test/target methods (Section 3.4). Similarly to the data during preprocessing this input is encoded using the BPE and dataset vocabularies and the model is used with the previously optimized weights to predict the test's *Then* section. This output is decoded using the BPE and dataset vocabulary, resulting in the *Then* section's source code.

3.1 Dataset

To train a model which predicts the *Then* section for a given test a dataset is needed that contains large amount of test methods, mapped to their target methods. Also the dataset needs to provide the information, what part of each test represents the *Given*, *When* and *Then*. While [28] and [29] have each contributed a dataset for test generation, neither of them can be used for the proposed approach. Although the dataset used in [28] is publicly available, contrary to that in [29], it only contains 52,703 datapoints. Also, it is already preprocessed and therefore does not include any other information than the test and target method. Therefore, it lacks the information needed to include the declarations of methods relevant to the test and target method in the model's input.

Consequently, a new dataset is created, which is used to train the model later presented in Section 3.3 on test completion. Creating such a dataset poses multiple challenges. Firstly the source code for a sufficiently large amount of projects has to be retrieved (Section 3.1.1). Then each project has to be analyzed to infer which method is tested by which test (Section 3.1.2) and finally the *Given*, *When* and *Then* within each test have to be identified (Section 3.1.3).

Note that while [30] provides a suitable dataset which could have been extended with the information required to train a test completion model, it was published after the dataset discussed in this section had already been created, as mentioned in Section 2.4.

3.1.1 Collecting relevant open-source projects

Due to the vast amount of open-source projects available on the internet, there is no shortage of code to build a dataset on. *GitHub*[1], the worlds largest platform for open source projects, is chosen as the data source. However, the challenge is to identify suitable *GitHub* projects. As only source code which includes tests is needed, it should be avoided to download projects without tests.

Also, a few limitations will be set to make the dataset more homogeneous, like limiting it to just one programming language. The programming language chosen here is *Java*. Judging by the number of available software engineering jobs *Java* is the most used language in 2020 [55], making it a representative choice. Also, *Java*'s explicitness could be assumed to give the model more information to learn from, as it is explicitly typed and generally considered to lead to rather verbose code. Additionally, the *Java* community is remarkably consistent when

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

it comes to following project file structure conventions. This makes it easier to identify packages containing tests, as a *Java* class and the corresponding test class are usually split up into `myproject/src/`**`main`**`/org/my/project/`**`MyClass.java`** and `myproject/src/`**`test`**`/org/my/project/`**`MyClassTest.java`**.

Next, by limiting the projects to only one testing framework the test structure will stay more consistent throughout different projects. Testing frameworks is another area where the Java community is remarkable consistent, as *JUnit* has been the industry standard for many years [68] without any notable competitors.

Finally, it is common in the *GitHub* community to fork popular repositories to persist their state, without making any changes to them. Therefore, forks which are just duplicates should be excluded from the dataset. However, to avoid excluding forks containing relevant changes to the original repository, they are only excluded if they have a non-relevant amount of *Git*Hub stars.

To put these limitations into effect, all *GitHub* projects can be filtered with the following criteria to identify those projects relevant to this dataset:

- Project contains at least one `.java` file, containing the substring `org.junit`, indicating a import from the *JUnit* package
- one of the following is true:
  - Project is not a fork of another project
  - Project is a fork of another project, but has more than 20 stars

While *GitHub* offers an API for filtering projects to match these criteria, it is rate limited to 5000 requests per hour [66]. With *GitHub* hosting more than 100 million repositories [69], filtering through all projects using the API is not feasible. However, *GitHub* has released a *BigQuery* dataset which is publicly available and regularly updated [70]. This 3TB+ dataset comprises the largest released source of *GitHub* activity to date [70]. It contains a full snapshot of more than 2.8 million open source *GitHub* repositories including more than 145 million unique commits, over 2 billion different file paths, and the contents of the latest revision for 163 million files [70]. While this still doesn't allow for finding all relevant projects on *GitHub*, it provides a sufficiently large data source to build the dataset from. With *BigQuery* being able to make such large amounts of data queryable [71], relevant projects can be identified in a more feasible manner than by using the *GitHub* API.

As it is best practice in *git* to only commit stable code to the *master* branch, the dataset is created using only code from the projects *master* branches. Tests in this branch are assumed to be correct and not failing. However, it should be noted that this assumption is not validated, as doing so would require compiling and executing

every project in the dataset. This would be computationally infeasible, as it implies downloading the dependencies for all projects. Also, automatically building the projects and executing their test suite without having any control over the project structures or employed build tools is not a trivial task.

3.1.2 Mapping test to target methods

Once the foundation for the dataset has been set, as explained in Section 3.1.1, the retrieved projects can be analyzed to map test methods to their target methods.

To do so, their source code is parsed using *JavaParser*[2], a *Java* library allowing its users to analyze, transform and generate ASTs for *Java* source code. This library enables identifying test methods, which method they test and what package and class they belong to, as follows.

Identifying test methods

As mentioned in Section 3.1.1, the *Java* community mostly follows the convention of locating test classes inside packages which have a package root named `test`. Also, *JUnit* requires test methods to be label with the `@Test` annotation.

Therefore, test methods can reliably be identified by retrieving methods annotated with `@Test`, which are part of a package located under a `test` root folder.

Identifying target methods

The main indicator when identifying the target method is the name of the test. For example, the test method in Listing 3.1 executes two method calls: `increment` and `getCount`. However, the name of the test method `testIncrement` indicates that it is the test's objective to test `increment`.

```
1 @Test
2 public void testIncrement() {
3     Counter counter = new Counter();
4     counter.increment();
5     assertEquals(counter.getCount(), 1);
6 }
```

Listing 3.1: A test targeting the method `increment` from the class `Counter`

[2] `https://javaparser.org/`

But this is not always the case. The test's method name can also represent a test scenario, while the class name describes the test's target, as demonstrated in Listing 3.2.

```
1  public class IncrementTest {
2      @Test
3      public void testSingleCall() {
4          Counter counter = new Counter();
5          counter.increment();
6          assertEquals(counter.getCount(), 0);
7      }
8
9      @Test
10     public void testMultipleCalls() {
11         Counter counter = new Counter();
12         counter.increment();
13         counter.increment();
14         counter.increment();
15         assertEquals(counter.getCount(), 3);
16     }
17 }
```

Listing 3.2: A test class with each method testing different aspects of `increment` from the class `Counter`. The name of the methods describe different test scenarios, while the name of the class `IncrementTest` indicates that `increment` is the target method.

Therefore, a test's target method is identified by ranking method calls within that method by their naming similarities to the method and class name.

Tokenizing method names

Before comparing method names for similarity they are tokenized. To do so, the properties of method and class naming patterns have to be considered. While they usually comply with the camel-case notation in *Java*, other notations like snake-case are also used. Also, it is a common naming pattern to add the description of a test case to the test's name joined by an underscore. For example, the method `createAllUsers` could be tested by `createAllUsers_succesful` and `createAllUsers_serversDown`, testing the successful user creation and an edge case were servers are not available.

Therefore, method names will be split by underscores and uppercase characters which are not followed by another upper case character. Also, the resulting tokens will be converted to lower case for better comparability. Additionally, the token

`test` is dropped, as most test names contain the word `test` without it adding meaning to the name.

Doing so the names `createAllUsers`, `create_all_users`, `CREATE_ALL_USERS`, `testCreateAllUsers` and `test_createAllUsers` would all result in the tokens `create`, `all` and `users`.

Similarity score

For each method call in the analyzed test method a similarity score is calculated, which is defined as

$$sim_t(m) := \frac{|TOK(t) \cap TOK(m)|}{|TOK(t)|}$$

with t being the test method, M_t being the set of all methods called in t and $m \in M_t$. TOK returns a set of tokens by tokenizing the given method's name, as previously described. Calls to *JUnit* assertions are excluded from M_t, as they are know not to be the method under test.

Using this score M_t^* can be defined, as the set of highest ranking methods

$$M_t^* := \left\{ m \in M_t \;\middle|\; sim_t(m) = \max_{n \in M_t} sim_t(n) \;\wedge\; sim_t(m) > \delta \right\}$$

with δ setting the limit for the lowest possible similarity score, which would still be considered a match. If $|M_t^*| > 1$ no mapping will be added to the dataset for t, as the choice for the highest ranking method is ambiguous. However, if $|M_t^*| = 0$ the similarity score will be recalculated using the name of the test method's class instead. If there still is no match found, the test is considered unresolvable.

Locating the Target Method's Declaration

After the target method has been identified the source code defining it has to be located. In which class it was declared can be inferred by identifying the type of the object the target method is called on. However, resolving the type of this object is not a trivial task, as type information is not encoded in the AST. For example, when looking at Listing 3.1 it seems fairly obvious that `increment` is defined in the class `Counter`. However, it can be less obvious as illustrated by Listing 3.3. In this case `CounterFactory.createCounter()` still returns a `Counter` object, but is statically typed to return an object implementing the interface `Incrementable`, which the class `Counter` does. Within the scope of the `testIncrement` method

```
1 @Test
2 public void testIncrement() {
3     Incrementable counter = CounterFactory.createCounter();
4     counter.increment();
5     assertEquals(counter.getCount(), 1);
6 }
```

Listing 3.3: A slight deviation from Listing 3.1 makes identifying the concrete type of `counter` harder.

there is no way of inferring the type of `counter`.

However, *JavaParser* provides a feature called *TypeSolver* which is able to infer the type of a given AST node by analyzing surrounding nodes and the ASTs of imported modules. With the help of this feature the class defining the target method is located and its source code is added to the dataset. This makes the *TypeSolver* essential to the creation of this dataset and the main reason why *JavaParser* has been chosen over other AST-parsing-libraries.

Nonetheless, there are some cases where the type of an object can not be resolved through static AST analysis. For example, if `CounterFactory.createCounter()` used in Listing 3.3 is implemented as shown in Listing 3.4 its concrete return type can not be resolved statically. Since the outcome of the condition in line 3 can only be determined at runtime it is not possible to statically identify if line 4 or 6 is executed. Therefore, datapoints where the target method is not resolvable are excluded from the dataset.

```
1 class CounterFactory {
2     static Incrementable createCounter() {
3         if (Math.random() > 0.5) {
4             return new Counter();
5         } else {
6             return new NegativeCounter();
7         }
8     }
9 }
```

Listing 3.4: A possible implementation of `CounterFactory.createCounter()` used in Listing 3.3 which makes resolving the concrete return type impossible, when using static AST anaylsis.

3.1.3 Extracting *Given*/*When*/*Then*

As it is the model's objective to predict a test's *Then* section, the dataset must label the test's lines of code as *Given*, *When* or *Then*. The following section will explain the rationale behind the heuristics employed to extract this information from a given test and present the algorithm implementing these heuristics.

When looking at Listing 3.5, it seems that a test could be split up into three separate sections: *Given* ranging from line 3 to 4, *When* in line 5 and *Then* in line 6.

```
1 @Test
2 public void testAdd() {
3     List<Integer> list = new ArrayList<Integer>();
4     int expected = 1;
5     list.add(expected);
6     assertEquals(list.get(0), expected);
7 }
```

Listing 3.5: Test method targeting `ArrayList.add()`.

However, Listing 3.6 proves this to be wrong, as a *When* call (here, `list.isEmpty()`) is part of the shown test's *Then* section. Also, it can be part of the *Given* section with multiple *When* calls making it impossible to draw a clear line between *When* and *Given*/*Then*, as illustrated by Listing 3.7. This highlights that the *Given* and *Then* are two separable sections, occurring one after another without overlap, while the *When* is one or many method calls happening throughout both of these sections. In this work's dataset a section is stored as the list of lines of code defining that section. Any given line in a test has to entirely be part of either the *Given* or *Then* section.

```
1 public void testIsEmpty() {
2     List<Integer> list = new ArrayList<Integer>();
3     assertTrue(list.isEmpty());
4 }
```

Listing 3.6: Test method targeting `ArrayList.isEmpty()`. In this case the *When* call is part of the *Then* section.

These assumptions about the structure of tests can be used to identify the *Given* and *Then* section, as well as the *When* call. However, since the target method's have already been identified at this point (Section 3.1.2) and the *When* simply is the invocation of the target method, finding the *When* call is trivial. This information can be leveraged to infer the other sections, when assuming that the *Given* section is

```
1  @Test
2  public void testAdd() {
3     List<Integer> list = new ArrayList<Integer>();
4     int firstEntry = 1;
5     list.add(firstEntry);
6
7     int secondEntry = 2;
8     list.add(secondEntry);
9
10    assertEquals(list.get(0), firstEntry);
11    assertEquals(list.get(1), secondEntry);
12 }
```

Listing 3.7: Variation of Listing 3.5 with multiple *When* calls in the *Given* section. This example illustrates that *When* is not a section which can be separated from the other sections, but a part of the *Given* or *Then* section.

located before a *When* call (building up state) and the *Then* section after it (asserting state). As discussed before, this is not strictly true as multiple *When* calls can happen as part of both sections. However, by definition the transition from a *Given* to *Then* section always is marked by a *When* call, while the call can be part of either section. In Listing 3.5 the *When* call in line 5 marks the transition to the *Then* section, with it still being part of *Given*. If however, the *When* call happens inside of an assertion it is part of the *Then* section, as illustrated by Listing 3.6. In the case of multiple *When* calls, the last *When* call is considered to mark the section transition. This is shown by Listing 3.7, where both *When* calls in line 5 and 8 are considered part of *Given*, while the call in line 8 marks the transition to the *Then* section.

Looking at Listing 3.5, 3.6 and 3.7 could lead to the assumption that identifying the section transition by searching for assertion statements would be the easiest approach to identifying the start of the *Then* section. However, it is common to use assertions as part of the *Given* section to make sure that the program is in the expected beginning state. For example, in Listing 3.8 line 4 asserts that `list` is not empty before mutating it. But since this assertion does not formulate an expectation of the target method's behavior it should be considered part of *Given* not *Then*. To take this into account, assertions before the first *When* call are assigned to the *Given* section.

Also, code preceding an assert statement could be part of *Then*. In Listing 3.9 line 7 initializes the variable `firstEntry`, which is used as part of the assertion. Therefore, this variable initialization should be part of the *Then* section, thereby proving that using *When* calls as transition indicators is the more reliable approach.

Nonetheless, assertion statements are used to identify which *When* call marks the transition to *Then*, in the case of multiple *When* calls. Concluding: the last *When* call being execute before or within the first assertion statement marks the transition from the *Given* to *Then* section, with *When* being part of *Given* if happening before the assertion and part of *Then* otherwise.

```
1 @Test
2 public void testAdd() {
3     List<Integer> list = new ArrayList<Integer>();
4     assertTrue(list.isEmpty());
5     int expected = 1;
6     list.add(expected);
7     assertEquals(list.get(0), expected);
8 }
```

Listing 3.8: Variation of Listing 3.5 where an assertion statement (line 4) is used before the *When* call (line 6). Therefore, it is considered part of the *Given* section. However, the assertion statement (line 7) executed after *When* is considered part of *Then*.

```
1 @Test
2 public void testAdd() {
3     List<Integer> list = new ArrayList<Integer>();
4     assertTrue(list.isEmpty());
5     int expected = 1;
6     list.add(expected);
7     int firstEntry = list.get(0);
8     assertEquals(firstEntry, expected);
9 }
```

Listing 3.9: Variation of Listing 3.5 illustrating that code before an assertion statement (line 7) can be part of the *Then* section.

Another factor to consider while extracting the *Given* section, is that most testing frameworks support setup methods which are automatically executed before every test in a class. This allows for setting up state needed for all tests that class defines. In *JUnit* this can be done by annotating methods with `@Before` or `@BeforeEach`. When annotated with `@BeforeClass` or `@BeforeAll` a method is executed once before all the tests in the class are executed. An example of this feature is shown in Listing 3.10. Since a setup method is used to initialize state it is considered part of all *Given* sections in that class.

While extracting *Given*/*When*/*Then*, two edge cases are considered to avoid faulty

```
1  class ListTest {
2     List<Integer> list;
3
4     @BeforeEach
5     public void setup() {
6        this.list = new ArrayList<Integer>();
7     }
8
9     @Test
10    public void testAdd() {
11       int expected = 1;
12       this.list.add(expected);
13       assertEquals(this.list.get(0), expected);
14    }
15 }
```

Listing 3.10: Variation of Listing 3.5 where `list` is a property and its initialization is outsourced to `setup()`. This makes `setup()` part of the *Given* section.

datapoints. For one, all tests have to comply with the Single Assertion Principle (SAP). SAP was first introduced by Dave Astels in [72] and is widely considered a best practice for writing unit tests. As its name suggests it advocates only using a single assertion per test. However, in practice it is less about the amount of assertions than only one aspect of the target method being tested at once. Listing 3.11 shows a test not complying to this principle. Since this test has two different objectives, it is not possible to extract a distinct *Given* and *Then* section. Therefore tests violating SAP are excluded from the dataset. Listing 3.11 shows that such a violation can be detected whenever *When* (line 8) is called although *When* has been called (line 5) and asserted (line 6) already.

The second edge case is when a test is lacking assertions. The most common occurrence of this are tests for methods expected to throw an exception. As illustrated by Listing 3.12, the way these kind of tests are implemented by `JUnit` leads to no assertion statement being part of their body. Since there is no *Then* section in this case, predicting a test's *Then* section is not possible. Therefore, tests not containing any assertions are excluded from the dataset.

Using the heuristics discussed in this section Algorithm 1 is defined, which is used to identify the *Given* and *Then* sections of tests in the dataset.

```
1  @Test
2  public void testAdd() {
3      List<Integer> list = new ArrayList<Integer>();
4      int expected = 1;
5      list.add(expected);
6      assertEquals(firstEntry, expected);
7      list = new ArrayList<Integer>();
8      list.add(null);
9      assertEquals(list.size(), 1);
10 }
```

Listing 3.11: Variation of Listing 3.5 introducing a breaking compliance with SAP. This tests the regular behavior of `add`, as well as how it behaves when given `null` values. Lines 3-6 and 7-9 would need to be two separate test to comply with SAP.

```
1  @Test(expected=IndexOutOfBoundsException.class)
2  public void testGet_emptyList() {
3      List<Integer> list = new ArrayList<Integer>();
4      list.get(0);
5  }
```

Listing 3.12: Test asserting that `ArrayList.get()` throws a `IndexOutOfBounds-Exception` if called on an empty list. In *JUnit* such an assertion can be formulated by adding the parameter `expected=IndexOutOfBoundsException.class` to the `@Test` annoation. Since this leads to no assertion statement being part of the test's body, it can't be GWT resolved using Algorithm 1.

Algorithm 1: *Given/Then* section extraction

input : list of code lines in the test method → *methodBody*
output: list of code lines in the *Given* section → *given*
list of code lines in the *Then* section → *then*

```
1  given = getSetupCode()
2  then = []
3  for line in methodBody do
4      if not then.empty() then
5          if containsWhenCall([line]) then
6              throw ViolatesSAP()
7          end
8          then.append(line)
9      else if (containsWhenCall(given) or containsWhenCall([line])) and
         containsAssertion(line) then
10         then.append(getLinesSinceLastWhenCall(line))
11     else if containsWhenCall([line]) then
12         given.append(getLinesSinceLastWhenCall(line))
13     end
14 end
15 if then.empty() then
16     throw NoAssertionFound()
17 return given, then
```

3.1.4 Results

Using the approach described in Section 3.1.1 99,015 repositories were downloaded, with a total compressed size of 535GB. In these repositories 6,040,446 test methods were found in 1,127,415 test classes. Of those test methods 2,182,225 have successfully been mapped to their target method in one of 430,934 target classes, as shown in Table 3.1. On the other hand, 3,858,221 tests could not be mapped, as no target method was found using the methodology described in Section 3.1.2, multiple methods scored the same, or the matched method could not be resolved using `TypeSolver`. Of the successfully mapped tests *Given*/*When*/*Then* was extracted for 1,336,360. This failed for 845,865 tests due to them not complying with SAP, or no assert statements being found.

As mentioned in Section 3.1.3 *When* calls can happen within the *Given* section, *Then* section, or both. Therefore, datapoints are also labeled with the location of the occuring *When* calls. This way a more homogeneous subset of the dataset can be selected to train a model by excluding tests where *When* calls can occur in the *Then* section, as this would require the model to predict *When* calls.

Therefore, 1,336,360 datapoints are available for training and evaluating the model later described in Section 3.3, of which 490,951 are dropped as they contain *When* calls in the *Then* section.

Mapping test to target methods

Datapoint type	**Amount**
Tests found	6,040,446
Tests successfully mapped	2,182,225
Mapped by test method name	1,861,327
Mapped by test class name	320,898
Tests which could not be mapped	3,858,221

Extracting Given/When/Then

Datapoint type	**Amount**
Extraction succeeded	**1,336,360**
When calls only in *Given*	845,497
When calls only in *Then*	484,005
When calls in both *Given* and *Then*	6,858
Extraction failed	845,865
Tests not complying with SAP	384,679
Tests without assertions	461,186

Table 3.1: Information about the composition of the dataset created as described in Section 3.1.

3.2 Model input

As previously mentioned, the most notable novelty of the approach proposed in this thesis is it encoding the code inputted into the model as a sequence of AST nodes and adding declarations of methods called by the target and test method to it. The following sections explain how the model's input is encoded to implement these concepts.

Figure 3.2 illustrates an overview of the encoding process, with the following sections explaining the individual steps in greater detail. A central component of this process is the resolution of the methods called by the test and target methods (referred to as context methods in the following) using AST analysis, as explained in Section 3.2.1. Then the ASTs of the test and target methods, as well as those of the context methods are combined into a custom tree called *Test Declaration* AST (Section 3.2.2). This custom tree is converted into a sequence of AST tokens as explained in Section 3.2.3. Then byte-pair encoding (Section 3.2.4) is applied to the tokens in this sequence, before it is encoded using their vocabulary IDs (Section 3.2.5) resulting in a numeric sequence which is forwarded into the model.

3.2.1 Resolving Context Methods

In the following the context of a method will refer to the set of methods called by it. The contexts of the called methods are also part of this context, therefore defining it recursively. This effectively makes a method's context the code ending up in the call stack during its execution. Listing 3.13 shows an example of this, where all methods in the context of `testInc()` are highlighted.

To allow the model to understand how a test works, it is important to provide it with the test's context. For example, in Listing 3.13 there is no way of knowing how `counter` is initialized in `testInc()` without knowing how `initCounter()` is defined.

A method's context is resolved by analyzing its AST using *JavaParser*. All method calls are retrieved and *JavaParser*'s *TypeSolver* feature is used to find their declarations. This is executed recursively on the found method declarations, terminating on methods not calling others, or which have been visited before, thereby bypassing cycles in the call graph.

Due to the self-attention mechanism (Section 2.2.1), the memory used by the employed transformer architecture discussed in Section 3.3 is quadratic on the length of the input. Since a test's context potentially contains a project's entire codebase, it could drastically increase the amount of memory required for self-attention, making

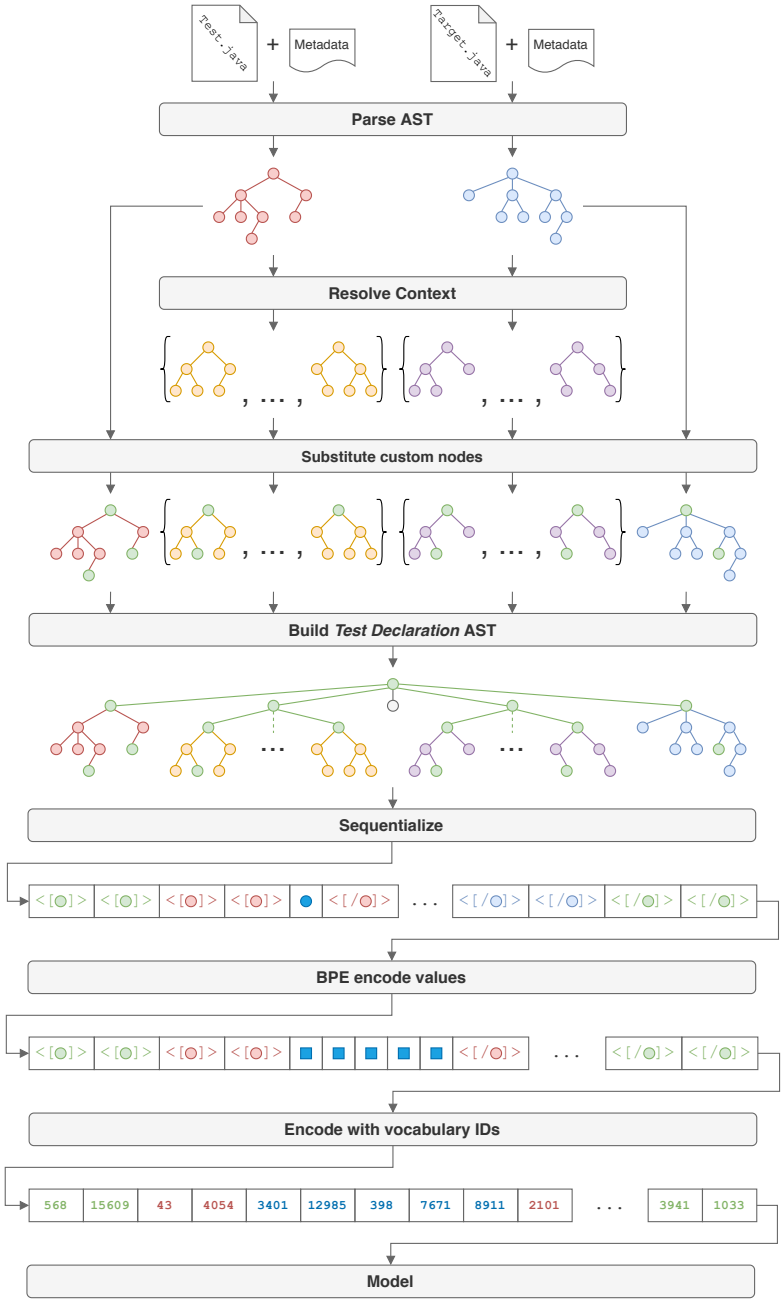


Figure 3.2: Illustration of how the model's input is encoded, with the input being the content of the test and target classes `.java` files, alongside metadata specifying which methods in those files are the test/target (top). Those methods are AST parsed and analyzed to identify methods called in their context (Section 3.2.1). The declarations of those context methods are also AST parsed resulting in a set of ASTs. Then, AST nodes which play a significant role to the test or target method are substituted with custom nodes labeling them as such (Section 3.2.2). For example, *MethodCallExpr* nodes which represent a method call to the target method are substituted with *WhenCallExpr* custom nodes. All these ASTs are combined in a custom tree called *Test Declaration* AST (Section 3.2.1), which is then converted to a sequence (Section 3.2.3). Byte-pair encoding is applied to the values in this sequence (Section 3.2.4), before it is encoded using their vocabulary IDs resulting in a numeric sequence (Section 3.2.5), which is inputted into the model.

```
1  class Counter {
2     private int count = 0;
3
4     void inc() {
5        this.setCount(this.getCount() + 1);
6     }
7
8     int getCount() {
9        return this.count;
10    }
11
12    void setCount(int c) {
13       this.count = c;
14    }
15
16    void reset() {
17       this.count = 0;
18    }
19 }
20
21 class CounterTest {
22    Counter initCounter() {
23       return new Counter();
24    }
25
26    @Test
27    public void testInc() {
28       Counter counter = this.initCounter();
29       counter.inc();
30       assertEquals(counter.getCount(), 1);
31    }
32 }
```

Listing 3.13: A simple counter class `Counter` and the test class `CounterTest` testing its `inc()` method in `testInc()`. Methods in the context of `testInc()` are highlighted.

the input infeasible to process. To limit the size of the model's input, a method's context will be limited to methods defined in the same file.

3.2.2 *Test Declaration* AST

As discussed in Section 2.4, previous work has shown better results inputting ASTs into machine learning models than plain code [36, 26, 37, 25]. This indicates that machine learning models are able to leverage on the semantic information encoded in ASTs. Thus, this work's model builds on this assessment by encoding the inputted code as an AST.

To provide the model with information the needed to predict a meaningful *Then* section, the ASTs of the test method (without the *Then* section), target method as well as the ASTs of their respective context methods are encoded in the model's input. All these ASTs are combined as subtrees in a tree referred to as *Test Declaration* AST in the following. Nodes in those ASTs which play an important role to the test are substituted with custom AST nodes labeling them according to their role. While substituting a AST node its child nodes remain the same, thereby preserving the general structure of the AST. More specifically, the following covers the information contained by the *Test Declaration* AST and discusses what the model could learn from this information:

I.1 Test name: the name of a test method encodes information about its purpose. Besides usually containing the name of the target method, it can also describe the test scenario. For example, `testAdd_nullValues()` indicates that the method's behavior when given null values is being tested.

I.2 Test body: the AST encoded body of the test method. As the *Then* section is not part of this AST, the place were it belongs in the AST is represented by a custom AST node called `ThenSection`. This acts as a placeholder indicating the location of the *Then* to the model.

Also, `MethodCallExpr` AST nodes representing a target method call are substituted with the `WhenCallExpr` custom node. Same is done for calls to context methods, which are substituted with `TestContextCallExpr`. As for all custom nodes, the child nodes of the substituted nodes remain the same. This way, information like the parameters used is preserved, whilst providing the model with labels it can learn from.

I.3 Test context: list of AST encoded method declarations in the test method's context. Analogous to the test body, calls to other methods in the test context

are substituted with `TestContextCallExpr`. As previously mentioned this context only includes methods defined in the same file as the test method.

I.4 When declaration: the AST encoded body of the target method. Similarly to the test body, calls to methods in the target method's context are substituted with `ContextCallExpr`.

I.5 When context: same as test context, but for the target method's context. Calls to methods in the context are substituted with `ContextMethodCallExpr`. This also is limited to methods defined in the same file as the target method.

As mentioned before, this information is encoded into the *Test Declaration* AST containing the relevant ASTs as subtrees. In the *Test Declaration* AST each of the before mentioned bullet points is represented by a custom node, as illustrated by Figure 3.3. By inputting this tree into the model, instead of the entire test and target classes AST, the model only receives information relevant to the test at hand, whilst keeping the size of the input to a minimum.

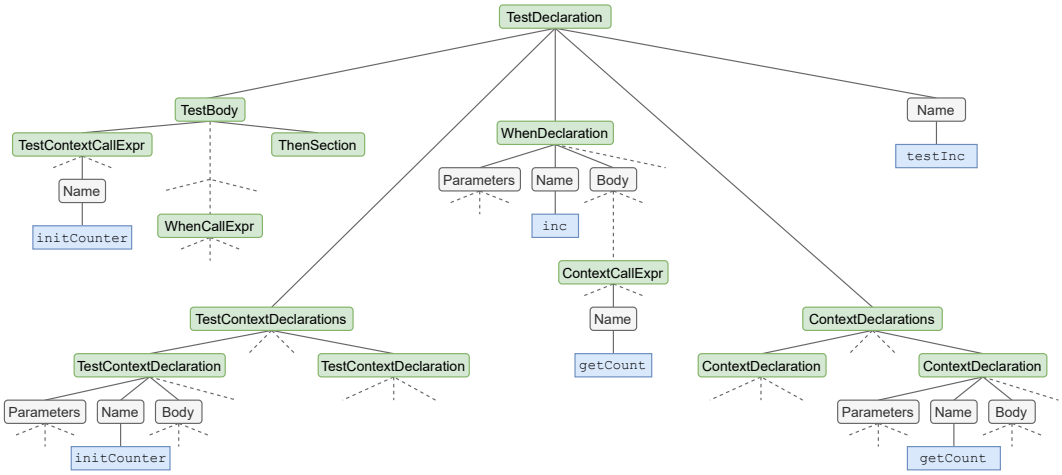


Figure 3.3: Simplified version of the *Test Declaration* AST generated for `testInc` shown in Listing 3.13. A custom *TestDeclaration* root node is created with the child nodes *Name* (I.1), *TestBody* (I.2), *TestContextDeclarations* (I.3), *WhenDeclaration* (I.4), *ContextDeclarations* (I.5). AST nodes are colored gray, value nodes blue and custom nodes which have either been added or substituted are colored green.

3.2.3 Sequentializing AST

Since the model described in Section 3.3 employs a sequence to sequence architecture, the *Test Declaration* AST has to be converted to a sequence first before being inputted into the model. This process will be called sequentialization in the following.

A tree could be sequentialized by traversing it pre-order while adding visited nodes to a sequence. However, while doing so information about parent-child relations is lost. As later discussed in Section 3.3, the model's output is encoded in the same way as its input. To make this output practical for real-world usage, it must be possible to fully decode it and generate the corresponding source code. However, decoding a sequentialized AST is not possible if it does not encode parent-child relations. To circumvent this, each node is encoded as a opening token `<[NodeType]>` and closing token `<[/NodeType]>`, with its child nodes in between. Nodes without children are represented by a single self-closing token `<[NodeType]><[/]>` to not increase the sequence size unnecessarily, while value nodes simply result in a token representing their value.

It should be considered that this increases the sequences size, as a AST with n nodes results in $2n - v - s$ AST tokens, with v being the number of value nodes and s the number of nodes without children. However, besides ensuring decodability, it is assumed that the model can also benefit from the additional information provided by the encoded parent-child relationships.

While the sequentialization process makes it possible to reconstruct the ASTs tree structure from a sequence of AST tokens, this tree still has to be converted to source code, to make the model's output useful. Once again *JavaParser* is used to solve this problem. In *JavaParser* each AST node is represented as an object. By calling a node object's `toString()` method the corresponding source code is returned. Therefore, the source code for an AST token sequence can be generated by initializing the encoded AST as *JavaParser* objects and calling `toString()` on the object representing the root node. However, additional information has to be encode in the AST sequence to make it possible to generate *JavaParser* objects from it.

By naming AST nodes like their corresponding *JavaParser* classes, the node objects can be recreated using Java's reflection feature [73], which allows for dynamically initializing objects given a string with the class and package name. However, to reconstruct the parent-child relation between two node objects, the corresponding AST token needs to encode which property of the parent's *JavaParser* object references the child node. For example, a `MethodDeclaration` node in the AST has a `SimpleName` child node, representing the method's name. Correspondingly, a `MethodDeclaration` *JavaParser* object has a `name` property referencing a `SimpleName` object. Therefore, to create those *JavaParser* objects from a AST sequence the information that the `SimpleName` child node is referenced by the property `name` must be encoded in that sequence.

To add this information to the AST, property names of child nodes are added to edges of the AST. The exception to this are value nodes and child nodes of list

types, as they aren't referred to by a property of the parent node. This is illustrated in Figure 3.4. Besides enabling AST to source code conversion, this also provides additional information the model could learn from.

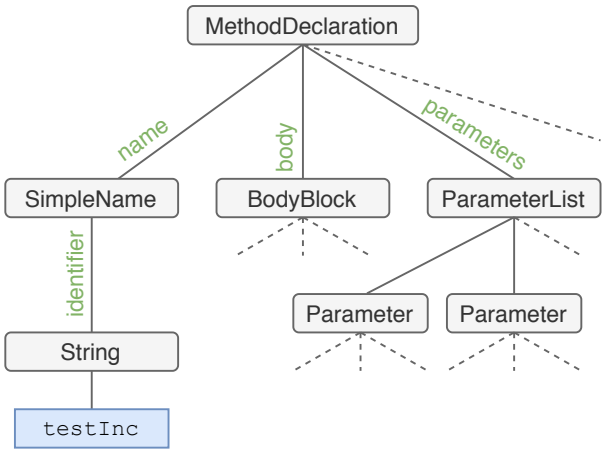


Figure 3.4: Simplified example of the AST generated for a method declaration. By adding the property names the child nodes have on *JavaParser*s `MethodDeclaration` object, it can be reconstructed from this AST. The exception to this are the `testInc` value node and `ParameterList`'s children, as they aren't referred to by a property of the parent node.

However, adding these edges as individual tokens to the sequence would grow a n sized sequence to $2n - v - l - 1$, with l being the number of list node types (like `ParameterList`). To prevent this increase in sequence size, the edge information is encoded in the nodes before sequentializing the *Test Declaration* AST. This is illustrated in Figure 3.5, which shows the same AST as Figure 3.4, but with edge information added to the nodes. Applying the sequentialization process to the tree illustrated in Figure 3.5 results in the following sequence: `<[MethodDeclaration]>`, `<[.name:SimpleName]>`, `<[.identifier:String]>`, `testInc`, `<[/.name:SimpleName]>`, `<[/.identifier:String]>`, `<[/.body:BodyBlock]>`, ... , `<[/MethodDeclaration]>`.

3.2.4 Byte-pair encoding

Contrary to the tokens in other NLP tasks, a value token (AST token representing a value node in the corresponding AST) in this sequence mostly does not represent a single word. Most value tokens contain names for entities like variables, methods, classes etc., usually concatenating multiple words in different ways. On the other hand value tokens can represent string initializers containing entire sentences. While tokenization usually is the answer to a token containing multiple words, choosing where to split a value token is not trivial. Camel-case notated entity name would

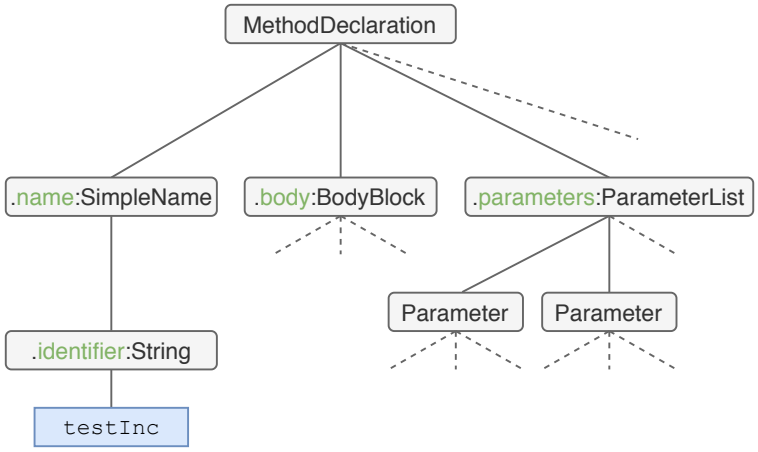


Figure 3.5: Simplified example of the AST generated for a method declaration. Illustration of the AST in Figure 3.4, but with the information from the edges encoded in the nodes. Those property names are highlighted in green.

have to be separated by upper case characters, snake case by underscores and the contents of strings by whitespaces.

Byte-pair encoding (Section 2.3) offers a flexible tokenization method, whilst also dividing tokens into pieces more comprehensible for the model. For example, the BPE vocabulary $\mathcal{V} = \{$`_get`, `Us`, `er`, `ID`$\}$ encodes the token `getUser` as [`_get`, `Us`, `er`] and `getID` as [`_get`, `ID`]. Since BPE separates both tokens by `_get` the model can leverage this information to recognize both of these methods as Getters[3]. Besides that, `getUserID` can also be encoded with $\mathcal{V}$, even though that variable name was not part of the dataset $\mathcal{V}$ was fitted to. Thus, BPE allows the model to apply its knowledge to tokens it has not seen during training.

As illustrated in Figure 3.1, the BPE vocabulary is fitted to value tokens in the dataset during preprocessing. This is done using the language-independent BPE library *SentencePiece*, introduced by Kudo et al. in [74]. The final size of the vocabulary is set to 16000. *SentencePiece* is fitted to a text corpus, by inputting a file with sentences separated by new lines. In this work's case a sentence is represented by a value token. However, while a new line splits up sentences in most NLP tasks, value tokens like string initializer can contain new line characters. To allow for encoding/decoding new line characters, they are substituted with the <_N_> token. This token is added as a special character to the BPE vocabulary, indicating that no byte-pairs should be added to the vocabulary containing this token. Meaning, that <_N_> will always be encoded as a individual token.

Since variable names often contain numbers, the characters for $n \in \{0, ..., 9\}$ are

[3] A method returning an object's private property

also added as special characters. This way the vocabulary won't be bloated by combinations of numbers, which have appeared in the dataset, whilst still allowing for encoding all numbers.

Using the fitted BPE vocabulary, the sequentialized AST resulting from Section 3.2.3 is byte-pair encoded. However, only tokens not starting with `<[` and ending with `]>` are encoded. This results in only the value tokens being encoded, as illustrated in Figure 3.6.

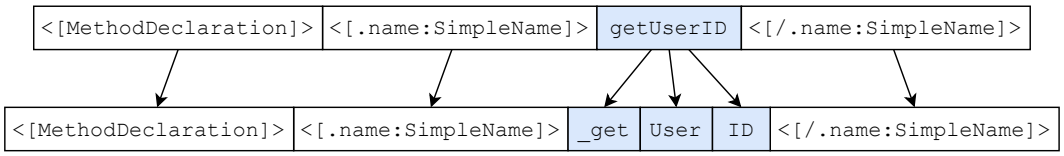


Figure 3.6: Illustration of an AST sequence (top) being byte-pair encoded (bottom). Value tokens are highlighted in blue.

3.2.5 Vocabulary ID encoding

During preprocessing all unique tokens in the byte-pair encoded dataset are stored in the so-called dataset vocabulary. This vocabulary contains all values of the BPE vocabulary in addition to all non-value AST tokens. A tokens position in this vocabulary can be used as a numeric identifier. By encoding the tokens in the byte-pair encoded sequence with their vocabulary identifiers it is represented numerically.

The size of the dataset vocabulary results in 16995, since the BPE vocabulary has a size of 16000 and 995 non-value AST tokens have been added.

3.3 Model architecture

In recent years models based on the Transformer architecture (Section 2.2) like GPT [19, 20, 21], BERT [16], and RoBERTa [17] have shown ground-breaking success in Natural Language Processing. Besides that, Transformers have also proven to outperform recurrent models in source code processing tasks such as code completion [75], code translation [22, 76], and bug fixing [77]. Therefore, this work employs a Transformer model, which is trained for test completion.

The employed architecture is equivalent to the one proposed in the original Transformer paper [11], using the same encoder-decoder structure and sinusoidal positional encoding described in Section 2.2. Also, since input and output are both equally encoded ASTs, the same embeddings can be used for the encoder and

decoder. As the Transformer outputs a probability distribution over the dataset vocabulary, the output layer size is equal to that of the dataset vocabulary: 16995.

The model is trained on the dataset presented in Section 3.1, after it has been encoded as described in Section 3.2 in the preprocessing step illustrated by Figure 3.1. Therefore, the source data is a sequence of vocabulary IDs, representing a sequentialized *Test Declaration* AST. As described in Section 3.2.2, this *Test Declaration* AST contains the ASTs of the test and target method, as well as the ASTs of their context methods. The target data is also a sequence of vocabulary IDs, which represents the sequentialized AST of the corresponding *Then* section. The start-of-sequence-token `<[THEN]>` is prepended to the target sequence and the end-of-sequence-token `<[/THEN]>` appended, indicating where a prediction starts and ends. Since different datapoints in a batch are unlikely to have the same size, they are padded to the same size using the `<[PAD]>` token during training.

3.4 Prediction pipeline

As already previewed in Figure 3.1, a prediction pipeline is built, to apply the trained model to unlabeled data. This section will explain how this pipeline works, while Figure 3.7 illustrates the process.

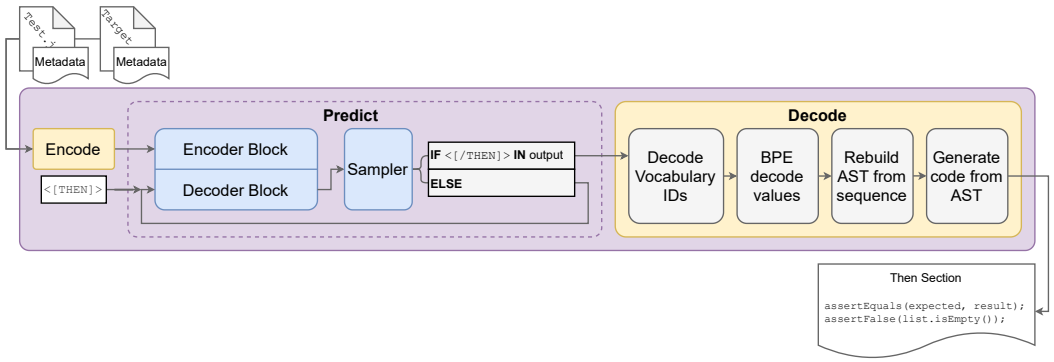


Figure 3.7: More detailed illustration of the prediction pipeline shown in Figure 3.1. The input is encoded as a numeric sequence (Section 3.2), which is then forwarded to the Transformer's encoder block. A start-of-sentence-token `<[THEN]>` is inputted to the decoder block, which outputs a probability distribution for each inputted token. The next token is sampled from the last probability distribution outputted by the decoder block. This token is appended to the input sequence of the decoder block and the prediction process is repeated until the resulting sequence contains the end-of-sequence-token `<[/THEN]>`. Once it does, the resulting sequence is decoded and the corresponding source code is generated.

The prediction pipeline's input is the content of the test and target classes `.java`

files, alongside metadata specifying which methods in those files are the test/target methods a *Then* section should be predicted for. As explained in Section 3.2, this input is represented as a sequentialized *Test Declaration* AST (Figure 3.2), which is encoded using BPE (Section 3.2.4) and the vocabulary IDs of the tokens (Section 3.2.5). This numeric sequence is entered into the Transformers encoder block. The start-of-sentence-token `<[THEN]>` is generated and inputted to the decoder block. Then the prediction is sampled from the probability distribution outputted by the decoder block. The different sampling methods used are discussed in the following section. Thereafter, the predicted token is appended to the decoder's input sequence and the process is repeated, until the prediction ends with the end-of-sentence-token `<[/THEN]>`.

To prevent the prediction pipeline from consuming too much computing resources, by endlessly generating tokens, a maximum sequence length is defined. If the predicted sequence reaches this length before a `<[/THEN]>` token is generated, the prediction process is terminated with an error. In this work the maximum sequence length is set to 512.

Once `<[/THEN]>` has been generated the models output is decoded. Analogous to the encoded input, tokens in the predicted sequence represent vocabulary IDs. After decoding them, value tokens are decoded using the byte-pair vocabulary. As covered by the Sections 3.2.2 and 3.2.3, the employed encoding allows for building an AST from the predicted sequence, which can be used to generate the corresponding *Java* source code. Since this is done by constructing *JavaParser* objects the prediction process will be terminated, if the model failed to generate a valid AST.

3.4.1 Sampling methods

As described in Section 2.2, the Transformer outputs the probability distribution $P(y_{t'}|y_{1:t'-1}, x)$ over the next token $y_{t'}$ in the vocabulary V, given the predictions for the previous $t'-1$ tokens $y_{1:t'-1} = (y_1, \ldots, y_{t'-1})$ and the model's input sequence x. Depending on the method employed by the prediction pipeline for sampling a token from this distribution, the resulting predictions can vary a lot. Therefore, different sampling methods are implemented by the prediction pipeline. How these sampling methods perform compared to each other is later evaluated by the experiments conducted in Section 4.3 and 4.4.

Greedy

The most obvious approach to sampling is the greedy method, which simply selects the token with the highest probability.

$$S_{greedy} \coloneqq \arg\max_{y \in V} P(y \mid y_{1:t'-1}, x)$$

Nucleus

The *Nucleus* sampler was introduced by Holtzman et al. in [78]. It filters the probability distribution by selecting the highest probability tokens whose cumulative probability mass just barely exceeds the pre-chosen threshold p [78]. To do so, the top-p vocabulary $V^{(p)} \subset V$ is defined as the smallest set such that

$$\sum_{y \in V^{(p)}} P(y \mid y_{1:t'-1}, x) \quad \geq \quad p$$

holds true [78]. The original distribution is then filtered by $V^{(p)}$ and re-scaled using $p' = \sum_{y \in V^{(p)}} P(y \mid y_{1:t'-1}, x)$ resulting in

$$P'(y \mid y_{1:t'-1}, x) \coloneqq \begin{cases} P(y \mid y_{1:t'-1}, x)/p', & \text{if } y \in V^{(p)} \\ 0, & \text{otherwise.} \end{cases}$$

From this multinomial distribution the final token is sampled:

$$S_{nucleus} \coloneqq sample\_multinomial(P'(y \mid y_{1:t'-1}, x)).$$

Based on the shape of $P(y_{t'}|y_{1:t'-1}, x)$ and the value of p, the size of the subset $V^{(p)}$ adjusts at each prediction step [78]. While a distribution with high peaks will lead to a small $V^{(p)}$, a flat distribution results in *Nucleus* sampling from a set of many moderately probable tokens [78].

In the following experiments $p = 0.95$ is used, as suggested in [78].

Copying

A distinct property of predicting *Then* sections is that variable and method names in the *Given* section are very likely to appear in the *Then* section. For example, if the variable `result` is initialized in the *Given* section, it is likely to be used in the corresponding *Then* section. In an attempt to leverage this property, a pre-sampling method is introduced masking probabilities of value tokens not appearing in the input sequence. One of the previously mentioned sampling methods is then used to sample from this distribution. This concept is called *copying* mechanism and has

proven successful in other NLP tasks [79]. Also, in [23] Uri et al. applied a *copying* mechanism to their model trained at code completion. By doing so, they were able to improve the quality of their model's predictions by almost 10% [23]. These results are promising, when considering test completion to be a more specific kind of code completion.

To only apply the *copying* mechanism to entity names (variables, methods, classes etc.), the pre-sampler S_{copy} is only executed when the model is predicting a value token at the current prediction step. Since AST nodes representing entity names are of the type `<[.identifier:String]>`, the prediction pipeline can identify if the next predicted token is part of an entity name and therefore apply S_{copy} to the model's output. For example, if preceding prediction steps have resulted in the AST sequence

..., `<[.name:SimpleName]>`, `<[.identifier:String]>`, `_get`

the next prediction step will either produce another value token, or close the identifier node by generating `<[/.identifier:String]>` (assuming the model generates a valid AST).

Contrary to the previously explained sampling methods, the *copying* pre-sampler S_{copy} is not applied to the probability distribution outputted by the Transformer's $softmax$ layer, but the logits l resulting from the preceding linear layer (Figure 2.5). Therefore, S_{copy} is defined as

$$S^*_{copy} := \begin{bmatrix} l_1, & \text{if} & 1 \in K & \wedge & softmax(l)_1 > \delta & \text{else} & -\infty \\ & & & \vdots & & & \\ l_{|V|}, & \text{if} & |V| \in K & \wedge & softmax(l)_{|V|} > \delta & \text{else} & -\infty \end{bmatrix}$$

$$S_{copy} := \begin{cases} S^*_{copy}, & \text{if } \sum_{x \in S^*_{copy}} x \neq 0 \\ l, & \text{otherwise} \end{cases}$$

with $K \subset V$ containing the value tokens from the model's input as well as `<[/.identifier:String]>`. The closing token is included in K to prevent the sampler from generating entity names longer than they should be. The threshold δ avoids the sampling of tokens with very low probabilities, in case none of the high probability tokens are in K. If no token from the input sequence has a probability larger than δ, l is not masked. This also enables the sampling of new variable names if the model is very confident about them. In this work $\delta = 0.001$ is used.

If the notation $S_{copy}|S$ is used in the following, the prediction pipeline filters the model's logits with the pre-sampler S_{copy} before they enter the $softmax$ layer, from

which the final output is sampled using the sampler S. As S_{copy} masks l with $-\infty$, the filtered probabilities resulting from the $softmax$ layer are set to 0.

3.4.2 Implementation

As mentioned in Section 3.2, the encoding relies on the *TypeSolver* feature unique to *JavaParser*, to resolve the test method's context and create the *Test Declaration* AST. Also *JavaParser* is required to generate the corresponding source code for the AST sequence predicted by the model. However, the model is implemented using the *PyTorch*[4] wrapper *PyTorch Lightning*[5]. Since this is a *Python* library, the overall prediction pipeline is implemented as a *Java* and *Python* application communicating with each other.

To achieve this, the functionality unique to *JavaParser* is implemented in a *Java* application, providing it through a simple TCP API. The main part of the prediction pipeline is implemented in *Python*, sending TCP requests to the *Java* server where the *JavaParser* functionality is required. This process is illustrated in Figure 3.8.

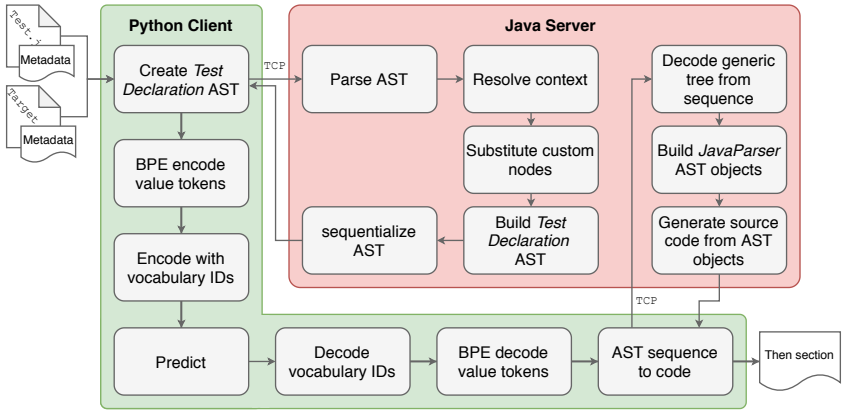


Figure 3.8: Overview of how the prediction pipeline is implemented. The *Test Declaration* AST can only be build by a *Java* application, as it relies on *JavaParser*s functionality. Therefore, the *Python* application implementing the prediction pipeline sends the input data to a *Java* server, creating the *Test Declaration* AST. The following encoding and prediction steps are handled by the *Python* client. While it also decodes the predicted sequence, it requires the *Java* server to generate source code for the AST sequence.

[4] https://github.com/pytorch/pytorch
[5] https://github.com/PyTorchLightning/pytorch-lightning

Chapter 4

Experiments and results

This Chapter covers different experiments conducted on the dataset and model presented by Chapter 3. Each experiment explores a different aspect of the proposed approach, discussing how their results can be interpreted to answer the research questions posed in Section 1.2.

After Section 4.1 has covered the setup used for the experiments, Section 4.2 tries to answer RQ1 by assessing the quality of the dataset. In Section 4.3 RQ2 is explored by examining the parsability of the model's output. The model's ability to generated *Then* sections is evaluated quantitatively in Section 4.4 and qualitatively in Section 4.5, thereby investigating RQ3. Afterwards, Section 4.6 assesses if the model benefits from its input being AST encoded (RQ4) and Section 4.7 analyzes if the model is able to understand the context information added to that input, as questioned by RQ5. Finally, a possible threat to the validity of those experiments is explored in Section 4.8.

4.1 Setup

There are a lot of variables while executing the experiments described in Chapter 4. To reduce variability and allow for reproducibility and comparability, this section describes the setup used for the following experiments.

First, the model (Section 3.3) is trained using the data described in Section 4.1.2 with the hyperparameters in Section 4.1.3. The trained model is then used in conjunction with the prediction pipeline (Section 3.4) to create predictions which are evaluated as explained by Section 4.1.4. The hardware used during execution is presented by Section 4.1.1.

Unless specifically stated otherwise all experiments follow this setup and evaluation

process.

4.1.1 Hardware

All experiments are executed on a server using an *Intel i9-10900X* CPU, 128 GB of RAM and four *NVIDIA GeForce 2080 Ti* GPUs with 11 GB video memory each. As mentioned in Section 3.4.2, the employed model (Section 3.3) is implemented using *PyTorch* which allows for accelerating the training process by utilizing the GPUs with CUDA [80]. While this significantly reduces the training duration, data has to be loaded into video memory instead of RAM, which is a more limited resource, therefore limiting the size of the processed sequences and batches.

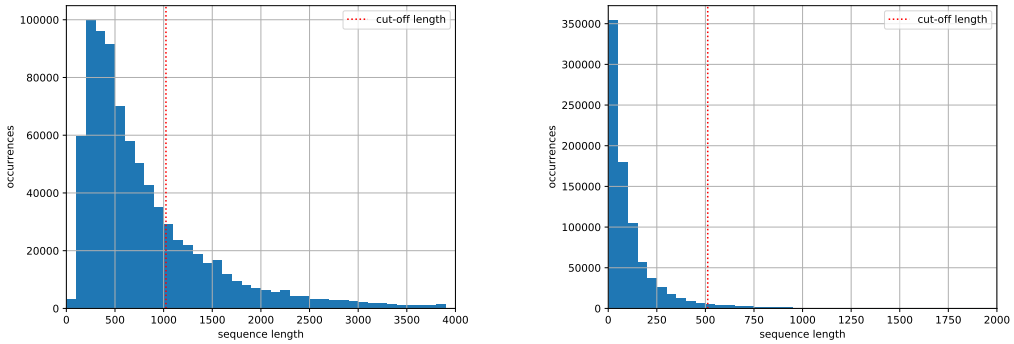
4.1.2 Data

The dataset created in Section 3.1 is used to train and evaluate the model (Section 3.3). Each datapoint in this dataset consists of the source data inputted into the model (sequentialized *Test Declaration* AST) and target data which the model is expected to predict (sequentialized AST of the *Then* section).

As mentioned in Section 3.1.4 the dataset is labeled with the location of occurring *When* calls. Since the model is trained to predict *Then* sections it also would have to predict *When* calls, if the dataset included datapoints where *When* calls happen within the *Then* section. To avoid this additional challenge, datapoints are excluded from the experiments if their *Then* section contains *When* calls. This eliminates 490,951 datapoints reducing the effective dataset size to 845,409.

Another factor leading to a reduction of the dataset is the size of the datapoints. As explained in Section 2.2 the memory used by the Transformer architecture is quadratic on the length of the input. Since the model is trained using GPUs (Section 4.1.1) and video memory is a limited resource the dataset has to be restricted to datapoints of a processable size. As illustrated by Figure 4.1a, limiting the size of the source sequences to 1,024 tokens reduces the video memory required during training, while eliminating 231,059 datapoints. With the targeted *Then* sections being considerably shorter than the inputted *Test Declaration* ASTs, the maximum size for target sequences is only set to 512. This results in 36,785 datapoints being dropped, as shown in Figure 4.1b. However, since there is an overlap between datapoints with oversized source and target sequences, only 250,038 datapoints are eliminated in total, leaving 595,371 datapoints.

As is common practice in machine learning, the dataset is split into a training, validation and test dataset, with the training split containing 80% of the datapoints



(a) source sequence lengths

(b) target sequence lengths

Figure 4.1: Sequence length histograms of all datapoints without *When* calls in their *Then* section. Also, the chosen maximum length is marked. Sequences longer than this limit are dropped.

and the other splits 10% each. After excluding the before mentioned datapoints, the splits result in the sizes listed in Table 4.1.

Split	Share	Datapoints
Training	80%	476297
Validation	10%	59537
Test	10%	59537

Table 4.1: Sizes of the employed data splits.

While creating the splits the assigned datapoints are randomly sampled from the dataset. This could be considered a naive approach, as this allows the training and test split to contain tests from the same project. Therefore, the model could potentially perform well on the test data by copying information from the training data if there is a high similarity between different tests in a project. However, this is a deliberate decision made due to the complexity of the tackled learning problem and the lack of previous research. By splitting the data less strictly the potential of the proposed approach (Chapter 3) can be explored in a somewhat simplified setting. Later Section 4.8 will present an experiment aimed at investigating the effect this naive split has on the model's performance.

4.1.3 Hyperparameters

This section explains how the model's hyperparameters have been chosen, while discussing the most important hyperparameters in greater detail.

In theory, the most optimized model can only be found by training the model

with each possible hyperparameter combination and picking that combination performing the best. However, this is not feasible in practice due to computational limitations. Therefore, most hyperparameters are picked as suggested by the original Transformer paper [11], while those assumed to have the biggest impact on the model's performance are optimized further. To do so, the hyperparameter optimization framework *Optuna*[1] is used. *Optuna* is able to minimize/maximize a function by finding the most optimal hyperparameters in a given set or interval of possible values [81]. Hyperparameters are sampled from the set of possible values using the Tree-structured Parzen Estimator presented in [82]. *Optuna* also handles running multiple function executions in parallel, while making sure resources are used effectively by pruning unpromising runs [81]. In the case of this work, a function training the model returning its validation loss is minimized. While the best results would most likely be yielded by minimizing the metrics the model is evaluated on, this would not be feasible, since running the evaluation process is too computationally expensive as Section 4.1.4 later explains. However, minimizing the validation loss is considered a sufficient substitute, which is computationally more efficient.

While *Optuna* allows for optimizing hyperparameters comparatively effectively, the optimization still is a computationally expensive task. Therefore, the hyperparameters and their possible values chosen for optimization have to be selected cautiously. The following will discuss the most relevant hyperparameters, explaining how their values have been selected or why they were chosen for optimization with `Optuna`.

Learning rate

The **learning rate** is often considered the most important hyperparameter when tuning machine learning models [67, p. 429] [83]. Therefore, optimizing the learning rate is an obvious choice. The set of possible values is defined as $\{0.01, 0.005, 0.001, 0.0001, 0.00001\}$. As the learning rate will be further modified in the following, this value will be referred to as *base_lr*.

Also, a **learning rate scheduler** is employed adjusting the learning rate during training. This can help in achieving faster convergence, while preventing oscillations and getting stuck in local minima [84]. Similarly to the original transformer paper [11], the employed schedule decays the learning rate by multiplying the inverse square root of the optimization step number *step* with a factor d.

$$lr\_schedule_d(step) := d\frac{1}{\sqrt{step}}$$

[1] `https://optuna.org/`

Additionally, a warmup phase is added linearly increasing the learning rate from 0 to $base\_lr$ in $warmup\_steps$ before using $lr\_schedule_d$ with $d = base\_lr\sqrt{warmup\_steps}$.

$$lr(step) := \begin{cases} step\dfrac{base\_lr}{warmup\_steps}\,, & \text{if } step < warmup\_steps \\ lr\_schedule_{base\_lr\sqrt{warmup\_steps}}(step)\,, & \text{otherwise} \end{cases}$$

This results in the learning rate evolving in relation to the number of optimization steps as illustrated in Figure 4.2.

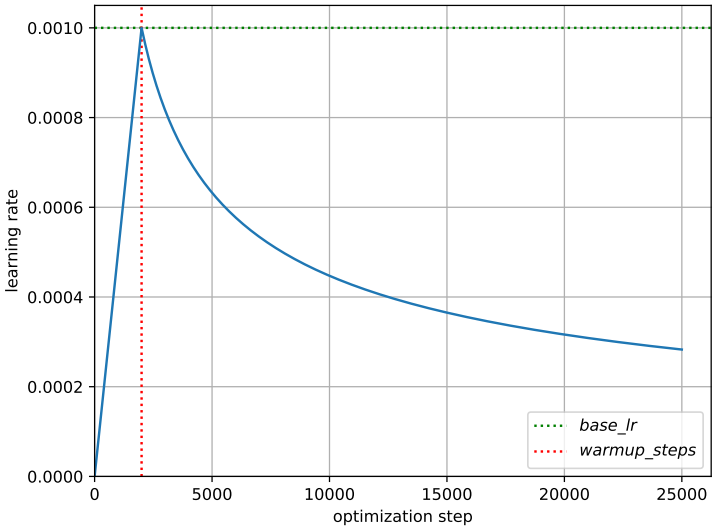


Figure 4.2: Plot of the learning rate evolving with the number of optimization steps. The $base\_lr$ is set to 0.001 and $warmup\_steps$ to 2000.

The number of **warmup steps** is another hyperparameter optimized with the set of possible values being $\{500, 1000, 2000, 4000\}$.

Batching

The **batch size** controls the number of datapoints used per optimization step. This can have an impact on the model's ability to generalize. Also, a large batch size can decrease the time needed for training, as the transformer architecture allows for processing multiple datapoints in parallel. However, this also increases the size of the model's input, thereby increasing the amount of memory required. Therefore, the batch size can't be set higher than 4, due to the video memory limitations of the used hardware mentioned in Section 4.1.1. While a higher batch size would be achievable by decreasing the maximum length of the source and target sequences

(Section 4.1.2), this would imply decreasing the dataset size even further. To avoid this, gradient accumulation is employed. This technique allows for simulating large batch sizes without requiring the same amount of memory. This is achieved by accumulating the gradients of N batches, only updating the model's weights after a given amount of steps, instead of each step. The **number of accumulated gradients** N is the hyperparameter which controls the effective batch size:

$$effective\_batch\_size \;:=\; batch\_size * N$$

Therefore, tuning the number of accumulated gradients allows for testing the effects of different batch sizes, without being limited by video memory. The number of accumulated gradients N is optimized from the set of possible values $\{16, 32, 64, 128\}$. As the batch size is set to 4 this results in the corresponding effective batch sizes being $\{64, 128, 256, 512\}$.

Loss function

As described by Section 3.3, the model outputs a probability distribution over the vocabulary. Therefore, the cross entropy loss is employed, which is the most common choice for a **loss function** evaluating probability distributions.

Applying label smoothing [85] to this loss was also explored, as [11] reported that doing so can improve the Transformer's performance. However, comparing the results the model trained with and without label smoothing achieved during the evaluation process later described in Section 4.1.4 showed that label smoothing actually decreased performance. The detailed evaluation results of this comparison are listed in Appendix A.

Optimization

During training, the model's parameters are updated based on the before mentioned loss using *AdamW* [86]. This optimization method improves upon the popular **optimizer** *Adam* [50] by decoupling the weight decay from the gradient update, which has shown to yield better generalization [86]. The hyperparameters for *AdamW* are set to $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

Transformer hyperparameters

The hyperparameters introduced by the employed transformer architecture are largely chosen as suggested by [11]. The **embedding size** is set to 512, **number of**

attention heads to 8, **number of encoder/decoder layers** to 6 and the **dimensions of the feed-forward layers** inside the encoder/decoder are set to 2048. However, the **encoder/decoder dropout** as well as the **positional encoding dropout** are further optimized. They are tuned on the interval $[0.1, 0.5]$.

Optimization results

The hyperparameter optimization conducted with *Optuna* explored 86 different hyperparameter combinations, each trained for up to 10 epochs unless pruned prematurely, which resulted in a runtime of about 16 days. The results of this optimization process are presented in Table 4.2 along with the preselected hyperparameter values.

Hyperparameter	Optimized on	Value
Learning rate	$\{0.01, 0.005, 0.001, 0.0001, 0.00001\}$	0.001
Learning rate scheduler		inverse square root
Warmup steps	$\{500, 1000, 2000, 4000\}$	1000
Batch size		4
No. accumulated gradients	$\{16, 32, 64, 128\}$	32
Loss function		Cross entropy
Optimizer		AdamW
AdamW β_1		0.9
AdamW β_2		0.999
AdamW ϵ		10^{-8}
Embedding size		512
No. attention heads		8
No. encoder layers		6
No. decoder layers		6
Feed-forward dimension size		2048
Encoder/Decoder dropout	$[0.1, 0.5]$	0.1954
Positional encoding dropout	$[0.1, 0.5]$	0.2185

Table 4.2: Overview of the hyperparameters used for the experiments in Chapter 4. Also, the sets/ranges are displayed in which *Optuna* searched for the optimal values.

4.1.4 Evaluation

To evaluate the performance of a model, it is trained until the validation loss has stopped improving for 3 epochs. This optimum was usually achieved by the model after 25 to 35 epochs, which took 2 to 3 days using the hardware described in Section 4.1.1. The model is then evaluated on the test split, using the weights resulting from the training epoch with the lowest validation loss. The following will introduce the used evaluation process, utilizing the prediction pipeline (Section 3.4)

and a set of metrics to assess the generated predictions and compare them to the corresponding target data in the test split.

Process

As explained in Section 2.2.5, the process of training transformers is sped up by inputting the entire target sequence into the decoder block, while using a mask in the self-attention layer to prevent tokens from attending to subsequent tokens [11] (*teacher forcing*). However, this is not possible when applying the model to unlabeled data, since the target output is not known ahead of time. Although the test split provides target data, using the prediction pipeline allows for evaluating how the model performs on unlabeled data, as it does not utilize *teacher forcing*. Instead the prediction pipeline repeatedly feds the model its own output until a end-of-sequence token is generated, as explained in Section 3.4. Therefore, generating a output of the length n requires executing $n + 1$ predictions compared to 1 prediction when using *teacher forcing*.

Another property of the prediction pipeline used to validate the model's practicability, is that it fully decodes the model's predictions and tries to generate the corresponding source code. Thereby it implicitly tests whether the predicted AST sequence represents parsable code, which is not considered during training.

However, as explained in Section 3.4.2, fully decoding the model's prediction involves calling a Java server to generate the outputted source code. The overhead of communicating with the Java server combined with the before mentioned n steps required to generate the prediction, makes the evaluation process too computationally expensive to be executed during training. To sped up this process, only 10,000 datapoints are used, which have been randomly selected from test split. Nonetheless, executing the evaluation process takes approximately 20 hours. This can vary depending on the evaluated model, as models tending to generate large sequences also take longer to evaluate. To allow for comparability of different results, all experiments use the same 10,000 datapoints from the test split.

Another reason for using the prediction pipeline during evaluation is that the outputted source code makes its predictions comparable to those of other models which do not employ an AST encoding. If a experiment requires comparing multiple variants of the model, each variant is individually trained and predictions are generated for the before mentioned 10,000 datapoints from the test split, using the previously described process. The resulting predictions are then evaluated by the metrics described in the following section. It is important to note that metrics which assess the quality of the predicted source code, like the later explained BLEU and ROUGE scores, can only be measured if the generated prediction represents

parsable code. Therefore, when comparing different models, only those datapoints are considered in the evaluation which all models were able to generate parsable code for, to ensure that their quality is measured using the same data.

Metrics

To quantify the model's performance and thereby enable comparability between different model versions, as well as future research, the following metrics are used.

Unparsable rate: measures the percentage of predictions resulting in unparsable code. As explained in Section 3.4, the prediction pipeline will raise an error if a prediction generated by the model is not parsable. By catching and counting those errors the unparsable rate is calculated.

When assessing the unparsable rate it should be considered that parsable code isn't necessarily compilable. However, compiling the predictions is not feasible, as it requires building the projects the predictions were created for, to make sure all imports can be resolved. This would immensely increase the size of the test split as it implies downloading the dependencies for all projects. Additionally, automatically building a project without having any control over the project's structure or build tools is not a trivial task. Nonetheless, the unparsable rate is a good indicator of whether a model is able to understand the syntactic structure of source code.

Max length exceeded rate: as explained in Section 3.4, the prediction pipeline stops the prediction process when the output's length exceeds 512 without generating a end-of-sentence token. While this does not necessarily mean that the model's prediction is entirely wrong, it indicates that the model failed to predict a end-of-sentence token when it should have, as the target predictions have been limited to 512 (Section 4.1.2). Similarly to the unparsable rate, this metric is measured by observing the percentage of datapoints for which the prediction process is interrupted because of exceeding the maximum length.

Unknown token generated rate: as the unknown token can't be decoded generating it results in unparsable code. Therefore, a model with a tendency to generate unknown tokens could skew the unparsable rate. To avoid this, the percentage of predictions containing unknown tokens is measured separately.

BLEU: the BLEU score was introduced by Papineni et al. in [87], providing a metric for comparing machine generated sequences with one or many target sequences. It has gained popularity evaluating various machine learning systems mainly in the field of Natural Language Processing. While initially developed for assessing Neural Machine Translation models [87], it has proven applicable to other NLP tasks like text summarization [88] and speech recognition [89].

The main reason for BLEU's success in the field of NLP is its ability to allow predicted sequences to score well, even if their order does not exactly match that of the target sequence. This is a common challenge in evaluating machine generated text, due to the ambiguity of language and its nature of formulating the same meaning in different ways. Since evaluating generated source code poses similar challenges, BLEU provides a promising metric.

The cornerstone of BLEU is a modified n-gram precision metric, measuring the n-gram overlap between a predicted and target sequence [87]. Contrary to the familiar precision measure, this modified version considers a n-gram exhausted after a matching candidate is identified [87]. The BLEU score is then calculated as the weighted geometric mean of the 1-gram, 2-gram, 3-gram and 4-gram precision, using the weights $w = (0.25, 0.25, 0.25, 0.25)$. Also, since BLEU is a precision based metric it is multiplied by a brevity penalty [87], to prevent short predictions from scoring too high. When applied to multiple datapoints the BLEU scores for the individual datapoints are micro-averaged.

When used in an NLP context, one of BLEU's disadvantages is that it evaluates the prediction's adequacy but not fluency of the language used. However, in this thesis the predictions "fluency" is already addressed by the unparsable score, evaluating its syntactic correctness.

To allow for better interpretability of the BLEU score, BLEU_n for $n \in \{1, 2, 3, 4\}$ is also measured. BLEU_n is defined as the BLEU score using the weights W_n with $W = ((1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1))$. This reveals the BLEU score's composition, uncovering if the tested system only performs well at matching n-grams of a specific size.

As explained in Section 4.1.4, the BLEU score is calculated on the source code outputted by the prediction pipeline. Since BLEU only works on sequences, the prediction's as well as target's source code have to be tokenized first. This is done using the *Python* library *javalang*[2] which allows for tokenizing *Java* code. An example of how the BLEU score is calculated for a given datapoint and how the n-gram precision works is illustrated in Figure 4.3.

Since BLEU scores range from 0 to 1, it is common practice to scale them by 100, to better highlight differences between evaluation results.

ROUGE: another metric for evaluating the quality of machine generated sequences is ROUGE, which was presented by Lin in [90]. Similarly to BLEU, ROUGE-n measures the n-gram overlap between a predicted and target sequence. However, it does so using the n-gram recall instead of precision. Meaning, ROUGE evaluates

[2] `https://github.com/c2nes/javalang`

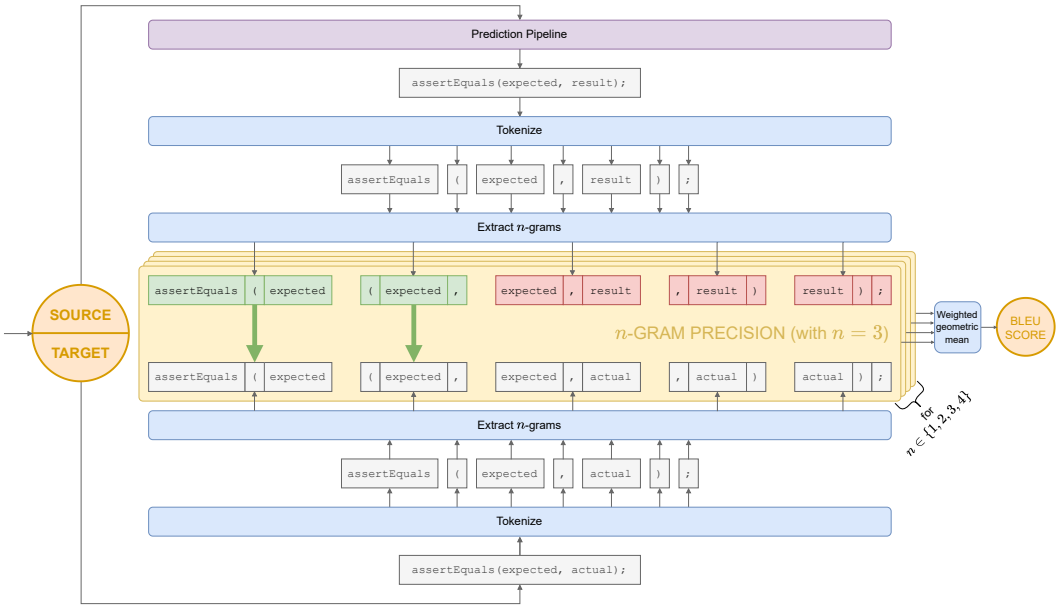


Figure 4.3: Illustration of how the BLEU score is calculated for a given datapoint (left). First, the source data is passed into the prediction pipeline (top). The predicted *Then* section (top) and target prediction (bottom) are then tokenized using *javalang* and their n-grams are extracted. Using these, the n-gram precision measures how many of the prediction's n-grams overlap with the n-grams of the target, in relation to the total number of n-grams in the prediction (middle). The prediction's n-grams which also occur in the set of target n-grams are labeled green, and red otherwise. By taking the weighted geometric mean of the n-gram precision for all $n \in \{1, 2, 3, 4\}$ the BLEU score is calculated (right). Note that this illustration leaves out details like the brevity penalty [87].

how well the target matches the prediction, while BLEU is measured the other way around [90]. Therefore, ROUGE is commonly used complementary to BLEU.

While ROUGE-n does not combine weighted scores it measured using different values for n, like BLEU does, there are different variants of ROUGE-n. Next to ROUGE-1 and ROUGE-2 this thesis uses ROUGE-LCS, where LCS stands for *longest common subsequence*. ROUGE-LCS is equivalently measured to ROUGE-n, however, n is dynamically set to the the size of the longest n-gram found in both the prediction and target sequence [90].

Similarly to how precision can't reveal a prediction as being to short, a perfect recall score can be achieved even if the prediction is too long. While BLEU uses the brevity penalty to counteract this shortcoming of the precision measure, ROUGE doesn't implement such a mechanism. Instead ROUGE-n is usually complemented with the metrics ROUGE-n_P and ROUGE-n_F, representing the n-gram precision and F-score respectively [90]. These can be used to put ROUGE-n scores into perspective, thereby uncovering irregularities in the evaluated system otherwise left unnoticed.

Analogous to BLEU, the ROUGE metrics are calculated on the code outputted by the prediction pipeline, after it has been tokenized using *javalang*. Also, all ROUGE scores are scaled by 100, to be in the same range as the BLEU scores. However, when comparing ROUGE and BLEU scores it must be considered that BLEU employs a micro-average on the achieved scores when applied to multiple datapoints, while ROUGE macro-averages them. As a micro-average weights how the tested system performs on long sequences more heavily, the resulting scores can represent a system's performance differently.

Adjusted BLEU/ROUGE: since only parsable predictions can be evaluated using BLUE and ROUGE, a model performing well with one parsable prediction while only generating unparsable code otherwise, could score high. This can lead to misleading results if those scores are assessed out of context, making BLEU/ROUGE less suitable for comparing the performance of different models.

To put BLEU and ROUGE more into context adjusted versions of those metrics are introduced. The adjusted score $adj(m)$ of the metric m is defined as

$$adj(m) := \frac{100 - unparsable\_rate - unkown\_token\_rate}{100} \; m$$

scaling the metric by the ratio of parsable code. As mentioned before, exceeding the maximum length does not necessarily mean that a prediction is unparsable. Therefore, the max length exceeded rate is not considered when calculating the ratio of parsable code.

By scaling BLEU/ROUGE like this, a similar effect is achieved to having unparsable predictions score 0. This way models are penalized if they are not able to consistently generate parsable code.

Manual evaluation

While metrics like BLEU and ROUGE can indicate if a model is doing better or worse than others, they do not point out what specifically it is good or bad at. As explained in Section 1.1.2 and 1.1.3, there are multiple aspects to what makes test completion a challenging task. Getting an understanding of how a model handles these different challenges can only be done through manual evaluation.

To enable manual evaluation of the model's predictions as well as the quality of the dataset, a user interface is created. This user interface has two components with three different views: the *Data Explorer* (consisting out of a list and detail view) and the *Prediction Explorer*.

As the name suggests, the *Data Explorer* allows for exploring the dataset. The user can browse through the datapoints using its list view (Figure 4.5) and assess individual datapoints in the detail view (Figure 4.4). The latter provides detailed information about the datapoint, like the *Given*, *When* and *Then* extracted for it, as well as information about the test's context. However, this context also includes methods outside of the test classes file, which therefore won't be part of the *Test Declaration* AST (Section 3.2.2).

The *Prediction Explorer* (Figure 4.6) view provides the user with the option to enter test and target code himself, displaying the predicted *Then* section based on the chosen model and sampler. This enables analyzing nuances in the model's behavior, which can not be measured by metrics like BLEU and ROUGE.

The user interface is implemented as a web-application using the *JavaScript* library *Vue.js*[3]. For the backend, the *Python* framework *Flask*[4] is employed to create an HTTP API for the user interface. This API provides endpoints for browsing through the dataset (used by *Data Explorer*), as well as creating predictions employing the prediction pipeline (used by *Prediction Explorer*).

[3] `https://vuejs.org`
[4] `https://flask.palletsprojects.com`

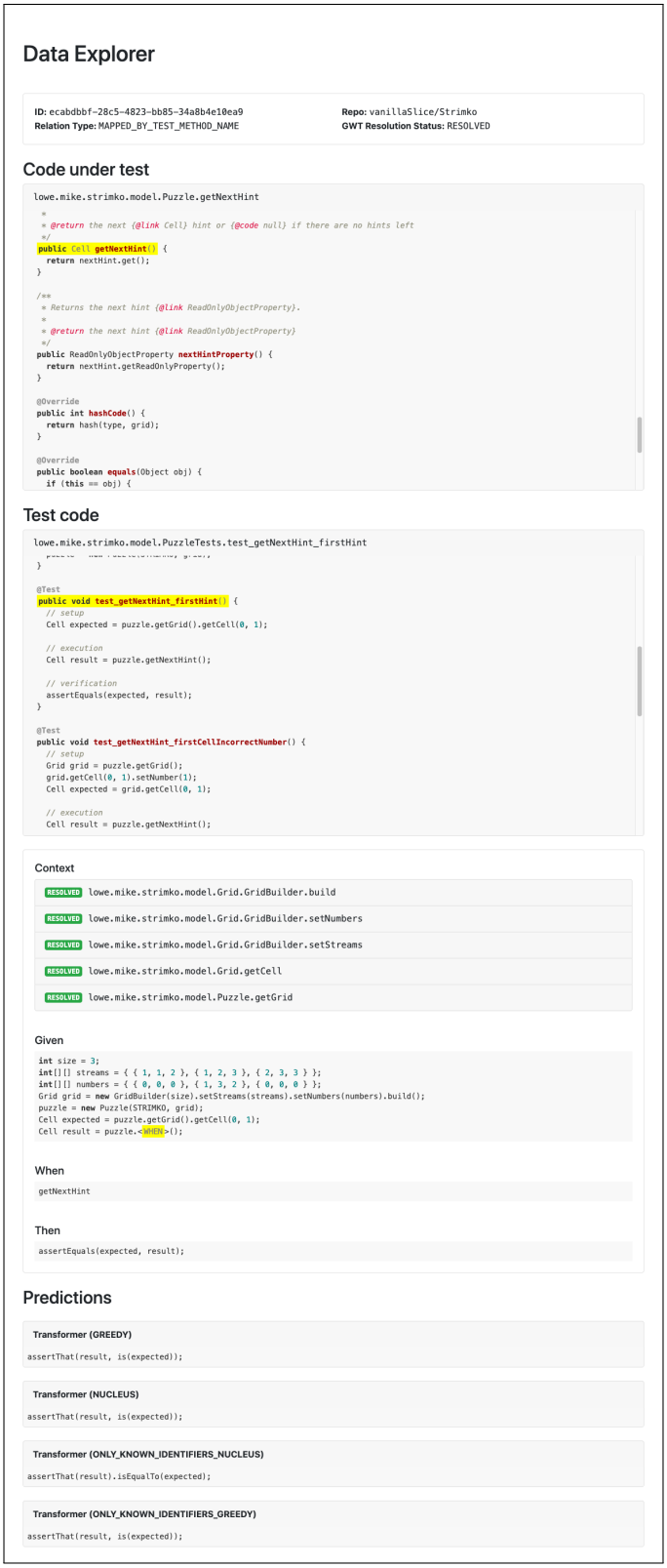


Figure 4.4: Scroll-through image of the *Data Explorer*'s detail view. It highlights the target as well as the test method, while the user can scroll through the corresponding files (top). The extracted *Given*, *When* and *Then* are shown, in addition to the test's context (middle). At the bottom predictions generated for this datapoint are shown using the model with different samplers (Section 3.4.1).

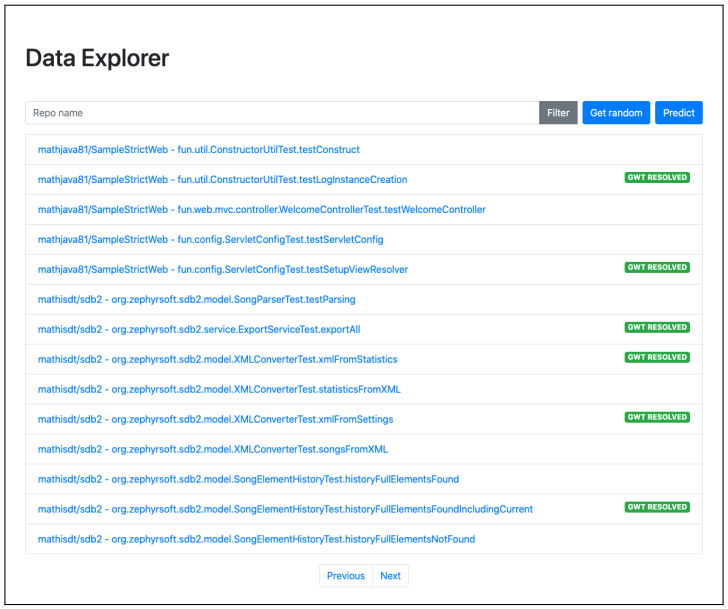


Figure 4.5: Image of the *Data Explorer*'s list view. It shows a paginated, searchable list of all datapoints, highlighting those which have been GWT resolved (Section 3.1.3). By selecting a datapoint in the list the *Data Explorer*'s detail view is opened (Figure 4.4) and by pressing the *PREDICT* button the user is navigated to the *Prediction Explorer*. Selecting *RANDOM* will open a randomly selected datapoint.

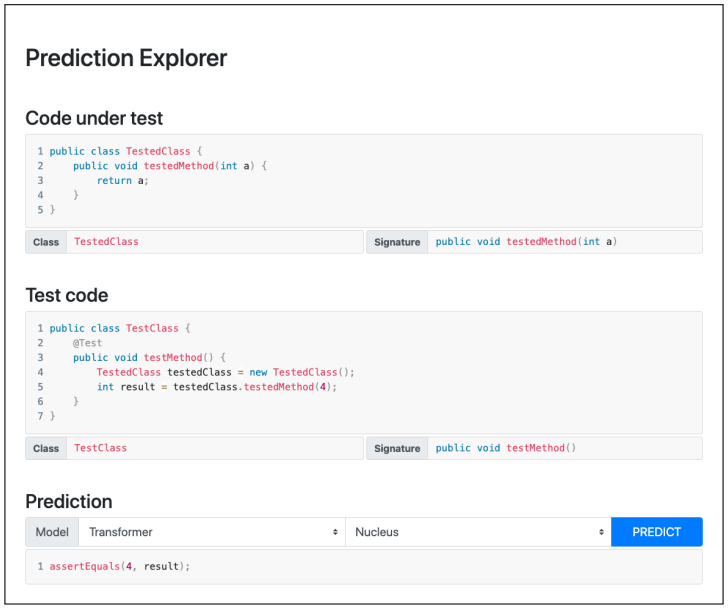


Figure 4.6: Image of the *Prediction Explorer* view. It provides two code editor fields, one for the target method (top) the other for the test method (middle). To identify the test/target method and class within the editors their names have to be provided by the user. When *PREDICT* is pressed after selecting the model and sampler to use, the predicted *Then* section is displayed to the user (bottom).

4.2 RQ1: Dataset quality

RQ1 posed the question if automatically finding tests in unlabeled source code, mapping them to their target methods and identifying their GWT sections, could be done reliably enough to create a dataset of sufficient quality to train a machine learning model on. With Section 3.1 proposing an approach to creating such a dataset, this experiment evaluates how reliably this approach identifies a test's target method and GWT sections.

4.2.1 Experiment

Since there is no way of automatically evaluating the quality of datapoints, a manual evaluation process is employed. To do so, 100 randomly selected datapoints are reviewed, assessing whether the correct target method and GWT sections have been identified. Since a falsely matched target method implies a incorrectly labeled *When*, the GWT sections are only reviewed if the target method has been correctly identified.

As this is the only experiment presented in Chapter 4 which does not involve training a machine learning model, it does not employ the setup described in Section 4.1. However, the *Data Explorer* user interface presented in Section 4.1.4 is used for manual assessment of the datapoints.

4.2.2 Results

The results of this experiment are presented by Table 3.1. They show that 94 of the 100 reviewed tests were correctly mapped to their target methods and the *GWT* sections were correctly identified for all of those tests. While these results should be taken with a grain of salt, due to the small sample size, they indicate that about 94% of the datapoints provide correct information.

Datapoint type	**Amount**
Correctly matched target method	94
GWT correctly identified	94
GWT falsely identified	0
Falsely matched target method	6

Table 4.3: Results of the experiment described in Section 4.2.1.

Having 6% falsely labeled datapoints in a dataset this size could lead to the model learning undesirable behavior. However, since most methods called in a test contribute to building up the state tested in the *Then* section, even falsely matched

target methods are not completely unrelated to the actual target method in most cases. For example, Listing 4.1 shows the test `write_SpecifiedExtensionIsJPEG_SpecifiedOutputFormatIsjpg` from the *thumbnailator*[5] project. In this case the method `setOutputFormatName` (line 5) was falsely identified as the target method, because it shares the tokens `output` and `format` with the test's name, while the correct target method's (line 6) name only shares the token `write`.

```
1  @Test
2  public void
   ↪write_SpecifiedExtensionIsJPEG_SpecifiedOutputFormatIsjpg
   ↪() throws IOException {
3      File f = TestUtils.createTempFile(TMPDIR, "JPEG");
4      FileImageSink sink = new FileImageSink(f);
5      sink.setOutputFormatName("jpg");
6      sink.write(new BufferedImageBuilder(100, 100).build());
7      assertEquals(f, sink.getSink());
8      assertTrue(f.exists());
9      assertEquals("JPEG", TestUtils.getFormatName(new
       ↪FileInputStream(f)));
10 }
```

Listing 4.1: The test `write_SpecifiedExtensionIsJPEG_SpecifiedOutputFormatIsjpg` from the class `FileImageSinkTest` in the *thumbnailator* project. Here the method `setOutputFormatName` (line 5) was falsely identified as the target method, because its name is considererd more similar to the test's name than the name of the correct target method `write` (line 6).

But even though `setOutputFormatName` is not the correct target method it still is fairly relevant to the test. Therefore, a model being trained with this information would not learn something completely wrong. Since these type of mismatches happen because a non-target method's name matches the most tokens with the test's name, the matched method usually still is relevant to the test in some way, hence it being in the test's name. Thus, all of the 6 falsely matched datapoints discovered during the reviewing process have been found to be relevant to the conditions tested in the *Then* section.

4.3 RQ2: Model's ability to generate parsable code

A central requirement the model has to fulfill to be able to generate meaningful *Then* sections is it having a understand of the syntactic structure of source code. Not only

[5] `https://github.com/coobird/thumbnailator`

is this an important skill for the model to interpret its input, but also to be able to generate parsable code as its output. Therefore, RQ2 questioned the model's ability to generate parsable source code, which will be addressed by the experiment in this section.

4.3.1 Experiment

As for all experiments, the model (Section 3.3) is trained using the setup explained in 4.1. The obvious metric choice for evaluating the trained model's ability to generate parsable code is the unparsable rate (Section 4.1.4). However, it is assessed in conjunction with the max length exceed and unknown token generated rate, to make sure the model isn't achieving a low unparsable rate by failing in other areas. For example, having a unknown token generated rate of 100% will also yield an unparsable rate of 0%, which should not be considered a success.

Since the chosen sampling method influences the predictions as explained in Section 3.4.1, the model is evaluated using different samplers to analyze if their abilities to sample parsable code differ. Therefore, S_{greedy}, $S_{nucleus}$ as well as their filtered counterparts $S_{copy}|S_{greedy}$ and $S_{copy}|S_{nucleus}$ are used.

4.3.2 Results

Metric	S_{greedy}	$S_{copy}\|S_{greedy}$	$S_{nucleus}$	$S_{copy}\|S_{nucleus}$
Unparsable rate	**1.92%**	1.93%	3.60%	4.29%
Unknown token generated rate	0.32%	**0.29%**	0.36%	0.31%
Max length exceeded rate	9.06%	11.69%	**0.76%**	2.47%

Table 4.4: Results of the experiment described in Section 4.3.1.

Table 4.4 presents the results of the experiment described in Section 4.3.1. With 1.92% being the lowest unparsable rate achieved, the model's ability to produce parsable code can be considered high. The greedy approach has proven to sample parsable sequences most reliably, with *Nucleus* performing notably worse. The copying pre-sampler has a negative effect on the unparsable rate in both cases.

Assessing the max length exceeded and unknown token generated rate, does not show any threats to the validity of the unparsable rate. However, although the greedy sampler outperforms *Nucleus* on the unparsable rate, it yields a noticeably higher max length exceeded rate. Once again, the copying pre-sampler only negatively effects the max length exceeded rate, although it shows some negligible improvements of the unknown token generated rate.

4.4 RQ3: Model's ability to generate *Then* sections

The most important criteria on which the success of the approach proposed in this thesis is judged is the model's ability to generate meaningful *Then* sections. Its ability do so is also questioned by RQ3 and will be quantitatively evaluated by the experiment conducted in this section.

4.4.1 Experiment

As explained in Section 4.1.4 the metrics BLEU and ROUGE are used to evaluate the quality of the model's predictions. To analyze the effects different samplers have on those scores, the same experiment setup and trained model is employed as for the experiment in Section 4.3.

4.4.2 Results

Table 4.5 presents the BLEU and ROUGE scores as well as their adjusted variants achieved by the model, depending on the sampling method used. These results show the greedy sampler to outperform other sampling methods across the board. Since greedy also yields the lowest unparsable rate, as shown by the previous experiment (Section 4.3.2), the adjusted scores go even more into its favor.

To put the achieved BLEU scores into perspective: the state of the art NLP model GPT-3 achieves BLEU scores ranging from 21 to 40.6 in various language translation tasks [19]. Due to the different nature of the tackled tasks, these results are hardly comparable to those in Table 4.5. However, language translation is generally considered a task handled fairly well by nowadays deep learning approaches. Therefore, this work's model yielding similar BLEU scores indicates promising performance.

4.5 RQ3: Model's ability to overcome the challenges of test completion

Although the BLEU and ROUGE scores achieved during the quantitative evaluation in Section 4.4.2 are competitive in the realm of neural machine translation, it is unclear how these scores translate to actual prediction quality in real-world test completion applications. Due to the lack of previous research, the achieved scores can't be compared to others to put the model's performance into perspective. For the same reason, there is no general understanding of how well state-of-the-art machine

Metric	S_{greedy}	$S_{copy}\|S_{greedy}$	$S_{nucleus}$	$S_{copy}\|S_{nucleus}$
BLEU	**38.192**	34.299	34.326	31.785
$BLEU_1$	**56.316**	53.630	53.200	51.699
$BLEU_2$	**42.353**	38.901	38.627	36.407
$BLEU_3$	**32.854**	28.904	28.927	26.349
$BLEU_4$	**27.152**	22.951	23.355	20.581
ROUGE-L	**68.607**	66.451	66.308	65.091
ROUGE-L_P	**72.806**	71.825	69.976	69.640
ROUGE-L_F	**70.644**	69.034	68.093	67.289
ROUGE-1	**70.044**	67.951	67.993	66.926
ROUGE-1_P	**74.487**	73.615	71.821	71.704
ROUGE-1_F	**72.197**	70.670	69.854	69.232
ROUGE-2	**55.035**	52.012	51.642	49.805
ROUGE-2_P	**57.886**	55.585	53.820	52.665
ROUGE-2_F	**56.425**	53.739	52.709	51.196
adj(BLEU)	**37.337**	33.538	32.967	30.323
adj($BLEU_1$)	**55.054**	52.439	51.093	49.321
adj($BLEU_2$)	**41.404**	38.038	37.098	34.732
adj($BLEU_3$)	**32.118**	28.262	27.782	25.137
adj($BLEU_4$)	**26.544**	22.441	22.430	19.634
adj(ROUGE-L)	**67.070**	64.976	63.682	62.097
adj(ROUGE-L_P)	**71.175**	70.230	67.205	66.436
adj(ROUGE-L_F)	**69.062**	67.501	65.396	64.193
adj(ROUGE-1)	**68.475**	66.443	65.300	63.847
adj(ROUGE-1_P)	**72.818**	71.981	68.977	68.406
adj(ROUGE-1_F)	**70.580**	69.101	67.088	66.048
adj(ROUGE-2)	**53.803**	50.858	49.597	47.514
adj(ROUGE-2_P)	**56.589**	54.351	51.688	50.243
adj(ROUGE-2_F)	**55.161**	52.546	50.621	48.841

Table 4.5: Results of the experiment described in Section 4.4.1.

learning systems perform at test completion, like there is for neural machine translation. In an attempt to develop such an understanding the following experiment qualitatively evaluates the model by examining predictions created by it, thereby analyzing its abilities/disabilities to handle the various challenges (Section 1.1.2) and complexities (Section 1.1.3) posed by test completion.

4.5.1 Experiment

To allow for manual assessment of predictions created by the model the *Prediction Explorer* presented in Section 4.1.4 is used. The model trained for the experiments in Section 4.3 and 4.4 is employed to generate the predictions. Also, only the greedy sampler is used, as it has shown to outperform other sampling methods in previous experiments (Section 4.3.2 and Section 4.4.2).

To assess how the model handles the challenges and complexity types presented in Section 1.1.2 and 1.1.3, artificial testing scenarios are constructed to reveal different aspects of the model's behavior. While this manual evaluation process can't quantify performance nor verify that observed behavior generalizes to other scenarios, it helps in getting a sense of the model's abilities and disabilities.

4.5.2 Results

This section presents the previously mentioned test scenarios, as well as the resulting *Then* sections predicted by the model (highlighted in green). Each scenario is discussed in detail, analyzing the challenges posed and how the model handles them. Also, it is observed how the model reacts to variations of the scenario, thereby concluding on hypotheses about the model's behavior.

To ensure that the model isn't just copying *Then* sections it has seen during training, it has been verified that none of the *Then* sections predicted in the following occur in the same (or similar) way in the dataset. This was done by querying the dataset for *Then* sections which have at least one line of code in common with the predicted *Then* section. If a datapoint was found it was manually assessed whether it should be considered similar to the predicted *Then* section or if they just happened to have a individual line of code in common without sharing any more similarities. Also, while only a few scenarios are presented in the following for brevity's sake, the described behaviors have been observed in multiple instances.

Scenario 1

Listing 4.2 shows Scenario 1. Here the method `ListFactory.create()` is tested by `ListFactoryTest.testCreate()`. As the target method initializes an `ArrayList` and mutates its state before returning it, the challenge in understanding this method comes through its state complexity (Section 1.1.3).

```
1  public class ListFactory {
2      public List<Integer> create() {
3          List<Integer> list = new ArrayList<Integer>();
4          list.add(1);
5          list.add(2);
6          list.add(3);
7          return list;
8      }
9  }
10
11 public class ListFactoryTest {
12     @Test
13     public void testCreate() {
14         ListFactory listFactory = new ListFactory();
15         List<Integer> list = listFactory.create();
16         assertEquals(3, list.size());
17         assertEquals(1, list.get(0));
18         assertEquals(2, list.get(1));
19         assertEquals(3, list.get(2));
20     }
21 }
```

Listing 4.2: The method `ListFactory.create()` initializes a `ArrayList<Integer>` with the values 1, 2 and 3. This behavior is tested by `ListFactoryTest.testCreate()`. The lines highlighted in green (line 16-19) contain the *Then* section predicted by the model.

The lines of code highlighted in green (line 16-19) contain the *Then* section predicted by the model for the test method `testCreate()`. This prediction reveals multiple behaviors of the model worth noting:

- The model seems to understand the interface of assert statements and that it is supposed to use them in the *Then* section to assert the *When* call meeting expectations.
- The model was able to infer from the *Given* section and test context that `list` is the variable containing the state that should be checked for correctness.

- The model was able to infer that `list` is of the type `List<Integer>` and therefore provides the methods `get()` and `size()`. It also seems to understand that those methods return integer values and therefore have to be compared to other integer values in the assert statements.

  Due to the `List` interface's popularity it has many occurrences in the training data. Therefore, the model most likely learned the methods of this interface by example instead of inferring it from the test's context. However, while `size()` always returns `int`, `get()` is generically typed returning `T` for an object of type `List<T>`. Nonetheless, the model was able to correctly predict that `get()` in line 17-19 returns `Integer`, since `list` is of the type `List<Integer>`.

- Most importantly, the model correctly predicted the state `list` is expected to be in, indicating that it was able to handle the state complexity posed by `ListFactory.create()`. It anticipated `list` to contain three items (line 16) and correctly assumed those to have the values 1, 2 and 3 at the corresponding positions (line 17-19).

It would be fair to assume that the model is just applying a pattern it has observed in the training data, without having any understanding of the before mentioned points. To put this to the test some variations are made to Listing 4.2, while observing if the model is able to adapt its predictions accordingly.

Firstly, `list.add(4);` is inserted after line 6, to find out if the model is able to generalize the shown behavior to `ListFactory.create()` initializing a different list. The *Then* Section predicted by the model for this variation is shown by Listing 4.3. The expected size of `list` has been adjusted to 4 (line 1) and an assertion checking for the new item 4 at position 3 has been added, showing that the model is able to adapt to this change.

```
1 assertEquals(4, list.size());
2 assertEquals(1, list.get(0));
3 assertEquals(2, list.get(1));
4 assertEquals(3, list.get(2));
5 assertEquals(4, list.get(3));
```

Listing 4.3: *Then* section predicted by the model, after inserting `list.add(4);` after line 6 in Listing 4.2.

When `list.add(5);` is added after `list.add(4);` the model predicts the *Then* section shown by Listing 4.4. Interestingly, it uses `intValue()` in this case to return the `int` values from the boxed `Integer` objects contained by `list`. While this isn't

necessary, as boxed types can be compared to their primitive counterparts, it isn't wrong either. However, although the model is able to check for the newly added item 5, it wrongly assumes `list` to have the length 4, showing that the model is not able to transfer the previously observed behavior unconditionally.

```
1 assertEquals(4, list.size());
2 assertEquals(1, list.get(0).intValue());
3 assertEquals(2, list.get(1).intValue());
4 assertEquals(3, list.get(2).intValue());
5 assertEquals(4, list.get(3).intValue());
6 assertEquals(5, list.get(4).intValue());
```

Listing 4.4: *Then* section predicted by the model, after inserting `list.add(4); list.add(5);` after line 6 in Listing 4.2.

While this prediction is right for the most part (lines 2-6) line 1 will make the test fail. This highlights how sensitive test completion can be to small mistakes by the model. Nonetheless, this prediction would score an (excellent) BLEU score of 97.470, assuming the same *Then* section with `assertEquals(5, list.size());` in line 1 to be the target data. That failing tests can achieve high BLEU scores shows that while BLEU is good for evaluating how similar a prediction is to target data, it is not suitable to assess how applicable a test completion model is to real-world usage.

Scenario 2

Scenario 2 was created to analyze how the model handles algorithmic complexity. It is an advanced version of Scenario 1 introducing additional algorithmic complexity through the addition of a `for`-loop in `ListFactory.create()`, as shown by Listing 4.5.

As the predicted *Then* section in Listing 4.5 shows, similar behavior is observed as in Scenario 1. Beyond that, the model seems to handle the algorithmic complexity well, interpreting how the `for`-loop in `ListFactory.create()` influences the returned list.

However, changing the *When* call in line 15 to `listFactory.create(4)` reveals that this isn't entirely true, as the resulting prediction in Listing 4.6 shows. While the model is not able to correctly predict the contents of `list` in this case, its prediction once again is not completely wrong. It correctly assumes the length of `list` to be 4 and it chooses the correct approach of accessing each item in the list asserting its value. Nonetheless, the asserted values are incorrect and the item at position 4 is accessed in line 10 although `list` only has 4 items.

```
1  public class ListFactory {
2     public List<Integer> create(int n) {
3        List<Integer> list = new ArrayList<Integer>();
4        for (int i = 0; i <= n; i++) {
5           list.add(i);
6        }
7        return list;
8     }
9  }
10
11 public class ListFactoryTest {
12    @Test
13    public void testCreate() {
14       ListFactory listFactory = new ListFactory();
15       List<Integer> list = listFactory.create(3);
16       assertEquals(3, list.size());
17       assertEquals(0, list.get(0));
18       assertEquals(1, list.get(1));
19       assertEquals(2, list.get(2));
20       assertEquals(3, list.get(3));
21    }
22 }
```

Listing 4.5: Advanced version of Scenario 1 (Listing 4.2) where the algorithmic complexity of `ListFactory.create()` has been increased through the addition of a `for`-loop.

```
1  @Test
2  public void testCreate() {
3     ListFactory listFactory = new ListFactory();
4     List<Integer> list = listFactory.create(4);
5     assertEquals(4, list.size());
6     assertEquals(4, list.get(0));
7     assertEquals(4, list.get(1));
8     assertEquals(4, list.get(2));
9     assertEquals(5, list.get(3));
10    assertEquals(6, list.get(4));
11 }
```

Listing 4.6: Variation of `ListFactoryTest.testCreate()` in Listing 4.5 calling `listFactory.create` in line 4 with the parameter 4 instead of 3. The *Then* section predicted by the model (lines 5-10) is highlighted in green.

Scenario 3

Scenario 3 is another advanced version of Scenario 1 introducing integration complexity (Section 1.1.3). As Listing 4.7 shows, `ListFactory.create()` implements the same behavior as Scenario 1. However, the actual operation (in `doVeryPrivateStuff()`) is hidden behind three methods integrating each other (`create()`, `doStuff()` and `doPrivateStuff()`). This scenario tests if the model is able to trace what a method is doing throughout its call stack.

As the prediction in Listing 4.7 shows, the model was able to generate a meaningful *Then* section. Arguably, the *Then* section predicted in this case is even better than those in Scenario 1 and 2, since it tests the same conditions with only one assert statement instead of four (like in Listing 4.2). This result indicates that the model is able to handle the integration complexity posed by this scenario. However, the following variations to the scenario challenge the idea that the model achieves this by understanding the resulting call stack.

In Listing 4.8 the variable name of the `List<Integer>` object in line 15 is changed from `list` to `result`. As shown in line 28, this variation results in the model generating an assert statement testing the state of `result`. However, the variable `result` is not defined in the scope of `testCreate()`. This indicates that the model wasn't able to fully understand that the methods `create()`, `doStuff()`, `doPrivateStuff()` are called by each other, returning the result of `doVeryPrivateStuff()`. Rather, it most likely just recognized the pattern of a list being initialized somewhere in the test's context, which has led to assertions being created for that list on previously seen data and therefore generates such assertions as well.

Also, this highlights how sensible the model is to variable names. The variable name `result` is commonly used to initialize variables in the *Given* section which are then used as part of the assertions in the *Then* section. So instead of inferring from the context which variable should be used in the *Then* section, the model seems to choose variables with a fitting name, even if that variable is not accessible in the scope of the *Then* section.

This behavior becomes even more apparent when looking at the variation in Listing 4.9. While following the same idea as the previous variation, it slightly obscures the roles of the `list` variables by using the variable names `a` (line 15) and `b` (line 27). This way the model would only be able to understand that `a` and `b` reference the same object by inferring the call stack. Also, it would need to understand that `a` references the return value of the *When* call to know that `a` should be used in the assert statements.

However, the resulting prediction (lines 28-31) shows that the model generates asser-

```
1  public class ListFactory {
2     public List<Integer> create() {
3        return this.doStuff();
4     }
5
6     private List<Integer> doStuff() {
7        return this.doPrivateStuff();
8     }
9
10    private List<Integer> doPrivateStuff() {
11       return this.doVeryPrivateStuff();
12    }
13
14    private List<Integer> doVeryPrivateStuff() {
15       List<Integer> list = new ArrayList<Integer>();
16       list.add(1);
17       list.add(2);
18       list.add(3);
19       return list;
20    }
21 }
22
23 public class ListFactoryTest {
24    @Test
25    public void testCreate() {
26       ListFactory listFactory = new ListFactory();
27       List<Integer> list = listFactory.create();
28       assertEquals(Arrays.asList(1, 2, 3), list);
29    }
30 }
```

Listing 4.7: Advanced version of Scenario 1 (Listing 4.2) introducing additional integration complexity by hidding the actual operation (in `doVeryPrivateStuff()`) behind three methods integrating each other (`create()`, `doStuff()` and `doPrivateStuff()`). The *Then* section predicted by the model (line 28) is highlighted in green.

```
12 // Definition of ListFactory ...
13
14    private List<Integer> doVeryPrivateStuff() {
15       List<Integer> result = new ArrayList<Integer>();
16       result.add(1);
17       result.add(2);
18       result.add(3);
19       return result;
20    }
21 }
22
23 public class ListFactoryTest {
24    @Test
25    public void testCreate() {
26       ListFactory listFactory = new ListFactory();
27       List<Integer> list = listFactory.create();
28       assertEquals(Arrays.asList(1, 2, 3), result);
29    }
30 }
```

Listing 4.8: Variation of Listing 4.7, where the lines 1-12 are omitted. The behavior of `doPrivateStuff()` also is the same as the original. However, the variable name of the object initialized in line 15 has been changed from `list` to `result`. The *Then* section predicted by the model (line 28) is highlighted in green.

```
12 // Definition of ListFactory ...
13
14    private List<Integer> doVeryPrivateStuff() {
15       List<Integer> b = new ArrayList<Integer>();
16       b.add(1);
17       b.add(2);
18       b.add(3);
19       return b;
20    }
21 }
22
23 public class ListFactoryTest {
24    @Test
25    public void testCreate() {
26       ListFactory listFactory = new ListFactory();
27       List<Integer> a = listFactory.create();
28       assertEquals(3, list.size());
29       assertEquals(1, (int) list.get(0));
30       assertEquals(2, (int) list.get(1));
31       assertEquals(3, (int) list.get(2));
32    }
33 }
```

Listing 4.9: Variation of Listing 4.7, where the lines 1-12 are omitted. The behavior of `doPrivateStuff()` also is the same as the original. However, the variable name of the object initialized in line 15 has been changed from `list` to `b` and the return value of `listFactory.create()` in line 27 is assigned to the variable `a` instead of `list`. The *Then* section predicted by the model (line 28) is highlighted in green.

tions using the variable `list`, although this variable name is not used anywhere in `ListFactory` nor `ListFactoryTest`. This further strengthens the assumption that the model just recognizes a pattern in `doVeryPrivateStuff()`, which it associates with the name `list`, instead of understanding that an object is initialized and assigned to the variable `a`.

The model not being able to fully understand how these pieces of code are connected to each other could be considered a failure. However, the correct predictions in Scenario 1 and 2 as well as the decent BLEU score in Section 4.4.2 indicate that recognizing known patterns while relying on variable names to put the pieces together is a fairly reliable workaround to the complexity posed by test completion. Although this approach will not lead to perfect predictions in all cases, it can lead to correct predictions in many.

4.6 RQ4: Model's ability to leverage AST encoded data

Applying the sequentialized AST encoding explained in Section 3.2.3 to the model's input leads to fairly large input sequences when compared to the size of the represented code. For example, using this encoding on the expression `int i = 3;` results in a sequence with 23 tokens, while simply tokenizing the same expression results in the sequence `["int", "i", "=", "3", ";"]` with only 5 tokens.

As discussed in Section 4.1.2, an increase of the model's input size considerably increases the cost of training it. Therefore, this experiment is aimed at finding out if the model is able to leverage the source code's syntactic structure encoded in the inputted AST. Besides answering RQ4, the motivation behind this experiment is evaluating if employing the sequentialized AST encoding warrants the computational cost by yielding better performance.

4.6.1 Experiment

To evaluate if the model benefits from the sequentialized AST encoding, a variant of it is used which receives tokenized source code as input and is trained to generate source code in the same format. As this variant does not use ASTs to encode the source code, it acts as a baseline to the main model (trained in Section 4.3 and 4.4). Comparing the main model's performance to that of this baseline reveals the difference sequentialized AST encoding makes.

Tokenized code as the model's input/output

As explained in Section 3.2.2, the main model's input isn't just a single connected piece of code which is then AST encoded, but a tree with multiple AST subtrees representing the test and target methods as well as the methods in their context. While those AST subtrees can be represented as sequences of tokenized code, their parent nodes can't as there is no code representation for the custom nodes used in the *Test Declaration* AST (Figure 3.3). To make sure differences between the main model and baseline are not the result of their inputs encoding different information, an alternative encoding of the *Test Declaration* AST is introduced, which contains the same information represented by the *Test Declaration* AST's subtrees without relying on a tree structure. This alternative encoding joins the tokenized code sequences (representing the AST subtrees) with special marker tokens symbolizing the custom nodes. Also, a marker is inserted in the code sequence representing the test method's body to mark the location of the *Then* section. An example of this is illustrated in Figure 4.7. As *javalang* is used during the main model's evaluation process (Section 4.1.4) to tokenize the prediction pipeline's output, it is also used for tokenizing the baseline's input and output.

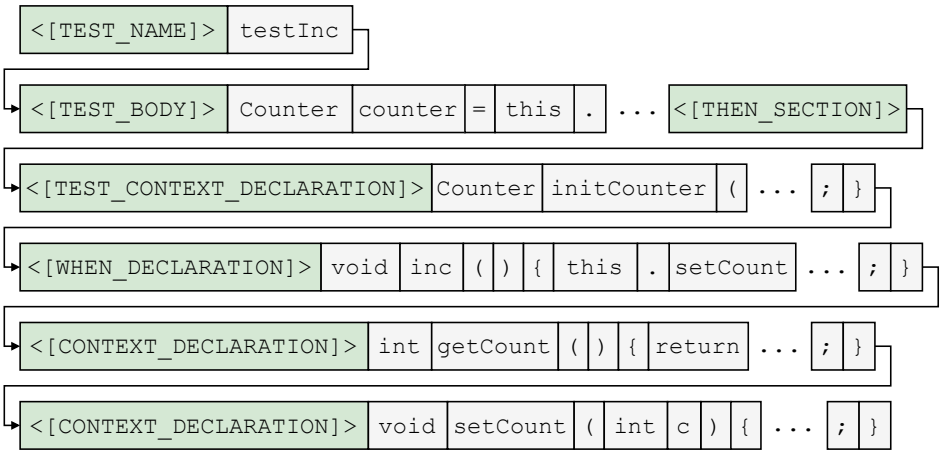


Figure 4.7: Illustration of how the *Test Declaration* AST in Figure 3.3 (which encodes Listing 3.13) is encoded using code tokenization for the baseline model. The marker tokens representing the *Test Declaration* AST's custom nodes are highlighted in green. While most markers occur exactly once, context markers (`<[TEST_CONTEXT_DECLARATION]>` and `<[CONTEXT_DECLARATION]>`) can have 0 to n occurrences depending on the size of the context.

To train the model to output tokenized code, the target data is also encoded using code tokenization. As the main model's target data is a single AST which does not use a custom tree format like the *Test Declaration* AST, the corresponding tokenized code is used as the baseline's target data.

Once the input/output is encoded using code tokenization, the resulting sequence is further encoded in the same way as the main model's sequentialized AST is (Section 3.2). However, since tokenized code does not encode which tokens are value tokens, BPE is applied to all tokens except for marker tokens.

As previously done for the data the main model was trained on, the dataset (Section 3.1) is encoded in a preprocessing step. Also, since the employed encoding results in different tokens than the main model's, a new BPE and dataset vocabulary are created during preprocessing. As BPE is applied to all tokens except for marker tokens, the size of the dataset vocabulary is equivalent to the BPE vocabulary (16,000) in addition to the marker tokens (6).

As described in Section 4.1.4 the evaluation process requires the prediction pipeline to decode the model's output. Conveniently, the decoding process for tokenized code is more straightforward than for sequentialized AST, as it does not require reconstructing the tree from the sequence to generate code from it. After they have been BPE decoded, the outputted tokens can simply be joined by whitespaces to receive parsable *Java* code (assuming the generate output is parsable). Although *JavaParser* is not needed for this decoding process, the decoded output is still parsed using it. Thereby the parsability of the baseline's output is verified in the same way it is for the main model. This parsability check is implemented using the *Java* server explained in Section 3.4.2.

Setup

Since the main model's hyperparameters were optimized in Section 4.2, the same has to be done for the baseline to ensure that difference in performance are not due to one model being better optimized than the other. Therefore, the same hyperparameter optimization process using *Optuna* is applied which has been described in Section 4.2.

Just like in Section 4.5, only the greedy sampler is used due to it outperforming other sampling methods in previous experiments (Section 4.3.2 and 4.4.2).

4.6.2 Results

Table 4.6 shows the results of evaluating the baseline and main model on the full test split. It should be noted that the main model's BLEU and ROUGE scores slightly deviate from those presented in Section 4.4.2, as only datapoints are considered for BLEU and ROUGE which both model's generate parsable predictions for.

Due to the previously mentioned differences in sequence length resulting from

Metric	Baseline (tokenized code)	Main model (sequentialized AST)
Unparsable rate	**1.07%**	1.91%
Unknown token generated rate	**0.24%**	0.27%
Max length exceeded rate	**3.79%**	9.41%
BLEU	33.702	**37.546**
BLEU_1	50.734	**56.284**
BLEU_2	37.485	**41.762**
BLEU_3	28.783	**32.123**
BLEU_4	23.570	**26.320**
ROUGE-L	67.935	**67.953**
ROUGE-L_P	70.092	**71.230**
ROUGE-L_F	68.996	**69.55**3
ROUGE-1	69.364	**69.456**
ROUGE-1_P	71.684	**72.972**
ROUGE-1_F	70.505	**71.170**
ROUGE-2	53.960	**54.036**
ROUGE-2_P	55.611	**56.240**
ROUGE-2_F	54.773	**55.116**
$adj(\text{BLEU})$	32.968	**37.054**
$adj(\text{BLEU}_1)$	49.628	**55.547**
$adj(\text{BLEU}_2)$	36.668	**41.215**
$adj(\text{BLEU}_3)$	28.155	**31.702**
$adj(\text{BLEU}_4)$	23.056	**25.975**
$adj(\text{ROUGE-L})$	66.454	**67.062**
$adj(\text{ROUGE-L}_P)$	68.564	**70.297**
$adj(\text{ROUGE-L}_F)$	67.492	**68.642**
$adj(\text{ROUGE-1})$	67.852	**68.546**
$adj(\text{ROUGE-1}_P)$	70.121	**72.016**
$adj(\text{ROUGE-1}_F)$	68.968	**70.238**
$adj(\text{ROUGE-2})$	52.784	**53.328**
$adj(\text{ROUGE-2}_P)$	54.399	**55.503**
$adj(\text{ROUGE-2}_F)$	53.579	**54.394**

Table 4.6: Results of the experiment described in Section 4.6.1.

tokenizing code compared to AST sequentialization, it comes as no surprise that the baseline's max length exceeded rate is much lower. On a similar note, regardless of a model's probability to predict a token making the input sequence unparsable, it is more likely to eventuate the longer the sequence becomes. Therefore, it is to be expected that the baseline yields a lower unparsable rate.

Besides those scores, the main model outperforms the baseline on every other metric, yielding a considerably better BLEU score (3.844). This also is the case for the adjusted BLEU/ROUGE metrics, although they factor in the baseline's superior unparsable rate.

However, considering that BLEU and ROUGE-N$_p$ both are precision based metrics, the discrepancy between how they portray differences in performance of the models seems striking. For example, although BLEU$_1$ and ROUGE-1$_p$ are based on the 1-gram precision, the difference between the main model's and baseline's BLEU$_1$ score is 5.55 while it only is 1.288 for ROUGE-1$_p$.

An explanation for this could be the difference between how BLEU and ROUGE average scores when applied to multiple datapoints. As explained in Section 4.1.4, BLEU micro-averages the results of multiple datapoints, while they are macro-averaged when using ROUGE. This theory can be verified by measuring BLEU for individual datapoints and then averaging all scores, which simulates a macro-averaged BLEU score. Doing so results in the BLEU scores presented in Table 4.7. Here the difference between both BLEU$_1$ scores is 1.086, which is more in line with the ROUGE-1$_p$ results.

Metric	Baseline (tokenized code)	Main model (sequentialized AST)
BLEU	38.222	**38.523**
BLEU$_1$	55.640	**56.726**
BLEU$_2$	45.050	**45.6307**
BLEU$_3$	36.926	**37.160**
BLEU$_4$	32.427	**32.477**

Table 4.7: BLEU scores resulting from the experiment described in Section 4.6.1 when they are measured per datapoint and then macro-averaged.

Since micro-averaging metrics weights a system's performance on long sequences stronger, there are two possible explanations for theses results. Either one of the models performs better on long sequences, or one model has a stronger tendency to generate sequences which are too long, therefore, yielding bad scores which are more heavily weighted. To further investigate this, Figure 4.8 illustrates the average BLEU scores the models score on individual datapoints in relation to the length of the predicted sequence. This reveals that the latter of the previously mentioned

explanations is the case. While both models seem to perform similarly on most datapoints, the baseline model generates considerably longer sequences for many. As these sequences are too long they score accordingly, which is weighted more heavily in the overall BLEU score.

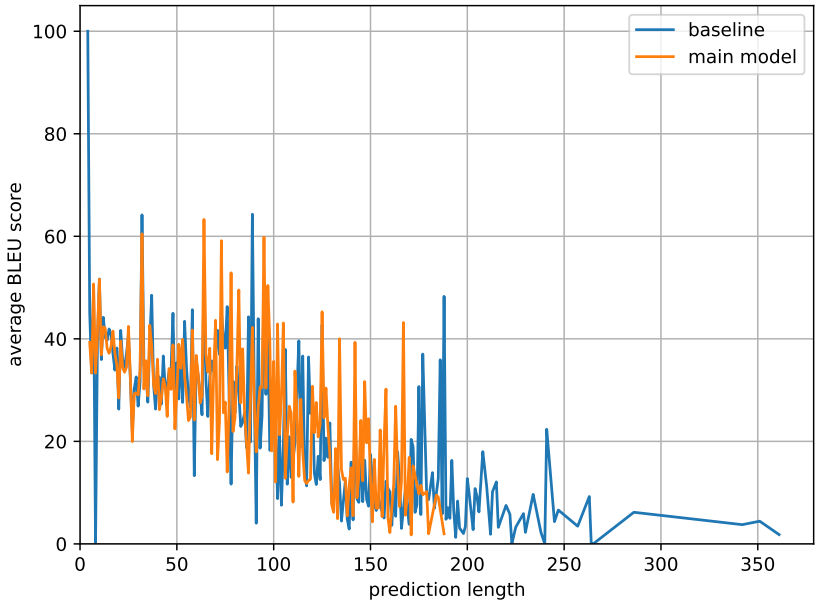


Figure 4.8: Visualization of the models average BLEU scores per datapoint (y-axis) in relation to the lengths of the code token sequences resulting from their prediction (x-axis).

It should be noted that the sequence lengths specified for the main model are not the lengths of the sequences outputted by the model, but of the sequences of tokenized code outputted by the prediction pipeline using the model (Section 4.1.4). Otherwise sequence lengths would not be comparable, as the AST sequences generated by the main model are considerably longer than those outputted by the baseline.

However, this also reveals that comparing the results in Table 4.6 could be considered somewhat unfair towards the baseline model. Due to length differences of the sequences outputted by the models, the prediction pipeline raises an max length exceeded error much earlier for the main model. Since datapoints exceeding the maximum prediction length are excluded from the evaluation with BLEU and ROUGE, the baseline ends up having predictions evaluated, which wouldn't be evaluated for the main model even if it predicted the same sequence (assuming no limitation in prediction length). Therefore, the baseline's BLEU score is penalized for predictions, which would be excluded from the main model's evaluation, as shown by Figure 4.8.

To counteract this disadvantage of the baseline model, both models are re-evaluated with the baseline's max length set to that of the longest code token sequence generated by the main model without exceeding the maximum length. The results of this are presented by Table 4.8.

Metric	**Baseline (max length = 188)**	**Main model**
Max length exceeded rate	14.45%	**9.41%**
BLEU	36.087	**37.819**
BLEU_1	53.811	**56.517**
BLEU_2	40.049	**42.037**
BLEU_3	30.921	**32.392**
BLEU_4	25.449	**26.584**
ROUGE-L	67.922	**68.166**
ROUGE-L_P	70.713	**71.354**
ROUGE-L_F	69.290	**69.724**
ROUGE-1	69.359	**69.660**
ROUGE-1_P	72.320	**73.085**
ROUGE-1_F	70.808	**71.332**
ROUGE-2	54.108	**54.288**
ROUGE-2_P	56.139	**56.421**
ROUGE-2_F	55.105	**55.334**
adj(BLEU)	35.614	**36.995**
$adj(\text{BLEU}_1)$	53.106	**55.285**
$adj(\text{BLEU}_2)$	39.525	**41.120**
$adj(\text{BLEU}_3)$	30.516	**31.685**
$adj(\text{BLEU}_4)$	25.115	**26.004**
adj(ROUGE-L)	67.033	**66.680**
adj(ROUGE-L_P)	69.787	**69.798**
adj(ROUGE-L_F)	68.382	**68.204**
adj(ROUGE-1)	68.450	**68.142**
adj(ROUGE-1_P)	71.372	**71.492**
adj(ROUGE-1_F)	69.881	**69.777**
adj(ROUGE-2)	53.400	**53.105**
adj(ROUGE-2_P)	55.404	**55.191**
adj(ROUGE-2_F)	54.383	**54.128**

Table 4.8: Results of the experiment described in Section 4.6.1, when the maximum prediction length of the baseline is limited to the length of the main model's longest prediction (188).

Decreasing the baseline's max length resulted in the removal of 1075 datapoints from the evaluation with BLEU and ROUGE, increasing its max length exceeded rate notably by 10.75 percentage points. Also, the main model still outperforms the baseline, although less convincingly than indicated by the initial results in Table 4.6.

To conclude, although the baseline yields a better unparsable rate, the main model also achieves a decent unparsable rate of 1.91%, whilst outperforming the baseline in every other aspect. Whereas the main model's BLEU and ROUGE scores are only slightly better, it has a notably better max length exceeded rate. This indicates that the model is benefiting from the information encoded in the AST, justifying the additional computational cost.

4.7 RQ5: Model's ability to understand a test's context

Information about a test's context is encoded in the *Test Declaration* AST inputted into the model (Section 3.2.2). RQ5 questions the model's ability to "understand" this information and apply this knowledge to predicting the test's *Then* section. This experiment is aimed at answering RQ5.

Similarly to the previous experiment in Section 4.6, there are two motivations to researching this question. For one, to understand the abilities and disabilities of the model for the sake of research. But on a more practical note, the size of the input sequence could considerably be decreased if the context information would turn out to not have any effect on the model's performance and therefore could be dropped from the *Test Declaration* AST. As previously explained in Section 4.6, this would decrease the computational cost of training the model.

4.7.1 Experiment

To evaluate the effect the context information has on the model's performance this experiment conducts an ablation study. For this a variation of the train and test data (Section 3.1) is created, in which the *TestContextDeclarations* and *ContextDeclarations* subtrees are dropped from the *Test Declaration* AST (illustrated in Figure 3.3) before sequentializing it.

Using this modified training split a new version of the model is trained, in the following referred to as *contextless* model, using the same hyperparameters as the main model. This model is evaluated using the regular evaluation process described in Section 4.1.4 and the results are compared to those of the main model. As in previous experiments, the greedy sampler is used during evaluation.

4.7.2 Results

Table 4.9 presents the results of the ablation experiment and compares the *contextless* model's performance to that of the main model.

Metric	*Contextless* model	Main model
Unparsable rate	**1.73%**	1.91%
Unknown token generated rate	**0.20%**	0.27%
Max length exceeded rate	16.71%	**9.41%**
BLEU	28.034	**38.409**
BLEU_1	47.782	**56.970**
BLEU_2	32.428	**42.661**
BLEU_3	22.752	**32.968**
BLEU_4	17.520	**27.162**
ROUGE-L	63.642	**68.961**
ROUGE-L_P	65.911	**72.297**
ROUGE-L_F	64.757	**70.589**
ROUGE-1	65.217	**70.444**
ROUGE-1_P	67.674	**74.017**
ROUGE-1_F	66.423	**72.187**
ROUGE-2	47.342	**55.291**
ROUGE-2_P	48.392	**57.531**
ROUGE-2_F	47.861	**56.389**
$adj(\text{BLEU})$	27.493	**37.572**
$adj(\text{BLEU}_1)$	46.860	**55.728**
$adj(\text{BLEU}_2)$	31.802	**41.731**
$adj(\text{BLEU}_3)$	22.313	**32.249**
$adj(\text{BLEU}_4)$	17.182	**26.570**
$adj(\text{ROUGE-L})$	62.414	**67.457**
$adj(\text{ROUGE-L}_P)$	64.639	**70.721**
$adj(\text{ROUGE-L}_F)$	63.507	**69.051**
$adj(\text{ROUGE-1})$	63.958	**68.909**
$adj(\text{ROUGE-1}_P)$	66.368	**72.404**
$adj(\text{ROUGE-1}_F)$	65.141	**70.613**
$adj(\text{ROUGE-2})$	46.428	**54.086**
$adj(\text{ROUGE-2}_P)$	47.458	**56.277**
$adj(\text{ROUGE-2}_F)$	46.937	**55.159**

Table 4.9: Results of the experiment described in Section 4.7.1.

While the unparsable and unknown token rate are fairly similar on both models, the *contextless* model's max length exceeded rate is even worse than the main model's. When looking at the other metrics the superiority of the main model becomes quite apparent. It outperforms the *contextless* model across the board and beats it by almost 10 BLEU points. This is a strong indication that the main model benefits from having the context information available and is able to apply this information to improve the quality of its predictions.

An explanation for these results can be found when looking at Listing 4.7, which was presented in Scenario 3 of Section 4.5. In this case there is no way of predicting a meaningful *Then* without knowing how `doVeryPrivateStuff()` is defined. However, when the context information is not encoded in the model's input, it does not know about this method. As it is very common for the tested logic to be implemented in a private method called by the target method, the *contextless* model is likely missing crucial information in a lot of cases, leading to poor prediction quality.

4.8 Project-based data split

A possible threat to the validity of the results presented in the previous sections is posed by the way the data split described in Section 4.1.2 was created. Since the datapoints in the splits were selected randomly from the dataset, the training and test split are likely to contain tests from the same project, possibly even testing the same class or method. This was a deliberate decision made due to the complexity of the tackled learning problem and the lack of previous research, to investigate the potential of the proposed approach (Chapter 3) in a somewhat simplified setting. However, it must be investigated whether the model was only able to achieve such promising BLEU scores by copying *Then* sections it has learned from the training data.

4.8.1 Experiment

To validate how depended the main model is on having previously seen tests from the project it is predicting for, it is trained and evaluated on a data split where no project has tests in more than one split. The proportions between the splits are the same as mentioned in Section 4.1.2 with the training split containing 80% of the data, while the validation and test split have 10%.

This data split is created by grouping all tests by their project name and sorting these project groups by their size in descending order. Project groups are taken from

that ordered list assigning them to one of the splits until that split has reached its relative target size. Once it has, the next split is filled with project groups. Since filling one split changes the relative size of the others, this process is repeated in a round-robin fashion until all data is distributed between the splits. By ordering the project groups first, it is made sure that all splits have similar proportions of large projects with many tests to small project with fewer.

The main model is then trained on the project-based training split using the hyperparameters resulting from the hyperparameter optimization in Section 4.1.3. As in previous experiments it is evaluated on a sample of 10,000 datapoints from the project-based test split and compared to the main model which was trained on the regular split in Section 4.4. During evaluation the greedy sampler is used for both models. However, it should be noted that they can't be evaluated on the same datapoints, due to the different test splits.

4.8.2 Results

Table 4.10 presents the results of the evaluation process executed on the main model trained on the project-based split and compares them to those achieved by the model trained on the regular split. Since these results reveal that the model performs significantly worse when trained on the project-based split in almost every aspect, two hypotheses are proposed providing possible explanations for these results:

H.1 When trained on the regular split the model learns general information about projects and their classes/methods, which it is able to apply to unseen test of that same project. Since it does not have this information when trained on the project-based split, it performs worse.

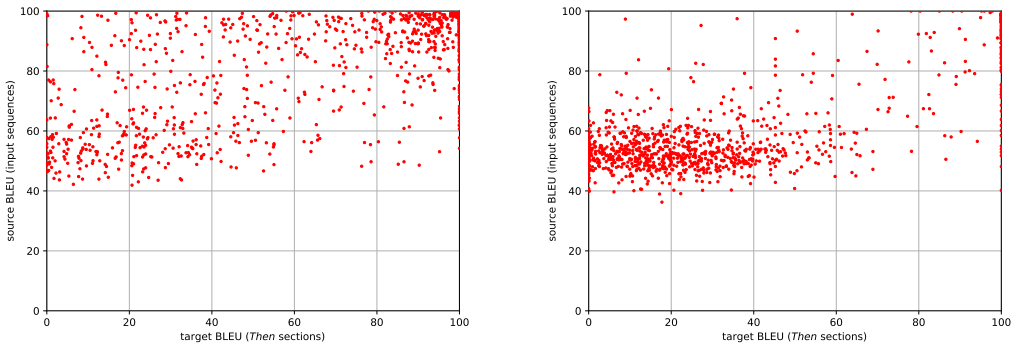
H.2 The model just learns to copy a *Then* section from a datapoint it has encountered during training, when receiving an input sequence similar to the one of that datapoint. If there are many datapoints in the regular test split which have similar source and target sequences to datapoints in the training split, the model could achieve good evaluation results using such an approach. As datapoints in the project-based split do not share these similarities, the model trained on it can't make use of the *Then* sections it has learned during training and therefore performs worse.

While both hypotheses could be true, the impact they have on the results shown in Table 4.10 is inversely proportional. Therefore, H.2 is investigated further, as disproving H.2 to be the most responsible for the results makes H.1 more likely to be.

Metric	Main model (project based split)	Main model (regular split)
Unparsable rate	**1.11%**	1.91%
Unknown token generated rate	**0.10%**	0.27%
Max length exceeded rate	30.43%	**9.41%**
BLEU	15.610	**37.268**
$BLEU_1$	37.981	**56.106**
$BLEU_2$	20.601	**41.485**
$BLEU_3$	11.021	**31.835**
$BLEU_4$	6.885	**26.033**
ROUGE-L	51.790	**67.648**
ROUGE-L_P	57.642	**70.875**
ROUGE-L_F	54.560	**69.224**
ROUGE-1	53.682	**69.165**
ROUGE-1_P	59.775	**72.628**
ROUGE-1_F	56.565	**70.854**
ROUGE-2	31.689	**53.581**
ROUGE-2_P	35.259	**55.756**
ROUGE-2_F	33.379	**54.647**
adj(BLEU)	15.421	**36.455**
adj($BLEU_1$)	37.520	**54.883**
adj($BLEU_2$)	20.351	**40.581**
adj($BLEU_3$)	10.888	**31.141**
adj($BLEU_4$)	6.802	**25.466**
adj(ROUGE-L)	51.162	**66.173**
adj(ROUGE-L_P)	56.942	**69.330**
adj(ROUGE-L_F)	53.897	**67.715**
adj(ROUGE-1)	53.031	**67.657**
adj(ROUGE-1_P)	59.050	**71.045**
adj(ROUGE-1_F)	55.879	**69.310**
adj(ROUGE-2)	31.305	**52.413**
adj(ROUGE-2_P)	34.831	**54.541**
adj(ROUGE-2_F)	32.974	**53.456**

Table 4.10: Results of the experiment described in Section 4.8.1.

For H.2 to have a significant impact on the results, the similarities between source sequences in the regular training and test split would need to correlate with the similarities of their corresponding target sequences, while there isn't such correlation for the project-based training and test split. To evaluate if this is the case, the similarity between the input sequence of a datapoint in the test set and every input sequence in the training split is measured. The datapoint in the training split which has the most similar input sequence is selected, as this is the datapoint from which the model would take the *Then* section if it was only copying. By evaluating the similarities between the *Then* sections of those two datapoints, it can be analyzed how well the model would score by outputting the *Then* section of the training datapoint with the most similar input sequence to the given test datapoint. The resulting similarity scores are plotted in Figure 4.9 after repeating the process for 1,000 datapoints randomly sampled from the regular test split (Figure 4.9a), as well as the project-based test split (Figure 4.9b) for comparison. The similarity measure used is the BLEU score, as copying a *Then* section which is similar according to BLEU will also yield good results during evaluation since BLEU is used as a metric. While calculating the BLEU scores the training split data is considered the prediction and the test data the target, as would be the case for a model copying from the training data.



(a) Regular split (b) Project-based split

Figure 4.9: For every datapoint in a 1,000 sample of the regular test split (a) and project-based test split (b) the similarity (measured with BLEU) between its input and the most similar one found in the corresponding training split is plotted along the x-axis. The y-axis shows how similar the *Then* sections of those two datapoints are.

As Figure 4.9a shows, there is a considerably higher amount of highly similar input sequences between the regular test and training split, than between the project-based ones (Figure 4.9b). However, while many of those datapoints in Figure 4.9a also correlate with highly similar *Then* sections, this is not the case for the majority of

those datapoints.

Looking closely at Figure 4.9a and 4.9b also reveals that there is a strikingly high amount of datapoints in both splits which have identical *Then* sections (BLEU = 100) independently of the input sequences similarity. A manual assessment of those *Then* sections exposed that they almost exclusively are variants of `assertEquals(-expected, result);`. Therefore, it comes as no surprise that this is a frequently occurring *Then* section. As this *Then* section does not correlate with similar input sequences as Figure 4.9 shows, the model still has to learn when to use it.

To conclude, the considerable amount of datapoints with highly similar source and target sequences in the regular split indicates that H.2 does have an impact on the model's performance. However, since the majority of datapoints with similar input sequences do not correlate with a similar target sequence, H.1 is likely to be the bigger factor on the model performing worse, when trained on the project-based split. This means that although Table 4.10 indicates that the model is not able to generalize its knowledge to unknown projects, it is able to apply information it has learned about a test's project to unknown tests of that same project.

Chapter 5

Conclusion

In this thesis a novel approach towards machine learning driven test completion was proposed. By doing so, two major contributions were made to the research community which could also leverage future work towards the ultimate goal of machine learning driven test generation. First of all, the largest dataset for test completion/generation available to date has been created, with almost 2.2 million datapoints mapping tests from 99k GitHub repositories to their target methods. Of those datapoints more than 1.3 million were successfully labeled using the GWT pattern.

The second contribution is a Transformer based model trained on this dataset to predict a test's *Then* section, given its target method and *Given* section. Besides being the first machine learning model to tackle the task of test completion, to the best of the author's knowledge, it also introduced two novel concepts, which could potentially benefit future work towards test generation and other source code processing machine learning approaches. Firstly, this thesis' approach was able to improve its model's performance by encoding the inputted source code using ASTs. While ASTs have been used in previous machine learning research [36, 26, 37, 25], they have not been in the existing test generation approaches [28, 29, 30]. Also, no previous research has been found which employs an AST sequentialization technique similar to that presented in Section 3.2.3, allowing to encode an AST as a sequence of tokens, whilst being able to fully decode it into source code without information loss.

However, the novelty benefiting the model's performance the most is the augmentation of the model's input with context information relevant to the currently predicted *Then* section. As explained in Section 3.2.2, AST analysis is used to identify the declarations of methods relevant to the test and target method. The ASTs of these methods are added as subtrees to the so-called *Test Declaration* AST which

is then inputted into the model. As the experiment in Section 4.7 has shown, this significantly increases the model's BLEU score from 28.034 to 38.409.

Research Questions

In Chapter 4 extensive experiments were conducted on the proposed approach to answer the research questions posed in Section 1.2. The following will briefly conclude on the results yielded by those experiments and how they contribute to answering the before mentioned research questions.

RQ1 questioned if a labeled dataset of sufficient size and quality, to train a test completion model with, could be created from unlabeled data. In Section 4.2 the quality of 100 datapoints was manually evaluated which were randomly selected from the previously mentioned dataset. While the GWT sections of all assessed datapoints were labeled correctly, 6% of the tests were matches with a incorrect target method. However, as all mismatched datapoints were mapped with a method very closely related to the target method instead, the quality of the datapoints can still be considered sufficient to not teach a model false behaviors. Since the dataset contains 1.3 million datapoints labeled for the task of test completion it also is more than sufficient in size.

RQ2 questioned the model's ability to generate parsable code. To address this research question, an experiment conducted in Section 4.3 evaluated which percentage of the predictions generated on the test data ended up being unparsable. With this unparsable rate being as low as 1.92%, the model showed great ability to understand the syntactic structure of source code. This also makes the presented model much better at generating parsable code than the previously introduced test generation models in [28] and [30]. However, it should be kept in mind that test generation is a more complex task while making this comparison.

RQ3 questioned the model's ability to generate meaningful *Then* sections and overcome the challenges of test completion covered by Section 1.1.2. Therefore, a qualitative evaluation of the model's performance was conducted in Section 4.4. During this evaluation the model achieved a BLEU score of 38.192, which is in line with BLEU scores yielded by state-of-the-art NMT models like [19] and higher than the BLEU score reported for the test generation model in [30]. While these model's are hardly comparable due to different tasks they tackle, it puts the achieved BLEU score into perspective, showing that it indicates promising performance.

This was further investigated by a qualitative evaluation in Section 4.5, showing that the model is able to generate meaningful *Then* sections in many cases. However, while the model did not struggle at any specific challenge (Section 1.1.2) or complex-

ity type (Section 1.1.3) posed by the test completion task, the results indicated that it heavily relies on recognizing certain patterns and variable names in the input. For example, when encountering the variable names `result` and `expected` in the test's *Given* section, it will very likely generated a *Then* section like `assertEquals(expected, result);`. While this can trick the model into wrong predictions in some cases, it is a legitimated approach yielding many meaningful *Then* sections, as supported by the BLEU scores resulting from the quantitative evaluation.

However, the experiment conducted in Section 4.8 revealed that there is an asterisk on these results. It showed that while generating a test's *Then* section the model is very reliant on information it has gathered from other tests in the same project. Therefore, Section 5.1 will make some suggestions on how future work could circumvent this shortcoming of the model, while leveraging its ability to apply knowledge it has gathered about a project to new *Then* sections for it.

RQ4 questioned the model's ability to leverage the additional syntactic information encoded in ASTs. By comparing the proposed model to a variant of it trained on data not AST encoded, Section 4.6 evaluated the impact the AST encoding has on the model's performance. Since the model performed better by 1.732 BLEU points when employing AST encoding, it seems to be able to leverage the additional information encoded in ASTs.

RQ5 questioned the model's ability to understand information about the test and target method's context embedded in its input. As proven by an ablation experiment conducted in Section 4.7, the presence of context information in the model's input has a significant impact on its performance. By removing that context information from the input, the achieved BLEU score dropped by 10.375 points, clearly indicating that the model is able to make sense of the provided context information and leverages it resulting in better performance.

5.1 Future Work

With the limited research conducted towards machine learning driven test completion and generation, there is a lot to explore for future work. The results yielded by the approach presented in this thesis indicate that it offers considerable potential to build on, but also noteworthy shortcomings in need of improvement.

The biggest disadvantage of the model is its before mentioned inability to perform as well at predicting *Then* sections for new projects, as it does for projects it has already collected knowledge on. To circumvent this shortcoming, future work could use transfer learning to leverage the model's ability to apply the information it has gathered about a previously encountered project on new predictions for that project.

In this case the model would be pretrained on a dataset like that introduced by this thesis. To then apply this model to a given project, the model would be finetuned on a few manually created tests for methods in that project. This could give the model the insights it needs to show similar performance on new projects. Adding another self-supervised pretraining step on the project's codebase before the finetuning [30], could potentially even further improve the model's ability to adjust to a specific project.

While adding context information to the model's input significantly improved its performance, that context is limited to method declarations in the same file as the test or target method. This limitation was set to keep the size of the model's input sequences in check, as the maximum input size of the Transformer architecture is heavily constrained by video memory. However, more recent research has introduced descendants of the Transformer like the Transformer-XL [91] or Reformer [92], which are less restricted in the size of their input sequences. Using such an architecture could allow for adding more complex information to the context, potentially further increasing the model's performance.

A interesting next step towards test generation would be the integration of the approach proposed by this thesis with a heuristic solution like *EvoSuite*. Although *EvoSuite* has shown great ability to achieve good code coverage [57], the studies in [9, 10] have revealed that it frequently fails at generating meaningful assert statements, and therefore *Then* sections. While this is the exact task this work's model was trained for, it is not able to generate full test suites like *EvoSuite* is. Therefore, using a heuristic solution in conjunction with a machine learning approach could potentially combine the best of both worlds.

However, with [30] being published in parallel to this thesis, a first successful attempt at machine learning driven test generation has been presented very recently. The success of employing transfer learning in their approach shows the potential of the future work suggestion made at the beginning of this section. Also, the improvements made in this work by employing an AST encoding and augmenting the model's input with context information could potentially also be applied to their approach. Most notably, the authors stated that during a qualitative evaluation they observed many of the falsely predictions being due to the model not having relevant context information at hand, as it only gets the target method as an input [30]. This indicates that future work could improve upon this approach by adding additional context information to the model's input, as proposed by this thesis.

List of Figures

List of Listings

List of Tables

Bibliography

[1] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[2] Donald Bradley Roberts and Ralph Johnson. *Practical analysis for refactoring*. University of Illinois at Urbana-Champaign, 1999.

[3] Stack overflow developer survey 2019. `https://insights.stackoverflow.com/survey/2019#work-_-unit-tests`. Accessed: 2021-01-10.

[4] José Campos, Annibale Panichella, and Gordon Fraser. Evosuite at the sbst 2019 tool competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32. IEEE, 2019.

[5] Eugenia Díaz, Javier Tuya, and Raquel Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 310–313. IEEE, 2003.

[6] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[7] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 2005.

[8] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 2007.

[9] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults

in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[10] Sina Shamshiri. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 1038–1041, 2015.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30:5998–6008, 2017.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27:3104–3112, 2014.

[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[15] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[17] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[18] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[19] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry,

Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[20] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[21] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[22] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[23] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. *arXiv*, pages arXiv–1910, 2019.

[24] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553. IEEE, 2018.

[25] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[26] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.

[27] Arseny Zorin and Vladimir Itsykson. Recurrent neural network for code clone detection. *Software Engineering and Infomation Management*, 47, 2018.

[28] Laurence Saes. Unit test generation using machine learning. Master's thesis, Universiteit van Amsterdam, 2018.

[29] Robert White and Jens Krinke. Testnmt: function-to-test neural machine translation. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, pages 30–33, 2018.

[30] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *arXiv preprint arXiv:2009.05617*, 2020.

[31] Mike Cohn. *Succeeding with Agile*. Pearson Education Limited, 2009.

[32] Mike Wacker. Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html. Accessed: 2020-12-11.

[33] Ralf Westphal. *Messaging As a Programming Model: Doing OOP As If You Meant It*. CreateSpace Independent Publishing Platform, 2013.

[34] Overview of microsoft intellitest. https://dannorth.net/introducing-bdd/. Accessed: 2021-01-10.

[35] Jeff Grigg. Arrange act assert. http://wiki.c2.com/?ArrangeActAssert. Accessed: 2020-12-12.

[36] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[37] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 858–868. IEEE, 2015.

[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[39] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[40] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[41] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[43] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[44] Ankur P Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*, 2016.

[45] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*, 2017.

[46] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

[47] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[48] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[50] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[51] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[52] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[53] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[54] How to automated junit generation with agitar one. `http://www.agitar.com/solutions/products/automated_junit_generation.html`. Accessed: 2021-01-10.

[55] Most popular programming languages. `https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/`. Accessed: 2020-11-09.

[56] Fred Glover. Future paths for integer programming and links to ar tifi cial intelli g en ce. *Computers operations research*, 13(5):533–549, 1986.

[57] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. Java unit testing tool competition-seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 15–20. IEEE, 2019.

[58] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[59] Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151, 2016.

[60] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[61] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *arXiv preprint arXiv:1710.11054*, 2017.

[62] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439. IEEE, 2016.

[63] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[64] Uday Kusupati and Venkata Ravi Teja Ailavarapu. Natural language to code using transformers.

[65] Wenhao Zheng, Hong-Yu Zhou, Ming Li, and Jianxin Wu. Code attention: Translating code to comments by exploiting domain features. *arXiv preprint arXiv:1709.07642*, 2017.

[66] Github api rate limits. `https://docs.github.com/en/free-pro-team@latest/developers/apps/rate-limits-for-github-apps`. Accessed: 2020-11-07.

[67] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[68] What are the most popular libraries java developers use? based on github's top projects. `https://blogs.oracle.com/java/top-java-libraries-on-github`. Accessed: 2021-01-10.

[69] Github 100 million repositories. https://github.blog/2018-11-08-100m-repos. Accessed: 2020-11-07.

[70] Github bigquery dataset. https://www.kaggle.com/github/github-repos. Accessed: 2020-11-07.

[71] Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, pages 202–203, 2015.

[72] One assertion per test. https://www.artima.com/weblogs/viewpost.jsp?thread=35578. Accessed: 2020-11-29.

[73] Oracle Corp. Java reflections. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection. Accessed: 2021-01-15.

[74] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[75] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*, 2020.

[76] Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems*, pages 12081–12091, 2019.

[77] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2019.

[78] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

[79] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*, 2016.

[80] Pytorch cuda support. https://pytorch.org/docs/stable/cuda.html. Accessed: 2020-12-18.

[81] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[82] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24:2546–2554, 2011.

[83] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[84] Nikhil Buduma and Nicholas Locascio. *Fundamentals of deep learning: Designing next-generation machine intelligence algorithms*. O'Reilly Media, Inc., 2017.

[85] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[86] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[87] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[88] Wojciech Kryściński, Nitish Shirish Keskar, Bryan McCann, Caiming Xiong, and Richard Socher. Neural text summarization: A critical evaluation. *arXiv preprint arXiv:1908.08960*, 2019.

[89] Ruiqiang Zhang, Genichiro Kikui, Hirofumi Yamamoto, Frank K Soong, Taro Watanabe, and Wai-Kit Lo. A unified approach in speech-to-speech translation: integrating features of speech recognition and machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 1168–1174, 2004.

[90] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[91] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[92] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Appendices

A Effects of Label Smoothing

Metric	Main model (with label smoothing)	Main model (without label smoothing)
Unparsable rate	**1.57%**	1.91%
Unknown token generated rate	**0.21%**	0.27%
Max length exceeded rate	17.60%	**9.41%**
BLEU	30.179	**38.995**
$BLEU_1$	51.003	**57.463**
$BLEU_2$	35.003	**43.234**
$BLEU_3$	24.619	**33.575**
$BLEU_4$	18.873	**27.723**
ROUGE-L	64.027	**69.437**
ROUGE-L_P	67.144	**72.633**
ROUGE-L_F	65.548	**70.999**
ROUGE-1	65.616	**70.885**
ROUGE-1_P	68.944	**74.308**
ROUGE-1_F	67.238	**72.556**
ROUGE-2	47.963	**55.858**
ROUGE-2_P	49.582	**57.972**
ROUGE-2_F	48.759	**56.895**
adj(BLEU)	29.641	**38.145**
adj($BLEU_1$)	50.095	**56.210**
adj($BLEU_2$)	34.380	**42.291**
adj($BLEU_3$)	24.180	**32.843**
adj($BLEU_4$)	18.537	**27.118**
adj(ROUGE-L)	62.887	**67.923**
adj(ROUGE-L_P)	65.949	**71.050**
adj(ROUGE-L_F)	64.382	**69.452**
adj(ROUGE-1)	64.448	**69.340**
adj(ROUGE-1_P)	67.716	**72.688**
adj(ROUGE-1_F)	66.042	**70.975**
adj(ROUGE-2)	47.110	**54.640**
adj(ROUGE-2_P)	48.700	**56.708**
adj(ROUGE-2_F)	47.892	**55.655**

Table 5.1: The evaluation results of the main model trained using label smoothing and without. As these results show, label smoothing decreases the model's performance.