ML4SE: Learning How to Mutate Python Source Code from Bug-Fixes

Héctor Bállega
Delft University of Technology
H.BallegaFernandez@student.tudelft.nl

Fabian Nonnenmacher
Delft University of Technology
F.M.Nonnenmacher@student.tudelft.nl

Ruiyang Ding
Delft University of Technology
r.ding@student.tudelft.nl

ABSTRACT

This paper describes the results of the project done in the class "Machine Learning for Software Engineering" of TU Delft in 2019. The goal was to replicate an existing paper to a similar problem. This project focuses on the topic of learning Python code mutants for mutation testing and takes [9] as a blueprint, which has researched the same topic but for Java code.

1 INTRODUCTION

Mutation testing is a testing technique which indicates the quality of the tests. During mutation testing the source code is mutated, so that it contains a bug and then the tests cases are checked if they can detect this bug [5]. It is easy to mutate the code to a buggy version of it, but for helpful mutation testing the mutations should be as similar as possible to real-world bugs. Because then they can guide the design of the testing system and can measure their effectiveness.[7]

However, how to generate good mutates can be challenging. The mutates generation by traditional methods takes too many efforts and the result could be too specific to represents the whole project.[9] So, the paper [9] proposed to using Machine Learning method to learn how to generate a good mutate from the bug-fixing commits across the whole GitHub. But, the paper only designed and implemented the method for Java which is not helpful for the developers who mainly write other languages, like Python.

So, in this project, we try to replicate the work done in [9] but for generating Python code mutants. This includes the analysis of 2.2 million bug-fix commits on GitHub, the preprocessing of the extracted source code to generate the datasets for the model and, finally, the design and training of a Seq2Seq model to automatically generate mutants of Python code methods.

2 DATA MINING AND PROCESSING

As stated in the Introduction, our main challenges in the project were to collect and preprocess the data correctly to be able to train a machine learning model. In [9] Tufano et al. describe which steps they have applied to Java code for getting the needed data. During our project, these steps have also been applied to Python code. Because of the different characteristics of Python and Java (e.g. dynamic vs static) not all steps are applicable in the same way. Therefore we discuss in this chapter the different processing steps and how we have adapted them to fit for Python code. Besides this we also discuss in Subsection 2.6 the tools and setup we have chosen to be able to process such a massive amount of data.

2.1 Identifying Relevant Commits

We have explored the GitHub public datasets available in google BigQuery, which contains more than 2.9M public, open-source licensed repositories. Firstly, we tried to follow the same approach proposed in [9] where they mine the commits from the events stored in the GitHub Archive. This approach was not the most suitable for us because we couldn't determine the programming language from the events and getting it using the GitHub API would have taken us around 80 days because of the API limits

Similar to the paper [9] we selected those commits having the following pattern in their message: (fix or solve) and (bug or issue or problem or error). Additionally, by using the GitHub public dataset instead of the GitHub Archive, we were able to filter out all commits, not in a 'python' labelled repository. However this label does not necessarily relate to the files modified in the commit, therefore another filtering step by using the GitHub API was necessary.

For being able to query the API and to download the files, we exported several attributes of the commit: the *commit id*, the *parent's commit id*, the *commit message* and the *name of the repository* the commit belongs to.

The 'bug fix' commits and its attributes were exported to a CSV file. During these steps, in total 2248906 commits were extracted from the years 2015 until October 2010 (data which was already available on the GitHub dataset).

2.2 Download Commits from GitHub

As explained in the previous section, it is not known which data/files the extracted commits have modified. To overcome this problem of the dataset, we have used for the BigQuery operation, the GitHub API has to be requested for the additional information.

By using the attributes exported in the previous steps, the GitHub API[1] request could be composed easily, because it follows this format:

```
GET /repos/:owner/:repo/commits
```

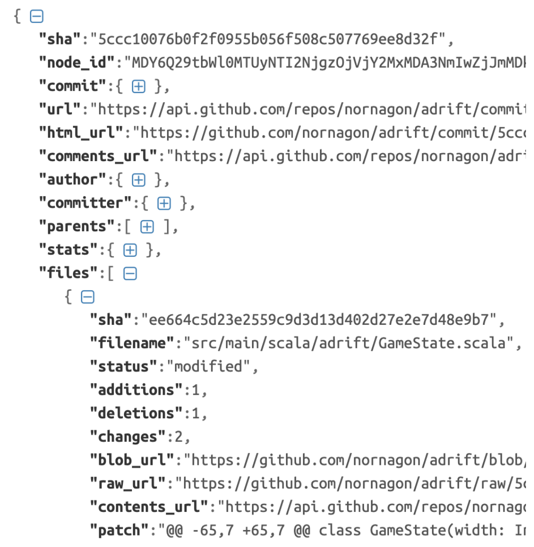
Part of example response content can be seen in Figure 1. From this API, we can get the list of changed files, and their path with in the repository. With this list of files, the repository name and the id of the commit, we can compose the path to all RAW files on GitHub. By composing the path for the bugfix commit and the parent commit, the content of the files before and after the commit can been downloaded.

Figure 1 gives an overview of other attributes provided by the GitHub API. These attributes were used to limit the commits further, so that only valuable commits were downloaded. Similar to [9], all files which are deleted or newly created in the commit are discarded, because it's not possible to extract a before and after

[1]https://developer.github.com/v3/repos/commits/

version from them. Furthermore, all commits which modify more than five python files are ignored, because similar as [9] we want to focus on bug fix modifying only a few files and are not spread across the system.

During this step we reduced the commits to 565140, we and downloaded all the source code pairs before and after the commit. We stored all of the source code (in total 41.4 GB) on a Amazon S3 storage in a structure as shown on Figure 2.



```
{ ⊟
   "sha":"5ccc10076b0f2f0955b056f508c507769ee8d32f",
   "node_id":"MDY6Q29tbWl0MTUyNTI2NjgzOjVjY2MxMDA3NmIwZjJmMDl
   "commit":{ ⊞ },
   "url":"https://api.github.com/repos/nornagon/adrift/commit
   "html_url":"https://github.com/nornagon/adrift/commit/5ccc
   "comments_url":"https://api.github.com/repos/nornagon/adr
   "author":{ ⊞ },
   "committer":{ ⊞ },
   "parents":[ ⊞ ],
   "stats":{ ⊞ },
   "files":[ ⊟
      { ⊟
         "sha":"ee664c5d23e2559c9d3d13d402d27e2e7d48e9b7",
         "filename":"src/main/scala/adrift/GameState.scala",
         "status":"modified",
         "additions":1,
         "deletions":1,
         "changes":2,
         "blob_url":"https://github.com/nornagon/adrift/blob,
         "raw_url":"https://github.com/nornagon/adrift/raw/5c
         "contents_url":"https://api.github.com/repos/nornago
         "patch":"@@ -65,7 +65,7 @@ class GameState(width: I
```

Figure 1: Part of JSON response of GitHub commit API

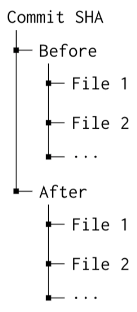


Figure 2: Directory structure of commit files

2.3 Extract Method Pairs

From the previous step we received the code before and after a bug-fix commit. In [9] these are the so called Transformation pairs (m_b, m_f). In which m_b represents the buggy code and m_f the corresponding fixed code. The basic idea is to use these pairs to train the RNN so that the model learns the transformation from the fixed code (m_f) to the buggy code (m_b). To simply the modifications in one commit the idea is to extract modified method pairs. In [9] they argue, that methods are a reasonable target for mutation, because they are small enough for to easily train a model, most likely they

some operations from the paper	similiar operations created by us
Insert VariableRead at Invocation	Insert NameLoad at For
Update VariableRead at Invocation	Update attr at AttributeLoad
Move Assignment from Block to CtAssignmentImpl	Move Expr at body
Delete Literal at Invocation	Delete Str at ListLoad

Figure 3: Comparison of action formatting

implement a single task and still they are providing enough context for learning mutations.

To extract the modified methods they use the tool Gumtree Spoon AST Diff [4] which is unfortunately only applicable for Java code. However this tool is based on Gumtree Diff[3] which basically provides the same functionality for multiple programming languages but with a bit more complicated Interface. While using this tool we figures out, that it is not very robust (e.g. throws unexplainable Nullpointer exceptions) and contains a few bugs(e.g. references to parent node are not correct). Due to lack of alternatives, we still decided to use it and try to work around the issues. However at the end we decided to just exclude method pairs, when they couldn't be processed by Gumtree without investigating all problems further.

Based Gumtree we wrote a Java program, which takes two python files (m_b, m_f) as arguments and returns the modified method names and the related edit actions A (sequence of edit actions that transforms the buggy method into the fixed method). Then this outcome was used to extract the code of the methods by using Python AST[2] and Python AST Unparser[3] . Another more practical challenge, was to call the Java program from Python our main language. We explain our approach in Subsubsection 2.6.5.

Similar to the approach in [9] we only consider modified methods (at least one edit action) and methods which can be parsed by Gumtree and Python AST. With this approach we effectively reduce the methods to Python 3 compatible methods, because Python AST only supports Python 3. Beside this, we are not considering deleted or newly created methods.

Unfortunately neither Gumtree nor the paper [9] are specifying the important information of the action and the structure of the action String. However this is important, because this information will be later on used to cluster the method pairs into different groups of bugs (see Section 3, Therefore we downloaded the *Extracted Bug-Fix Pairs* used in the paper[4] and tried to format our actions in a similar way, the result can be seen in Figure 3.

Within the data from 2015 to 2019 we extracted in total XXX method pairs. For every extracted method pair we stored 3 files: `before.py` (the code of the method before the commit), `after.py` (the code of the method after the commit) and `operations.txt` (list of edit actions).

2.4 Identify idioms

Before reducing the extreme large vocabulary by replacing identifiers and literals with a common token, the idioms have to be identified. In the paper [9] the terms "idioms" refers to often used literals and identifiers, which can almost be treated as keywords of the language and therefore must not be replaced.

To identify the idioms, in [9] they analyse the frequency of used literals and identifiers in a randomly sampled subset. Because the

[2]https://docs.python.org/3/library/ast.html
[3]https://astunparse.readthedocs.io
[4]data is available on paper's website: https://sites.google.com/view/learning-mutation

idioms were necessary for continuing with the following steps, we were not able to wait until all method pairs were extracted. Therefore, we couldn't work here methodologically and so we just analysed the method pairs extracted from 2019, instead of a random sampled subset. However we assume, that this does not make a big difference, because also the number of idioms used is not well justified.

To identify all used identifiers and literals, we implemented a visitor which counts all identifiers and literals used in a tree received after parsing the code with Python AST.

As described above we analysed all methods pairs from 2019, which are in total 40265 method pairs. These pairs contain in total 438563 different literals and idioms. From this we extracted the 272 most used idioms to get the same absolute number of idioms as the authors of the papers. The most 10 used identifiers and literals can be seen in Table 1. It might be noticeable, that different to [9] we are not distinguishing between a variable and a method, we will discuss the reasons for this in the following subsection.

Count	Identifier/Literal	Type
136127	0	Literal
121171	1	Literal
43971	append	Identifier
43458	format	Identifier
39920	len	Identifier
37660	get	Identifier
36200	2	Literal
35705	i	Identifier
32393	str	Identifier
28175	print	Identifier

Table 1: 10 most used Identifiers and Literals

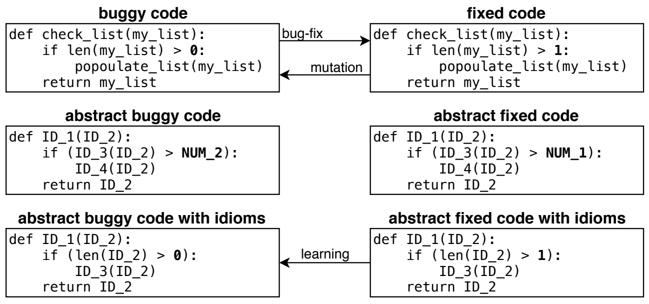
2.5 Create an Abstract Code Representation

As already mentioned, the goal of the next step is to replace all identifiers and literals except the idioms with index IDs. To keep the semantics of the code for different types of literals or method different IDs are used.

Tufano et al. have published their tool *src2abs*, which they used for creating their abstract code representation on github[5]. Unfortunately this tool can only handle Java Code. Because we didn't find a similar tool for python we had to implement our own version. As before, for extracting the idioms, we implemented a visitor which visits the AST tree generated by Python AST and replaces the name of the nodes with indexed IDs.

In paper [9] the authors are considering 7 different types of IDs. METHOD_X is used for replacing method names, VAR_X for variables, Type_X for classes and STRING_X, CHAR_X, INT_X, FLOAT_X for the related literals. First of all, that python does not distinguish between integer and float literals. Consequently we replacing all number literals with the ID NUM_X. The same for chars and strings, so we use STR_X for both. Different to java in python functions are first class citizens, which means that they can be used as objects and can be for example assigned to variables [8]. As a result functions and other variables cannot easily distinguished

[5]https://github.com/micheletufano/src2abs



```
def check_list(my_list):
    if len(my_list) > 0:
        popoulate_list(my_list)
    return my_list
```

```
def check_list(my_list):
    if len(my_list) > 1:
        popoulate_list(my_list)
    return my_list
```

```
def ID_1(ID_2):
    if (ID_3(ID_2) > NUM_2):
        ID_4(ID_2)
    return ID_2
```

```
def ID_1(ID_2):
    if (ID_3(ID_2) > NUM_1):
        ID_4(ID_2)
    return ID_2
```

```
def ID_1(ID_2):
    if (len(ID_2) > 0):
        ID_3(ID_2)
    return ID_2
```

```
def ID_1(ID_2):
    if (len(ID_2) > 1):
        ID_3(ID_2)
    return ID_2
```

Figure 4: Transformation pair Example

in the AST and therefore we decided to use the same ID type (ID_X for both. Furthermore, because python is a dynamic language, types are barely used. Therefore, we chose the easiest way to handle types and we just replace them with an ID_X token, too. So in total we replace all literals and identifier with 3 types of IDs: ID_X , NUM_X and STR_X.

Figure 4 summarises how the transformation works. As it can be seen, in both code snippets, the same literals and identifier are replaced with the same IDs. (This figure is actually a python version of Fig. 1 in [9] and therefor also shows how our abstract code representation differs from the representation in [9].)

In contrast to Java python supports many syntactical sugar for example named and default parameters. Even when these functionality is only used in a few methods, we had to make sutre, that our code abstraction generator correctly handles them.

Because we want to focus on small methods similar to [9], we also follow their guidelines to filter the method pairs. We are only considering methods for which the transformation contains less then 100 actions and small methods whose code consists of less than 50 tokens. Furthermore Tufano et al. are splitting the pairs into 2 sets: TP_{ident} and $TP_{ident-lit}$. The former set TP_{ident} allows buggy methods which contain different literals then the fixed method. The latter set $TP_{ident-lit}$ only contains method pairs where the buggy method does not contain any new IDs. Neither literals nor identifiers. For simplifying our project we just focus on the set $TP_{ident-lit}$ and are sorting out all methods pairs where the buggy method contains IDs (ID_X , NUM_X and STR_X) which are not contained in the fixed method. This will make training the model easier, because no new IDs have to be generated.

2.6 Tools and Setup

2.6.1 *Challenges.*

Besides the challenge to adapt the data processing pipeline from [9] to python, we were also facing technical challenges introduced by the amount of data:

- **The total size of data**
  It's not easy to download and store such huge amount of data with personal computer and low bandwidth network.
- **GitHub API has a rate limitation**
  It is limited to 5000 API requests/hour per user.
- **Amount of computational resources**
  Before getting the final dataset that can be feed to the model, the course code must go through multiple process. These

process will take considerably amount of time, as well as computational resources.

- **All computation tasks are long running and might fail** Especially the first time our programs run against the real world, they are not very robust. Therefore we need short feedback cycles and strategies how to handle failure without having to reprocess the already processed data.

2.6.2 *Monthly Chunks.*

The first step to get the control over the huge amount of data was to put all data of the same month together into one chunk. We extended the initial BigQuery-SQL statement, so that all commits off the same month were stored together in one CSV file.

This allowed us, to process different months independently from each other. First of all we were able to test later processing steps on smaller data sets and in case of identified bugs, we could easily rerun them on the monthly subset. Further more it also allowed us to process multiple chunks in parallel just by executing the same script multiple times against different months without having the need to write our python programs in a parallel way.

2.6.3 *Multiple GitHub Tokens.*

As described in the previous section splitting the data into chunks of month allows us to process month in parallel. If we run the extractors in parallel anyway, it is easy to use a different GitHub Token for every extractor. By asking our friends, we were in control of 12 Tokens and were so able to download all month of a year in parallel. Because an average month contains approximately 40000 commit we were able to download a full year over night.

2.6.4 *Cloud & Container.*

We use AWS, Docker and multiple tokens to solve our challenges.

We containerlize the each part and push the images to DockerHub to allow us to easily run our programs on AWS cloud. When run the container, the user can set the range of months he'd like to mine and GitHub token by passing arguments to CMD. So it is easy for us to parallelized the each steps by the unit of one month. Specially to commit mining step, it can be also used for the negative effect from API rate limitation by easily set different tokens for different containers.

Initially we used Amazon Elastic Compute Cloud(EC2) instance and Docker Compose to set up multiple containers at same virtual machine. But the limited performance of EC2 instance slowed down the whole process. So Instead of EC2, we leveraged Amazon Elastic Container Service (ECS). The Fargate mode of ECS provides us the higher computation power with lease configure effort. With this setup, we can easily run multiple containers to process data from different months at same time, without competing resources between each others.

With this setup, we can get the commit file of each month in around 6 hours, and finish the method extraction for each month in around 3 hours.(Various depends on size of data from each months)

2.6.5 Calling Java from Python. As described above (Section 2.3, for extracting the method pairs we had to call Gumtree a Java Program from our python data mining program. The first naive approach was to create a java command line tool, which we called from python as a subprocess [6] and passed the method pair as parameters. As a result a new JVM was started for every method pair we had to analyse which made the processing extremely slow.

Therefore we decided to change the approach and we implemented the java program as an RPC server by using gRPC[7]. This change, allowed us to just start the JVM once at the beginning of the processing and to communicate afterwards with the running java server by using RPC. This change reduced the processing of one month from approximately 12 hours to under 3 hours.

3 DATA CLUSTERING

In order to generate good quality mutants, we are going to cluster our method pairs according by the similarity of their list of AST actions. We believe that by clustering the method pairs and training a model specifically for each cluster, we can improve its performance. For each method-pair, we take its list of AST actions and compute a fixed-size vector representation following a *word2vec* approach [6]. After computing the vector representation for each list of actions we consider that the closer two vectors are, the more similar its list of AST actions are. To this goal, we have used k-means to cluster the vectors. One of the challenges that we faced, as similarly stated in [9], regards into choosing k (the number of clusters) so that we can train different mutation models but we still have enough training examples to make the model infer the patterns. Similar to the paper we run the k-mean algorithm multiple times with different seeds and were evaluating the quality of the cluster based on the Silhouette Score. We tried cluster sizes form 3 to 6 and it has been shown, that the Silhoutte Score of k=4 and k=5 are often very similar. Therefore we decided for a cluster size of 4, because then the datasets per cluster are bigger. Finally, we divided the dataset $TP_{ident-lit}$ into the clusters: C_0, C_1, C_2, C_3. The sizes of the clusters can be seen on Table 2

Cluster	Number of elements
C_0	3237
C_1	4694
C_2	5949
C_3	2402

Table 2: Sizes of clusters

4 MODEL TRAINING

After having the method pairs assigned to different clusters, we assume that every cluster contains a similar type of bug. And therefore the pairs of one cluster can now be used to train an RNN model, which should learn to mutate a correct method into a bug of this specific time. During the project only a small prototype has been created, which demonstrated general functionality, but which must be developed further to be compared with the results of [9] In the following sections this prototype is introduced before.

4.1 Dataset Preparation

In [9] the authors directly feed the abstract code representation (Subsection 2.5) into the model by splitting all tokens. However,

[6]https://docs.python.org/3/library/subprocess.html
[7]https://grpc.io/

compared to Python, Java code has one advantage, the white-space characters (e.g. newlines and indent) are not having any meaning. So the model does not need to know about this. However for python this is important. Therefore we decided to define the beginning of a new block with the token 'BLOCK_START' and the end of a block with the token 'BLOCK_END'. Additionally a 'START' and 'END' token has been added. The newline chars were not removed, because they mark the end of a statement and have therefore a similar meaning as the semicolon ';' in Java. An example transformation can be seen on Figure 5.

After creating the token streams, all method pairs of the same cluster were separated into 3 random sets: the train set (80%), the validation set (10%) and the test set (10%) and were written into separate files, so that they could be processed easily by RNN model.

When feeding the data into the model we also realised, that we have not created an abstract representation for tokens. For example we missed inline import statements. Therefore we also took this processing step as an additional filter step and filtered out all pairs with forgotten tokens to replace. Unfortunately even then we realised that our model were not able to process tokens newly introduced in the training set.For this reason, we filtered these samples from the training set.

4.2 Sequence-to-Sequence Model

In this work, we apply a Sequence-to-Sequence (Seq2Seq) model to learn how to generate buggy code from a fixed one. The model consists of three components, a Encoder, a Decoder and an Attention layer. Both Encoder and Decoder are Gated Recurrent Units (GRU). The decision to to choose GRU instead of LSTM is due to the better results reported in [9] for the task of generating mutants from Java source code. Besides, GRUs are simpler and easier to modify because they have less parameters and unlike LSTM, the memory is completely exposed. We have based our implementation in [1] and [2], but due to time restrictions we were not able to configure them in a multilayer bidirectional approach as suggested by [9]. The Encoder and the Decoder were both using 2 hidden layers with a dimension of 512. For dealing with the variable input length the bucketing and padding mechanism provided by PyTorch was used.

5 RESULTS

In this section, we evaluate our models by measuring how close the buggy source code generated by the model is compared with the target one. With this results, we want to focus on answering the following research questions:

- **RQ1:** How effective is our Seq2Seq model to generate good mutants?
- **RQ2:** What challenges are involved into generating mutants for a non-typed programming language?

5.1 RQ1: BLEU Score

We evaluate the result by comparing the target data (buggy code) with our model prediction. We have calculated the BLEU score for each cluster by comparing how much differs, the output from the model with expected one. The results can be seen one Figure 6. As we can see all the datasets present a similar similar result, being the dataset built using the Cluster 1 the one who achieves the highest score (19.75). The results are not so good as compared with the reference paper [9], where they achieve a score around 70%. Our implementation tried to keep the model very simple and also we couldn't follow exactly the same approach by using a multilayer bidirectional GRU as proposed. Apart from that, we think that inferring the patterns in Python code is much more difficult than Java due to the ambiguity on its semantic.

Dataset	BLEU (%)
Cluster 0	18.64
Cluster 1	19.75
Cluster 2	16.36
Cluster 3	18.56

Table 3: BLEU Score of Each Dataset

5.2 RQ2: Mutants in non typed programming languages

The Figure 6 shows a transformation pair example from the test dataset of the second cluster. We can determine that the model is able to learn the general structure of the buggy code and detect where the mutation occurs (after the return statement). However, this translation is not perfect, the model has problems to generate some special idioms such as ID_5. We think that the reason why our model is not able to specify those cases compared with the reference paper is because due to a lack of a GPU we couldn't train the model for more than 1k epochs (compared with 40k in [9]).

6 IMPROVEMENT/FURTHER STEPS

As described above, the model cannot yet be seen as a fully trained and balanced model. Instead it is more a first prototype which needs a lot of adjustment.

First of all the training data must be reviewed. After mining so many commits it's kind of unsatisfying, that we only have a few thousand training pairs per cluster. It must be discussed, if all restrictions (e.g. only 50 tokens per method) are really useful in the context of python.

Afterwards the clusters have to be analysed further - are really similar bugs in one cluster? The extremely different BLEU scores per cluster are indicating that the clusters are not yet very homogeneous

And then obviously the seq2seq model must be tested with different configurations. Here it makes sense to orientate at the paper [9] first, but then also to try out other encoder and decoders.

Last but not least with all the restrictions data is filtered, the bug generation is only applicable for a subset of real world methods. The biggest limitation is probably the size of 50 tokens, which make sit only applicable for very small methods.

Concluding the work, we have shown in this report, that we can gather python functions in a similar way as Tufano et al. has gathered java methods. With all this preprocessing, we can generate a similar stream of tokens. It has been shown that this data is sufficient to train a first prototype which generates python mutates. Based on our data, further work might investigate if the model can be improved, so that it generates similar results as the model in [9]. Furthermore the code provided by us, might be used as a basis for implementing code abstracting tools like src2abs[8] but for python.

[8]https://github.com/micheletufano/src2abs

abstract code representation:

```
def ID_1 (ID_2, ID_3):
    if ID_2 > ID_3:
        return ID_2
    return ID_3
```

stream of tokens:

```
START def ID_1 ( ID_2 , ID_3 ) : \n BLOCK_BEGIN if ID_2 > ID_3 : \n BLOCK_BEGIN return ID_2 \n return ID_2 \n BLOCK_END BLOCK_END END
```

Figure 5: Transformation from abstract code representation to stream of tokens

FIXED:

```
START @ ID_5 def ID_1 ( self ) : \n BLOCK_BEGIN return ( self . type == ID_4 . ID_3 ) \n BLOCK_END END
```

BUGGY:

```
START @ ID_5 def ID_1 ( , ) : \n BLOCK_BEGIN ID_4 ( , . ID_2 { return . ID_3 ) BLOCK_END END
```

PREDICTED:

```
START def ID_1 ( self ) : \n BLOCK_BEGIN return ( ( ( ( ) \n BLOCK_END END
```

Figure 6: Transformation pair Example

7 REFLECTION

After reading this report, the weak point of our result/project becomes obvious. Instead being able to provide a machine learning model, which can be compared with the model in the paper we were just able to get a basic Seq2Seq model with very limited functionality. We show how the data must be gathered, so that it fits to the training data described in [9]. However at the end we were running short on time and were not even able to implement the model proposed in the paper. What was the problem?

Also here the answer is time management - as in most projects. But the problem was not directly the amount of time we invested. It was more related that we underestimated the data processing and that we were running in a dead end at the beginning.

As described in section 2 we were following firstly the approach of the paper and were trying to extract all commits from the GitHub Archive. Unfortunately this was resulting in a dead end. We realised at some point that this is not suitable for us and stop all the progress we made with the GitHub Archive and were investigating alternatives. However stopping was probably the worst decision we made, because no everything else was blocked. And we had to wait for the new approach and the evaluation of available datasets. In retro perspective it would have been definitely better to start working with a small set of data instead and use this as basis for making all scripts running and robust.

The next big issue was, that we were not aware of the complexity of the data processing pipeline. The processing steps were quite extensive and not that easy. It involved the usage of many different technologies e.g. BigQuery, GitHub API, Java which just takes time to get familiar with. Furthermore many tools which the authors used for processing in their pipeline were not suitable for python. This means we had to reimplement those tools by our own. Admittedly, the python AST package was very helpful when implementing this processing steps, nevertheless it also took time to understand it completely.

And the tools which were available also for Python mainly gumTree was in a very bad state. The documentation was not very good. Especially, because the version on github differed extremely from the jar-files published on maven central. Furthermore this tool was kind of buggy and therefore we had to experiment a lot and build workarounds.

The next big thing was the amount of data, we discuss how we solved the challenges related to this in Section 2.6. To make it short, we executed our programs as docker containers on AWS. This worked quite well, but added again a new layer of complexity. Another problem related with this amount of data (monthly chunks were still big) were the feedback cycles. Often we started a program over night and on the next day we found out, that it has stopped in the middle of the night, because we have overseen a corner case.

In conclusion we were facing many challenges during the data processing and obviously we have not chosen the best strategies for tackling these challenges right from the beginning. Now looking back, we should have tried to get some data as fast as possible even in bad quality, so that we could have started with training the model earlier. Due to our lack of experience in Machine Learning, we had a lot difficulties on building a very complex model, Therefore we were just able make a simple Seq2Seq model which it is not perfect but proves that we understood the problem. We were not able to evaluate our results with the results of the paper, but also we knew that it would be hard due to the many differences presented between Java and Python code.

But besides all the challenges, we actually had a lot of fun during this project and we learned a lot about python and many tools out there. Especially after mining the data from GitHub, which took us more than and a month and delayed the programming of the Seq2Seq model to the last weeks. But nevertheless, we worked with a super high focus and actually managed to get a lot of things done.

REFERENCES

[1] [n.d.]. bentrevett/pytorch-seq2seq: Tutorials on implementing a few sequence-to-sequence (seq2seq) models with PyTorch and TorchText. [IN PROGRESS]. https://github.com/bentrevett/pytorch-seq2seq. (Accessed on 10/25/2019).

[2] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[3] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[4] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*. Västeras, Sweden, 313–324. https://doi.org/10.1145/2642937.2642982 update for oadoi on Nov 02 2018.

[5] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

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

[7] Samar Mouchawrab, Lionel C Briand, Yvan Labiche, and Massimiliano Di Penta. 2010. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Transactions on Software Engineering* 37, 2 (2010), 161–187.

[8] Durga Swaroop Perla. [n.d.]. Functions as First class citizens in Python. https://medium.com/@durgaswaroop/functions-as-first-class-citizens-in-python-2e70cdc6e357. (Accessed on 10/23/2019).

[9] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Learning How to Mutate Source Code from Bug-Fixes. *CoRR* abs/1812.10772 (2018). arXiv:1812.10772 http://arxiv.org/abs/1812.10772