information

Article

Automatic Acquisition of Annotated Training Corpora for Test-Code Generation

Magdalena Kacmajor [1,*] and John D. Kelleher [2]

1 Innovation Exchange, IBM Ireland, Dublin 4, Ireland
2 ADAPT Centre & ICE Research Institute, Technological University Dublin, Dublin 2, D08 X622, Ireland; john.d.kelleher@dit.ie
* Correspondence: magdalena.kacmajor@ie.ibm.com

Received: 21 January 2019; Accepted: 13 February 2019; Published: 17 February 2019

Abstract: Open software repositories make large amounts of source code publicly available. Potentially, this source code could be used as training data to develop new, machine learning-based programming tools. For many applications, however, raw code scraped from online repositories does not constitute an adequate training dataset. Building on the recent and rapid improvements in machine translation (MT), one possibly very interesting application is code generation from natural language descriptions. One of the bottlenecks in developing these MT-inspired systems is the acquisition of parallel text-code corpora required for training code-generative models. This paper addresses the problem of automatically synthetizing parallel text-code corpora in the software testing domain. Our approach is based on the observation that self-documentation through descriptive method names is widely adopted in test automation, in particular for unit testing. Therefore, we propose synthesizing parallel corpora comprised of parsed test function names serving as code descriptions, aligned with the corresponding function bodies. We present the results of applying one of the state-of-the-art MT methods on such a generated dataset. Our experiments show that a neural MT model trained on our dataset can generate syntactically correct and semantically relevant short Java functions from quasi-natural language descriptions of functionality.

Keywords: test automation; code generation; neural machine translation; naturalness of software; statistical semantics

1. Introduction

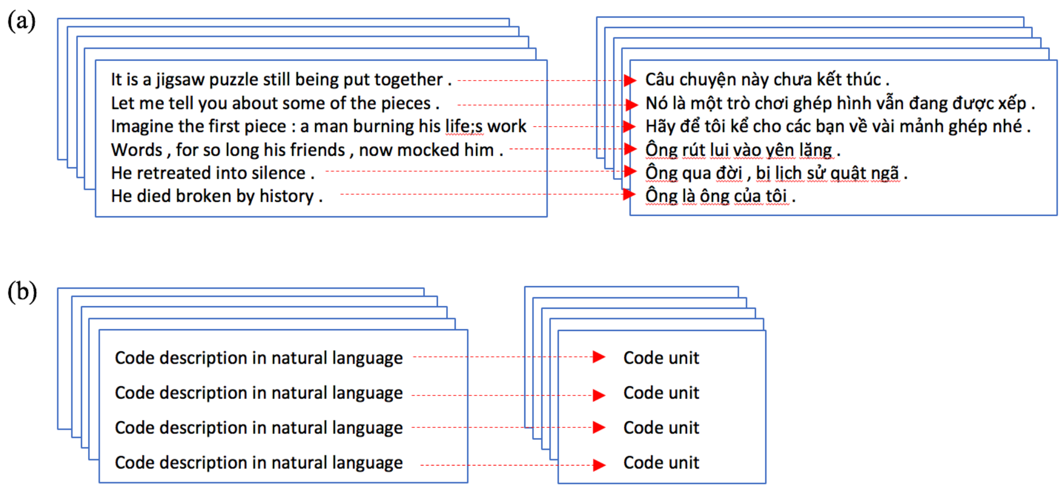
As digitization spreads into all areas of business and social life, the pressure on software development organizations is growing. The sheer amount of code being created, and the increasing complexity of software systems, fuels the need for new methods and tools to support the software development process.

A widely adopted framework addressing the challenges of the modern software delivery lifecycle is the DevOps model [1], which is founded on the principles of continuous integration, continuous delivery, and continuous testing. Both the wisdom of the crowd and academic evidence [2] speak for the efficiency of DevOps practice, but adopting DevOps brings its own challenges, including a significant increase in the volume and frequency of testing. In fact, on a large-scale project it is not feasible to implement DevOps without test automation—and writing automated test cases is time- and resource-consuming. Not surprisingly, automated test case generation methods are being actively studied. In general, to generate unit test cases, existing approaches use information extracted from other software artifacts, such as code under test, specification models, or execution logs [3].

State-of-the-art test generation tools can significantly improve test coverage; however, it has been shown that their fault detection potential is problematic: many faulty code portions are never executed, or are executed in such a way that defects are not detected [4]. These tools stem from the tradition of research on code analysis and code generation that is concerned with formal semantics and structural information about the code. Such research takes advantage of the formality, consistency, and unequivocalness of programming languages—that is, the properties that distinguish source code from natural languages. A more recent research trend switches the focus to statistical semantics. This exciting alternative can be now fully explored thanks to the much-increased availability of source-code resources stored in online open source repositories. It has been argued that large source-code corpora exhibit similar statistical properties to those of natural language corpora [5], and indeed, statistical language models developed for Natural Language Processing (NLP) have proved to be efficient when applied to programming languages [6].

However, even billions of lines of code scraped from online repositories are not sufficient to satisfy the training requirements for some types of tasks. Many applications—such as code generation from natural language descriptions, code search by natural language queries, or automated code documentation—require joint processing of natural languages and programming languages. This means that the source-code corpora used for training these systems need to be appropriately annotated with natural language descriptions. The main challenge here is the acquisition of fine-grained natural language annotations that accurately and consistently describe the semantics of code.

As a concrete example, statistical translation models used in NLP require training on parallel corpora—also called **bi-texts**—in which many sentences written in a source language are aligned with their translations in a target language. A schematic example of a parallel corpus with aligned sentences in natural languages is presented in Figure 1a. To apply statistical translation models to source code—that is, to train a model capable of mapping textual sequences to sequences of source code—it is necessary to obtain a **text-code** parallel corpus in which a large number of code units are aligned with their descriptions in natural language (Figure 1b).



(a)
It is a jigsaw puzzle still being put together .
Let me tell you about some of the pieces .
Imagine the first piece : a man burning his life;s work
Words , for so long his friends , now mocked him .
He retreated into silence .
He died broken by history .
Câu chuyện này chưa kết thúc .
Nó là một trò chơi ghép hình vẫn đang được xếp .
Hãy để tôi kể cho các bạn về vài mảnh ghép nhé .
Ông rút lui vào yên lặng .
Ông qua đời , bị lịch sử quật ngã .
Ông là ông của tôi .
(b)
Code description in natural language
Code description in natural language
Code description in natural language
Code description in natural language
Code unit
Code unit
Code unit
Code unit

Figure 1. Schematic examples of parallel corpora: (**a**) for NLP translation (source English sentences aligned with target Vietnamese sentences); (**b**) for joint processing of natural and programming languages.

Aligned corpora for statistical machine translation (MT) of two natural languages (bi-text datasets) can be gathered from existing collections of translated texts. However, obtaining a parallel corpus for a natural language coupled with a programming language (a text-code dataset) is much less straightforward.

The question of what should be the nature and level of detail of the natural language descriptions provided in such a corpus does not have a definite answer and requires more investigation.

Nonetheless, it seems reasonable to assume that the practical value of a text-code corpora depends on the following properties:

1. Size, and the potential to scale in size in the future, is of particular importance for deep learning models which require large amounts of training data.
2. Acquisition cost, in terms of human effort and the complexity of the procedure.
3. Level of noise in the acquired data.
4. Granularity of the natural language descriptions.

Several recent studies have proposed more or less sophisticated methods of obtaining text-code corpora (see Section 2). The proposed methods vary in terms of the properties listed above, but regardless their practical value, none of them are applicable to the testing domain. The main contribution of this paper is a novel method of automatically synthetizing large text-code datasets containing textual descriptions of single testing tasks, each matched with the code implementing that task. Moreover, in this paper we demonstrate that machine learning models trained on our datasets can generate complete, compilable, and semantically relevant automated test cases based on quasi-natural language descriptions of testing tasks. These results were obtained using a neural MT model [7] designed for learning from bi-text corpora, in which the degree of equivalence between source and target languages is very high. We find that this off-the-shelf neural MT architecture performs well on our code-text corpora, which suggests that the quasi-natural language descriptions obtained using our approach are precise and consistent enough to allow direct translation to code.

There are two aspects of the potential implications of the presented work. First, from the perspective of the testing community, we present an efficient, inexpensive, and scalable method for annotating test code with textual descriptions. The availability of such annotated datasets can accelerate the application of the latest advances in machine learning to the testing domain. Second, from the perspective of research on applying statistical models to source code, our datasets may provide better insight into the desired characteristics of text and code sequences in a training corpus. Understanding what type of annotations works well or what is the optimal translation unit for the source code may be valuable for researchers concerned with synthesizing text-code datasets.

The remainder of this article is organized as follows. Section 2 provides an overview of existing solutions for text-code corpora acquisition. In Section 3 we provide the rationale of our approach and explains how it works. Section 4 describes in detail the procedure of synthesizing training corpora used in our experiments. Section 5 presents the experimental setup and the results of training a neural MT model on a text-code dataset generated using our method. In Section 6 we discuss the results, and Section 7 concludes the paper.

2. Related Work

In this section, we do not attempt to show the full range of techniques of matching natural language descriptions to code that have been proposed throughout the literature. Rather, our aim is to investigate which approaches can yield datasets that meet the training needs of statistical text-code language models. Thus, the focus of this review is on studies which are explicitly concerned with applying language models to source code, and which provide some evidence for the performance of text-code language models trained on the proposed datasets.

The performance of the many of the models covered in this review, and indeed the models we present later in the paper, are evaluated using BLEU [8]. BLEU is a de facto standard measure for MT. BLEU compares machine output with human-authored ground-truth translation, and scores how close they are to each other, on a scale from 0 to 100, with 100 indicating a hypothetically perfect translation. In the context of source-code generation from text input, BLEU is calculated by comparing the output of the model to the source-code ground truth from the corpus.

Perhaps the most straightforward solution to creating a dataset of aligned text and code is reported in [9], where a software engineer was hired to manually annotate 18,000 Python statements with pseudo-code. This approach is neither scalable nor cheap, but the study provides interesting insights. The dataset was used to train a phrase-based and a tree-to-string SMT models to generate pseudo-code from source code. The tree-to-string model outperformed the phrase-based model by a large margin (BLEU score of 54 compared to 25), suggesting that correct code-to-text mapping necessitates parsing program structure, and even line-by-line, noise-less descriptions are not sufficient to support a plain phrase-based translation model.

For the work reported in [10] two text-code datasets, one containing Java and the other Python code, were created. In both datasets the code units were aligned with descriptions that combine structured and unstructured text. These datasets were used to train a neural model which generated code from a mix of natural language and structured inputs. The model achieved an impressive performance of 65.6 BLEU scores. Furthermore, the authors trained two neural translation models as baselines, one augmented with their structured attention mechanism. The augmented translation model outperformed the plain translation model on both datasets (BLEU scores of 50 and 44 compared to 34 and 29).

The remaining papers included in this review use a Big Data approach. This research follows two main directions: one toward exploiting Big Code (primarily GitHub (https://github.com)), and the other toward mining programming-related Q&A websites, primarily StackOverflow (https://stackoverflow.com).

The Big Code route involves scraping API documentation (`Javadoc`, Python `docstrings`) from online source-code repositories, and using it as natural language description of code fragments. The research reported in [11] created a massive parallel corpus of over 7.5 million API sequences annotated with excerpts from `Javadoc`. These API sequences are not raw code sequence, they are rather parsed representations of general API usage. Consequently, a neural MT model trained on this corpus would not generate code, instead given some description of required functionality it would produce hints on the APIs that can be used. This model was augmented with information on API importance, and achieved BLEU score of 54.

Another corpus exploiting API documentation [12] consists of over 150,000 Python function bodies annotated with `docstrings`. The authors used a back-translation approach [13] to extend this corpus with a synthetic dataset of 160,000 entries. The performance of a (non-augmented) neural translation model trained on the extended corpus was low (BLEU score of 11).

The second route—assembling datasets from user queries matched to code fragments mined from Q&A websites—has recently attracted a lot of attention. In [14], two training corpora were created from C# snippets extracted from responses to StackOverflow and Dot Net Perls questions, and matched with the titles of these questions. Furthermore, general-purpose engine queries that produced clicks to the questions were added as alternative natural language descriptions. A bi-modal source-code language model trained on the resulting dataset was evaluated in terms of retrieval capability. The model had much better performance when retrieving natural language descriptions (with code snippets as queries) compared with retrieving code snippets, with NL descriptions as queries (Mean Reciprocal Rank of 0.44 as compared to 1.18).

Datasets collected from Q&A websites are large, and have the potential to grow as new questions and answers are added to the websites, but the level of noise in the data is very high. Queries can have irrelevant or very informal titles, and the code snippets are often incomplete and non-compilable. This problem was partly addressed in [14], by applying simple heuristics, but other researchers deemed this approach insufficient and proposed extracting quality datasets from noisy Q&A corpora by applying machine learning models, trained on human-annotated seed datasets, as filters. For example, in one study after collecting C# and SQL snippets produced in response to questions posted on StackOverflow and paired with the titles of these questions, the authors manually annotated a small subset of data and trained a semi-supervised classifier to filter out titles that were irrelevant to the corresponding code snippet [15]. The resulting cleaned corpora (containing over 66,000 pairs for C# and 32,000 of

SQL) were used to training a neural MT model for code summarization (that is, for generating text from code, not code from text), which achieved BLEU scores of 20.5 (C#) and 18.4 (SQL).

Systematic mining of question-code datasets retrieved from Stack Overflow was the main focus of two other studies. In [16], user queries matched with Python and SQL code snippets were subject to a series of cleaning steps. First, a logistic regression classifier (with human-engineered features) was trained to select questions of type "how-to-do-it", in which the user provides a scenario and asks how to implement it. Next, a subset of over 5000 question-code pairs was manually annotated by hired students who judged whether a snippet constitutes a standalone solution to the corresponding question. A novel model, called the Bi-View Hierarchical Neural Network, was trained on the annotated data and used to select over 147,500 Python and 119,500 SQL question-code pairs, to be included in the final dataset (https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset).

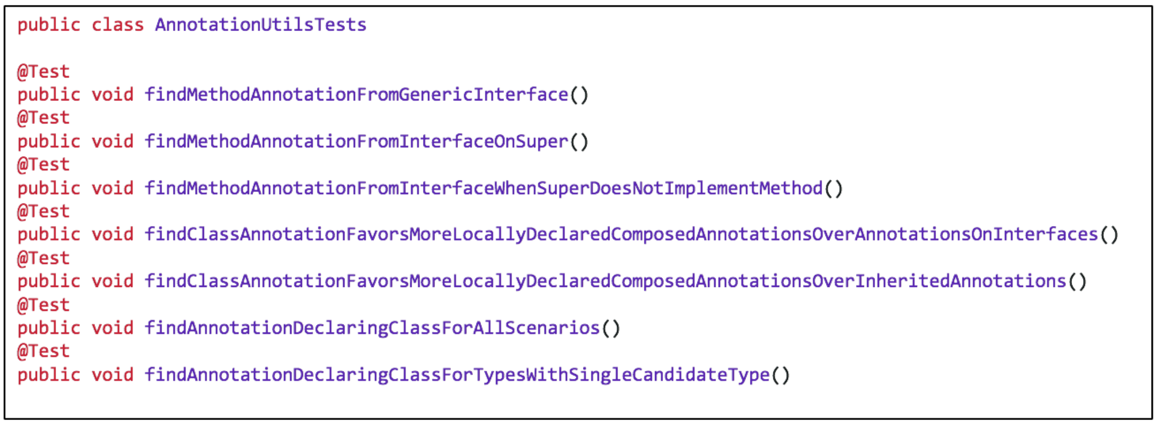
Another complex method to mine high-quality aligned data from Stack Overflow was described in [17]. First, the authors manually engineered a set of code structure features needed to determine the syntactic validity of a code snippet. Second, a subset of collected StackOverflow posts was manually annotated, using a carefully designed annotation procedure, to label specific elements in each post (intent, context, snippet), and to filter out non "how-to-do-it" questions. Next, a neural translation model was trained to learn "correspondence features"—that is, to learn the probability of the intent given a snippet, and the probability of the snippet given an intent. Finally, the calculated probabilities for each intent and snippet were fed to a logistic regression classifier as the correspondence features, together with the manually engineered structural features. After training, the classifier was used to create an automatically mined training corpus containing almost 600,000 intent-snippet pairs (https://conala-corpus.github.io/). Given the small set of annotated data used for training the correspondence features, the authors acknowledged existing threats to validity, and provided an additional dataset of 3,000 examples manually curated by annotators. A baseline neural MT model trained on 100,000 examples from the automatically mined corpus combined with the curated data achieved BLEU score of 14.26. The performance of the model trained on curated data only was even lower (BLEU score 10.58).

3. Text-Code Corpora Acquisition from Self-Documenting Code

In Section 2 we outlined two main approaches to the large-scale acquisition of annotated source code: collecting developer-defined descriptions extracted from API documentation, and collecting user-defined descriptions, extracted from users' questions posted on Q&A websites and matched with code snippets posted as answers. Neither of these approaches is applicable to the testing domain. `Javadoc` or `docstring` only exist for the code that is a part of a public API, and this type of documentation is not available for test automation code. Data collected from Q&A websites contains code snippets that can help in solving concrete programming issues but not in writing specific test cases. Only a small fraction of questions on Stack Overflow are related to testing, and dedicated websites on software quality are far behind in terms of popularity (For example, Software Quality Assurance & Testing website (https://sqa.stackexchange.com/) stores 8500 questions, as compared to 17,000,000 at Stack Overflow). Furthermore, the performance of machine learning models trained on the existing large-scale text-code datasets it low.

In our approach we take advantage of a programming routine known as self-documenting code. Although research on applying statistical language models to source code is relatively new, software developers have long been aware that source code is written for two recipients: one is the machine, and the other—a software developer who will be reviewing, extending, or maintaining the code. The need to secure the interests of the second recipient has been embodied in Clean Code paradigm [18]—a well-known set of best practices focused on making programming code readable and easy to understand for humans. One of the key Clean Code principles is to create variable identifiers and functions names that are meaningful and reveal programmer's intent. To that end, the developer uses multiple words to formulate an adequate description of a function, and then squeezes all the words into the function name, using some convention that helps the reader to recognize individual words (such as `camelCase` in Java, or `snake_case` in Python). Code comments are discouraged, as they carry high risk of being outdated and are often redundant; instead, it is recommended that the code should be self-explanatory.

The method we propose is based on the observation that self-documentation through descriptive method names is widely adopted in test automation, in particular unit testing. Figure 2 shows real-life examples drawn from the `spring-framework`, an open source Java project stored on GitHub (https://github.com/spring-projects/spring-framework). Each of these self-documenting test function names is a concise summary of a specific testing task, written in a quasi-natural language, and observing a consistent naming convention. The body of each function is the implementation of that task in a programming language. Thus, a parallel text-code corpus can be formed from function names split into individual words and aligned with function bodies.



```
public class AnnotationUtilsTests

@Test
public void findMethodAnnotationFromGenericInterface()
@Test
public void findMethodAnnotationFromInterfaceOnSuper()
@Test
public void findMethodAnnotationFromInterfaceWhenSuperDoesNotImplementMethod()
@Test
public void findClassAnnotationFavorsMoreLocallyDeclaredComposedAnnotationsOverAnnotationsOnInterfaces()
@Test
public void findClassAnnotationFavorsMoreLocallyDeclaredComposedAnnotationsOverInheritedAnnotations()
@Test
public void findAnnotationDeclaringClassForAllScenarios()
@Test
public void findAnnotationDeclaringClassForTypesWithSingleCandidateType()
```

Figure 2. Examples of long, descriptive function names from the `spring-framework` project.

4. Methodology

The text-code dataset creation method we propose exploits the self-documenting code and can, in principle, be applied to test cases written in any high-level programming language, provided that the code has been written with the consideration for readability. The generic language-independent procedure can be summarized as follows:

1. Collect automated test cases from source-code repository. Depending on the use case, datasets can be assembled from data within a single repository (for training custom, project- or organization-specific models) or from multiple repositories (for training generic models).
2. For each collected test case:
    (a) Split function identifiers into individual words, according to the adopted naming convention.
    (b) Tokenize the function body, preserving punctuation marks.
    (c) Add the split function name to the corpus as a source sequence. Add the tokenized function body to the corpus as the corresponding target sequence.

On a lower level, however, preprocessing details become language-specific. All the examples presented in this article are based on code written in Java—the second most popular programming language on GitHub (https://blog.github.com/2018-11-15-state-of-the-octoverse-top-programming-languages/). Furthermore, all the test cases we used for dataset synthesis were created with Junit (https://junit.org/)—an open source unit testing framework for Java.

In the experiments we use three datasets synthesized from open source code stored on GitHub. Two of them (labeled **sf** and **rx**) were each extracted from a single large Java repository, and the third one (**multi**) was assembled from test code pulled from over 700 repositories. In the following paragraphs we first describe how we acquired data from a single repository, and then describe the procedure used to assemble the multi-repository corpus.

4.1. Processing Data from A Dingle Depository

To create the project-specific datasets, **sf** and **rx**, we chose two actively maintained GitHub repositories: the `spring-framework` and `RxJava` (https://github.com/ReactiveX/RxJava). This choice was guided by two criteria: the size of the repository and the popularity of the repository, as indicated by the number of stars assigned by users and the number of created forks (repository copies created by members of the GitHub community). We assume that the popularity of a repository is correlated with the quality of code. Table 1 summarizes the properties of the two selected repositories.

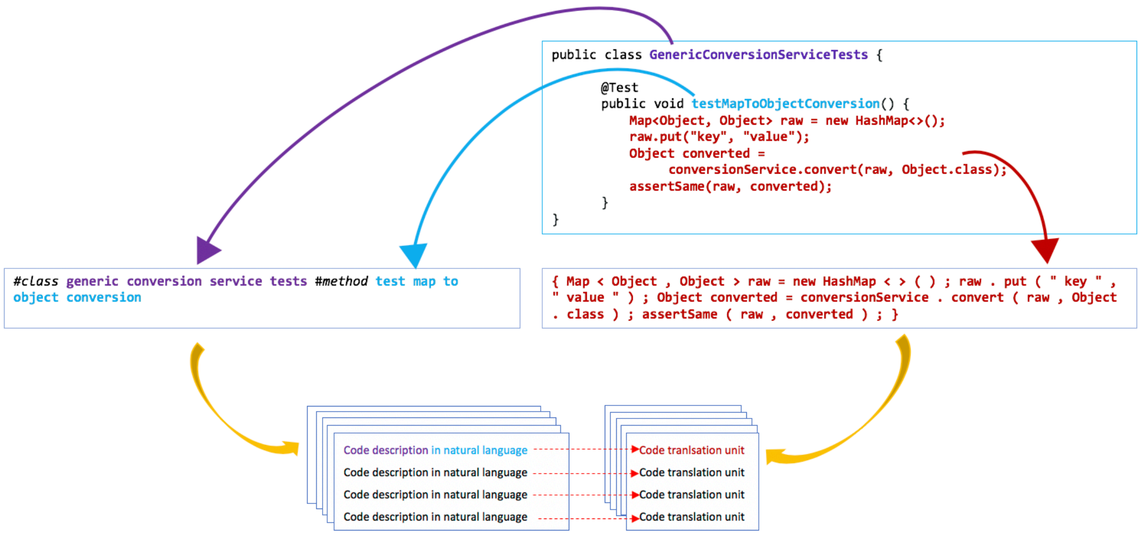
Table 1. Properties of the repositories used for creating project-specific datasets, as of January 2019.

Repository	Size (MB)	Stars	Forks
`spring-framework`	115	25,920	15,546
`RxJava`	72	37,178	5340

We crawl each repository to retrieve all test class files, identified as those containing the `@Test` annotation string. We process every test class using the JavaParser library (http://javaparser.org/) to extract and parse the test functions. Each test function is an automated test case. A parsed test function is represented as a JSON object whose properties comprise function name, function body, parent class name, and some metadata for identification. Any inline comments are separated out from the function body and rejected. The result of this step is a JSON array containing all parsed test functions from the repository.

The next step is the actual synthesis of the corpus. From each JSON file, we take the class name and the function name, which are both camel case strings, convert them into space-separated strings, and prepend with the special tokens `#class` and `#method`, respectively. The quasi-natural language description is created by concatenating the two resulting strings. The programming language sequence is created by simply tokenizing the function body. We assume that tokens such as parentheses, punctuation marks or mathematical symbols should be treated as individual words, and parse the code by surrounding each such character with white-spaces.

Figure 3 shows schematically how the quasi-natural language sentence and the corresponding programming language sequence are derived from a single unit test case.



```
public class GenericConversionServiceTests {

        @Test
        public void testMapToObjectConversion() {
            Map<Object, Object> raw = new HashMap<>();
            raw.put("key", "value");
            Object converted =
                  conversionService.convert(raw, Object.class);
            assertSame(raw, converted);
        }
}
```

```
#class generic conversion service tests #method test map to
object conversion
```

```
{ Map < Object , Object > raw = new HashMap < > ( ) ; raw . put ( " key " ,
" value " ) ; Object converted = conversionService . convert ( raw , Object
. class ) ; assertSame ( raw , converted ) ; }
```

Code description in natural language
Code description in natural language
Code description in natural language
Code description in natural language
Code tranlsation unit
Code translation unit
Code translation unit
Code translation unit

Figure 3. Extracting a text-code sequence pair from a test case.

When adding sequence pairs to the corpus, we apply two filters:

1. Duplicates filter. Any repeating sequence pairs are skipped, although we allow repeating code sequences, as long as they have different natural language descriptions.
2. Length filter. Any sequence pair with a programming language sequence beyond a predefined length (see Table 2) is rejected. This filter has dual purpose. First, it removes code sequences whose size drastically deviates from the average. For example, before applying the length filter, sequences of the length up to 300 tokens comprised over 97% of test cases extracted from `RxJava` repository. The sizes of the remaining 3% ranged from 300 to more than 1000. Second, it mitigates the disproportion between the size of the quasi-natural language sequences and programming language sequences.

4.2. Collecting Data from Multiple Repositories

The data used to create the multi-repository corpus was collected using the github3 (https://pypi.org/project/github3.py/) library—a Python wrapper for the GitHub API. The search query included two parameters: `language:java` and `stars:>1000`. We were not concerned with the size of individual repositories. The query returned almost 2000 repositories. Over 700 of them contained Junit test cases which we used for dataset creation.

The repositories were processed one by one according to the procedure described in Section 4.1. In addition to applying the duplicate and length filters to each individual repository, we also checked for and removed any duplicates across the repositories. We also applied some simple heuristics to exclude functions with meaningless names, such as `test1()`, `test2()` etc.

Table 2 lists the metadata for the three datasets used in the experiments. The source-code vocabulary sizes are very large. This is not surprising, given that each programming API brings in new source-code tokens. For the **rx** and **sf** corpora we kept the full PL vocabularies. For the multi-repository dataset, we limited the PL vocabulary size by discarding all tokens that occurred less than n times in the corpus. In the dataset, words excluded from the vocabulary were mapped to a special *unknown* token. We created two versions of the multi-repository dataset, one with $n = 5$ and the other with $n = 10$.

Table 2. Properties of datasets used in the experiments described in Section 5. NL refers to natural language and PL—to programming language.

	rx	sf	multi-min5	multi-min10
Corpus size	10,069	13,792	297,728	297,728
Trainset size	8055	11,033	267,689	267,689
Testset size	2014	2759	30,039	30,039
Vocabulary size (NL)	1067	2679	18,805	18,805
Vocabulary size (PL)	3032	18,422	70,417 *	31,738 *
Sequence length limit	300	200	150	150
Avg. sequence length (NL)	9	11	11	11
Avg sequence length (PL)	94	76	45	45

* The size after rare tokens have been mapped to the *unknown* token.

5. Experiments

The purpose of the experiments reported in this section is to investigate whether test code annotated with descriptions extracted from meaningful function names provides good quality data for training statistical language models. To do this, we use the corpora described in Section 4 to train a well-known neural translation model to generate test code from quasi-natural language descriptions.

Neural translation models are an example of end-to-end learning: these models take a source language sentence as an input, encode it into a dense vector (inter-lingual) representation, and decode this representation to generate the translation target language sentence. The model used in our study belongs to the class of sequence-to-sequence models [19], and is a TensorFlow implementation of the attention-based architecture proposed in [7] (https://github.com/tensorflow/nmt/).

We trained three models, one on each of the datasets. In the preliminary phase we performed trial runs to establish a set of reasonable hyperparameters, and then kept them constant throughout the experiments. We used 2-layered LSTMs for both the encoder and the decoder, with the Bahdanau attention mechanism[20]. For the optimizer we used Adam with a starting learning rate of 0.001. Regarding regularization, we set dropout probability to 0.2, as recommended for LSTMs [21].

Each model was evaluated on a test set that was randomly sampled from the dataset the model was trained on. These test sets were sampled prior to training and were held-out from the training process. In case of the multi-repository corpus, the test set is the concatenation of test sets sampled from each contributing repository. The results are presented in Table 3.

Table 3. BLEU scores from training on three datasets.

Training Corpus	BLEU
rx	82.44
sf	58.00
multi-min5	48.85
multi-min10	50.50

BLEU scores on code generation are not directly comparable with scores on natural language translation due the differences in the length of the translated strings and the inherent differences in the (natural vs. programming) language structures. Moreover, it is well known in the MT community that BLEU scores are not necessarily a true reflection of translation quality. These caveats aside, however, the results from our experiments are promising and indicate that our approach of generating test case source code from parsed class and function names is feasible (BLEU score obtained on multi-min10 is somewhat higher than the score obtained on multi-min5, which may seem surprising, given that in the context of natural language translations, the higher number of unknown words has been shown to have a negative impact on performance [20,22]. However, since the aim of our experiments was to confirm the usefulness of the parallel corpora built from self-documenting code (rather than the evaluation of a specific machine learning model), we put limited effort into hyperparameters optimization, and it is possible that a more extensive search of the hyperparameter space would provide slightly different results. Investigating the impact of the vocabulary size in the context of programming language would be useful, but remains out the scope of this paper). In the following section we present a qualitative, example-based analysis of the performance of our models with a view to better understanding and contextualizing the results of training an NMT model on the three datasets extracted from self-documenting code.

6. Discussion

Acknowledging the limitations of BLEU metrics as a tool for evaluating the quality of generated code, we carefully reviewed the outputs of our models and analyzed them in the context of the examples available in the training corpora. In this section, we discuss the findings from this qualitative analysis of the three datasets.

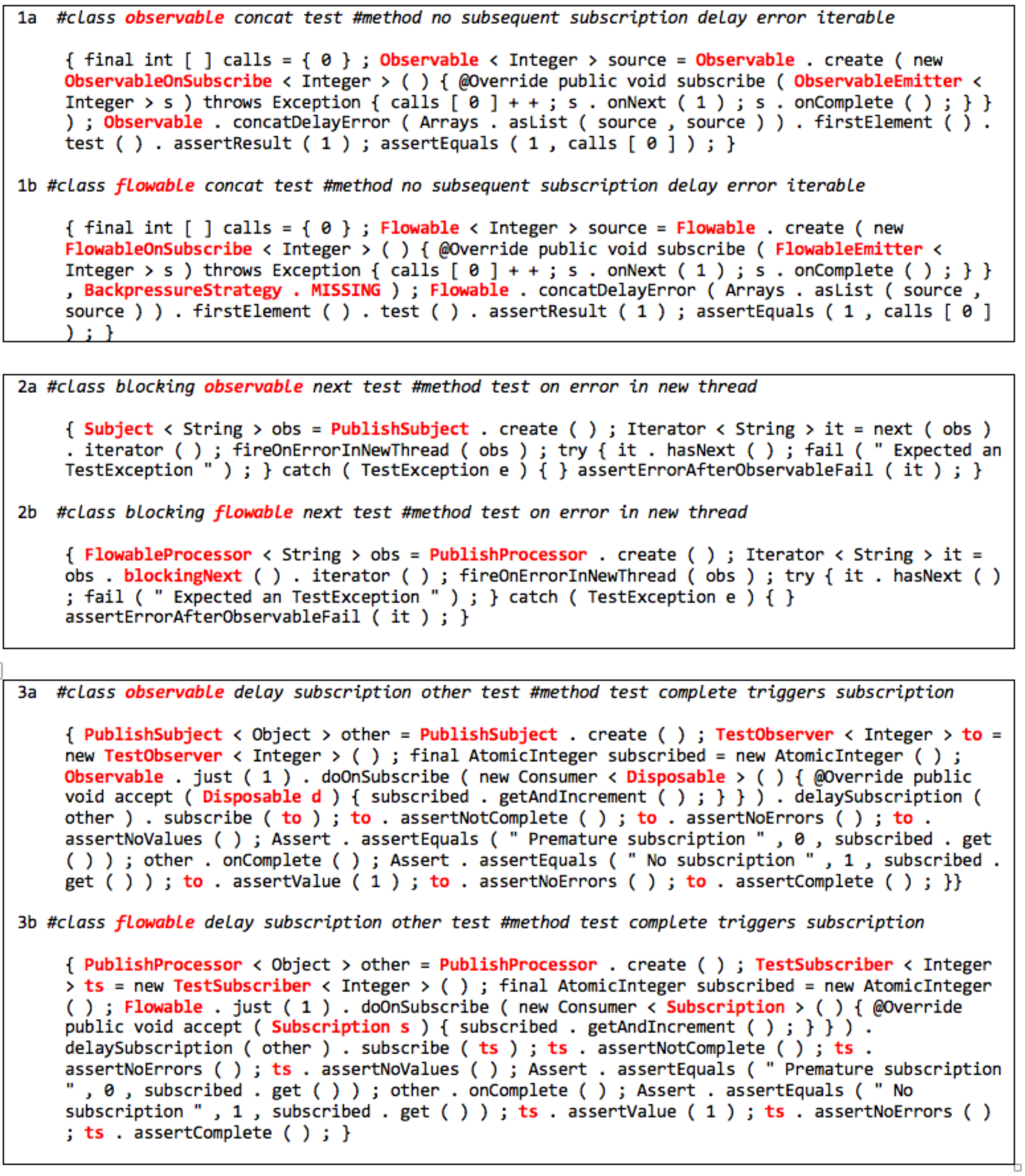
6.1. Dataset rx

The unusually high performance of the model trained on the **rx** dataset becomes less surprising after a closer look into the RxJava project test code. RxJava features several base classes exhibiting standardized behaviors that can be—and are—tested by analogous test functions. For instance, a large body of analogous test code exists for the base classes `Observable` and `Flowable`. Examples of highly equivalent functions testing these two classes are shown in Figure 4. Each test function is presented in the parsed format, as a text-code pair. The parts forming textual descriptions are marked in italics. The differences between equivalent function versions (a, b) are highlighted in bold.

The replacement patterns emphasized in Figure 4 (`Observable` vs. `Flowable`, `PublishSubject` vs. `PublishProcessor`, `TestObserver` vs. `TestSubscriber`, `BackpressureStrategy` and `blockingGet` vs. the absence of these patterns) are highly repeatable throughout the corpus, making the learning task much easier. In fact, the model was able to fully generate the unseen long function body shown in 3a, based on the NL description (*#class flowable delay subscription other test #method test complete triggers subscription*) which was a part of our test set.

After completed training of the **rx** model, we reviewed all the code generated against NL descriptions from the test set. Out of 2014 examples in the test set, 965 (47.9%) generated sequences that fully matched unseen ground-truth source code. Clearly, the repeatable patterns occurring in analogous test functions strongly contributed to this result, but the model was also able to exploit other regularities in the training data.

In the case of short code sequences, it is relatively easy to track back at least some of the pieces of information used by the model to generate test functions. Table 4 presents the analysis of four correctly predicted test functions. The top test-code pair (bold font) in each section of the table is pulled from the test set. The remaining text-code pairs (plain font) are the pulled from the trainset. For the **rx** dataset, we selected a function (rxe) that—in contrast to the patterns depicted in Figure 4—did not have an analogous counterpart in the trainset. Training examples (rxt) reveal how the model learned the code sequence representing the string *timeout* in the description.



```
1a  #class observable concat test #method no subsequent subscription delay error iterable

    { final int [ ] calls = { 0 } ; Observable < Integer > source = Observable . create ( new
    ObservableOnSubscribe < Integer > ( ) { @Override public void subscribe ( ObservableEmitter <
    Integer > s ) throws Exception { calls [ 0 ] + + ; s . onNext ( 1 ) ; s . onComplete ( ) ; } }
    ) ; Observable . concatDelayError ( Arrays . asList ( source , source ) ) . firstElement ( ) .
    test ( ) . assertResult ( 1 ) ; assertEquals ( 1 , calls [ 0 ] ) ; }

1b #class flowable concat test #method no subsequent subscription delay error iterable

    { final int [ ] calls = { 0 } ; Flowable < Integer > source = Flowable . create ( new
    FlowableOnSubscribe < Integer > ( ) { @Override public void subscribe ( FlowableEmitter <
    Integer > s ) throws Exception { calls [ 0 ] + + ; s . onNext ( 1 ) ; s . onComplete ( ) ; } }
    , BackpressureStrategy . MISSING ) ; Flowable . concatDelayError ( Arrays . asList ( source ,
    source ) ) . firstElement ( ) . test ( ) . assertResult ( 1 ) ; assertEquals ( 1 , calls [ 0 ]
    ) ; }
```

```
2a #class blocking observable next test #method test on error in new thread

    { Subject < String > obs = PublishSubject . create ( ) ; Iterator < String > it = next ( obs )
    . iterator ( ) ; fireOnErrorInNewThread ( obs ) ; try { it . hasNext ( ) ; fail ( " Expected an
    TestException " ) ; } catch ( TestException e ) { } assertErrorAfterObservableFail ( it ) ; }

2b  #class blocking flowable next test #method test on error in new thread

    { FlowableProcessor < String > obs = PublishProcessor . create ( ) ; Iterator < String > it =
    obs . blockingNext ( ) . iterator ( ) ; fireOnErrorInNewThread ( obs ) ; try { it . hasNext ( )
    ; fail ( " Expected an TestException " ) ; } catch ( TestException e ) { }
    assertErrorAfterObservableFail ( it ) ; }
```

```
3a  #class observable delay subscription other test #method test complete triggers subscription

    { PublishSubject < Object > other = PublishSubject . create ( ) ; TestObserver < Integer > to =
    new TestObserver < Integer > ( ) ; final AtomicInteger subscribed = new AtomicInteger ( ) ;
    Observable . just ( 1 ) . doOnSubscribe ( new Consumer < Disposable > ( ) { @Override public
    void accept ( Disposable d ) { subscribed . getAndIncrement ( ) ; } } ) . delaySubscription (
    other ) . subscribe ( to ) ; to . assertNotComplete ( ) ; to . assertNoErrors ( ) ; to .
    assertNoValues ( ) ; Assert . assertEquals ( " Premature subscription " , 0 , subscribed . get
    ( ) ) ; other . onComplete ( ) ; Assert . assertEquals ( " No subscription " , 1 , subscribed .
    get ( ) ) ; to . assertValue ( 1 ) ; to . assertNoErrors ( ) ; to . assertComplete ( ) ; }}

3b #class flowable delay subscription other test #method test complete triggers subscription

    { PublishProcessor < Object > other = PublishProcessor . create ( ) ; TestSubscriber < Integer
    > ts = new TestSubscriber < Integer > ( ) ; final AtomicInteger subscribed = new AtomicInteger
    ( ) ; Flowable . just ( 1 ) . doOnSubscribe ( new Consumer < Subscription > ( ) { @Override
    public void accept ( Subscription s ) { subscribed . getAndIncrement ( ) ; } } ) .
    delaySubscription ( other ) . subscribe ( ts ) ; ts . assertNotComplete ( ) ; ts .
    assertNoErrors ( ) ; ts . assertNoValues ( ) ; Assert . assertEquals ( " Premature subscription
    " , 0 , subscribed . get ( ) ) ; other . onComplete ( ) ; Assert . assertEquals ( " No
    subscription " , 1 , subscribed . get ( ) ) ; ts . assertValue ( 1 ) ; ts . assertNoErrors ( )
    ; ts . assertComplete ( ) ; }
```

Figure 4. Examples of functions written to validate analogous behavior of `Observable` (1a, 2a, 3a) and `Flowable` (1b, 2b, 3b).

Table 4. Examples of correctly predicted unseen functions, together with seen instances that provided necessary information. Each row in the table contains the following data: Lbl (label)—created by concatenating the dataset label (rx, sf, m for multi) with a suffix indicating whether the text-code pair comes from the test set (suffix e) or the trainset (suffix t). ; Text—the quasi-natural description (input sequence); Code—the test code (output sequence). Text-code pairs coming from the test set (rxe, sfe, me) are additionally highlighted in bold.

Lbl	Text	Code
rxe	**#c completable await test #m blocking get**	**assertNull (Completable . complete ( ) . blockingGet ( )) ;**
rxt	#c completable await test #m blocking get timeout	assertNull (Completable . complete ( ) . blockingGet (1 , TimeUnit . SECONDS)) ;
rxt	#c completable await test #m blocking get error	TestException ex = new TestException ( ) ; assertSame (ex , Completable . error (ex) . blockingGet ( )) ;
rxt	#c completable await test #m blocking get error timeout	TestException ex = new TestException ( ) ; assertSame (ex , Completable . error (ex) . blockingGet (1 , TimeUnit . SECONDS)) ;
sfe	**#c evaluation tests #m test rel operator ge 01**	**evaluate (" 3 > = 6 " , " false " , Boolean . class) ;**
sft	#c evaluation tests #m test rel operator gt 01	evaluate (" 3 > 6 " , " false " , Boolean . class) ;
sft	#c evaluation tests #m test rel operator lt 01	evaluate (" 3 < 6 " , " true " , Boolean . class) ;
sft	#c evaluation tests #m test rel operator le 01	evaluate (" 3 < = 6 " , " true " , Boolean . class) ;
me	**#c traverser test #m for graph depth first pre order single root**	**Iterable < Character > result = Traverser . forGraph (SINGLE_ROOT) . depthFirstPreOrder (’a’) ; assertEqualCharNodes (result , " a ") ;**
mt	#c traverser test #m for graph breadth first single root	Iterable < Character > result = Traverser . forGraph (SINGLE_ROOT) . breadthFirst (’a’) ; assertEqualCharNodes (result , " a ") ;
mt	#c traverser test #m for graph depth first pre order two trees	Iterable < Character > result = Traverser . forGraph (TWO_TREES) . depthFirstPreOrder (’a’) ; assertEqualCharNodes (result , " ab ") ;
mt	#c traverser test #m for graph depth first post order two trees	Iterable < Character > result = Traverser . forGraph (TWO_TREES) . depthFirstPostOrder (’a’) ; assertEqualCharNodes (result , " ba ") ;
mt	#c traverser test #m for graph breadth first two trees	Iterable < Character > result = Traverser . forGraph (TWO_TREES) . breadthFirst (’a’) ; assertEqualCharNodes (result , " ab ") ;
me	**#c thread dump servlet test #m returns 200 ok**	**assertThat (response . getStatus ( )) . isEqualTo (200) ;**
mt	#c thread dump servlet test #m returns text plain	assertThat (response . get (HttpHeader . CONTENT_TYPE)) . isEqualTo (" text / plain ") ;
mt	#c ping servlet test #m returns 200 ok	assertThat (response . getStatus ( )) . isEqualTo (200) ;
mt	#c ping servlet test #m returns text plain	assertThat (response . get (HttpHeader . CONTENT_TYPE)) . isEqualTo (" text / plain ; charset = ISO - 8859 - 1 ") ;

6.2. Dataset sf

The phenomenon of analogous functions is not observed in the test code pulled from `spring-framework`, which probably explains why the NMT model trained on the **sf** dataset achieved less spectacular performance than the one trained on the **rx** dataset. Nevertheless, the BLEU score of 58 obtained by the **sf** model indicates that consistent naming conventions are useful for learning mappings between textual descriptions and code sequences. As we did previously with the **rx** dataset, we analyzed the examples generated by the model from the descriptions in the test set. Table 4 shows an interesting example of a generated function (sfe), in which the model was able to correctly predict mathematical symbols corresponding to the string *ge* (*greater-than-or-equal-to*) based on the seen examples *gt*, *lt*, *le* (*greater-than*, *less-than* and *less-than-or-equal-to*).

The total number of model predictions that fully matched the ground-truth was 224 out of 2759 examples in the test set (8%).

6.3. Dataset Multi

The BLEU score obtained by the model trained on multi-repository dataset was significantly lower that than the score of the **sf** model, but the percentage of fully matched predictions was actually slightly higher (9.7%, or 2,908 out of 30,039). On analyzing these fully matched outputs of the model trained on the multi-repository dataset, we did not find evidence that the model learned to use information across projects originating from different repositories. For example, the correct predictions produced for the two functions from the multi-repository test set shown in Table 4 (labeled *me*) were produced based on examples originating from the same project as the generated sequence (i.e., the same parent project). However, it seems that generating a correct code sequence does not require a large parent project. The parent projects of the two examples in Table 4 are `google/guava` (https://github.com/google/guava) and `dropwizard/metrics` (https://github.com/dropwizard/metrics), represented in the trainset by 288 and 355 examples, respectively.

Although the model trained on the **multi** dataset performed worse than the project-specific models, it is worth remembering that the vanilla neural translation models trained on the parallel corpora extracted from API documentation [12] or Stack Overflow queries [17] achieved BLEU scores of 11 and 14, respectively (It is not possible to make direct comparisons with the results obtained using models dedicated for code processing task (exploiting structural/syntactical information about the code, building Abstract Syntax Tree representations etc.). The comparisons of the potential of training corpora only make sense for experiments that used a similar ML model. From the studies discussed in Section 2, four papers report the results of training a pure NMT model. However, the results reported in [11,15] do not refer to code generation task (the first paper focuses on generating parsed API usage examples, and the second is concerned with code summarization (generating text from code))). The relatively high score of the model trained on data assembled from 700 GitHub repositories seems to confirm that the principle of self-documenting code is popular among test automation developers and can be relied upon as a source of training data.

7. Conclusions and Future Work

In this paper, we have presented a method that exploits the availability of source code in open software repositories to automatically construct an aligned text-code dataset. This method leverages self-documenting code—the software engineering practice of using meaningful class and function names to describe the functionality of program modules. Furthermore, we have demonstrated how datasets constructed in this way can be used to train (MT-inspired) text to source-code generation systems. Moreover, we have shown that it is feasible to use this machine translation-based code generation approach for automatic test case generation.

A key differentiation between our approach and the methods discussed in Section 2 is that the textual descriptions in our parallel corpora are not expressed in true natural language. In NLP, a lack of the naturalness is often considered a weakness, but in the context of the software testing domain, the quasi-natural language nature of the text in our generated datasets does not affect usability, because this language has been devised by the software developer community. This form of communication has a lot in common with Controlled Natural Languages [23]: it uses simplified syntax but preserves most of the natural properties of its base language and can be intuitively understood. As elaborated on in Section 6, this compliance by software developers with naming conventions results in consistent repeatable patterns within the generated datasets which make the learning task feasible. Furthermore, unlike previous attempts to leverage developer-defined descriptions of code [11,12], our approach can be applied to generating code that is not exposed as a public API and therefore lacks inline documentation. Admittedly, escaping one limitation (the restriction to code sourced from public APIs) comes at the cost of another limitation (the restriction to self-documenting code only).

We have demonstrated the feasibility of our approach within the software testing domain. Specifically, our experiments have been on the generation of unit test cases, which can be described in a single quasi-natural language sentence and which have relatively short code bodies. As a result, one open question that remains is whether our approach can be generalized to other (potentially more complex) code domains. This is a question we will address in future work. However, even with this potential limitation the current results are very worthwhile because the demand for automated tests in the modern software development cycle is very high, and we believe it is important to fill the gap in the availability of training data for developing test automation tools, even if the approach is not universal. The fact that existing neural translation models trained on this type of data can achieve satisfactory performance is evidence of the high quality of the text-code parallel corpora synthesized from class and function names. Indeed, we believe that the performance of these initial systems is at a level that permits the immediate application of our approach in the area of software engineering.

That said, the evaluation of the true value of the generated code requires far more effort. As pointed out in Section 5, a BLEU score is an approximate indicator of the quality of translation of natural languages. We have not identified any empirical studies investigating the applicability of BLEU to source code, and therefore the results reported in this and other papers need to be treated with caution.

A more reliable evaluation would involve retrieving feedback from human users. Our current efforts are focused on developing a Test Recommendation Engine trained on the corpora extracted from self-documenting code. The Engine, which is a part of a Horizon 2020 project (https://elastest.eu/), will be released publicly, and we have plans for the collaboration with several industry test teams to gather their feedback.

We envisage two use cases benefiting from automated test generation. The first one involves training a project-specific model, similar to the ones trained on the **rx** and **sf** datasets. The tools built using such a specialized model are likely to produce accurate test cases for the new code written as developers add or modify features in an already mature project. The second use case involves a generic model trained on multiple repositories. In this case, the predictions are likely to be less accurate (and so require some editing) but the model would still be of value for test teams working on new projects with a minimal codebase.

Author Contributions: Conceptualization, M.K. and J.D.K.; methodology, M.K.; software, M.K.; investigation, M.K; Writing—original draft preparation, M.K.; Writing—review and editing, J.D.K.; Supervision, J.D.K.

Funding: This work was partly funded by the EC project ElasTest, 731535 (H2020-ICT-2016-2017/H2020-ICT-2016-1, Research and Innovation Action). The work was also supported by the ADAPT Centre which is funded under the SFI Research Centres Programme (Grant 13/RC/2106) and is co-funded under the European Regional Development Fund.

Acknowledgments: Magdalena Kacmajor would like to include special thanks to Micael Gallego Carrillo from Universidad Rey Juan Carlos in Madrid, who first brought her attention to self-documenting code as a possible source of training data.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CNL	Controlled Natural Language
DevOps	Development and Operations
ML	Machine Learning
MT	Machine Translation
NL	Natural Language
NLP	Natural Language Processing
NMT	Neural Machine Translation
PL	Programming Language
SMT	Statistical Machine Translation

References

1. Virmani, M. Understanding DevOps & bridging the gap from continuous integration to continuous delivery. In Proceedings of the 5th International Conference on the Innovative Computing Technology (INTECH 2015), Galicia, Spain, 20–22 May 2015; pp. 78–82. [CrossRef]
2. Perera, P.; Silva, R.; Perera, I. Improve software quality through practicing DevOps. In Proceedings of the 2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer), Colombo, Sri Lanka, 6–9 September 2017; pp. 1–6. [CrossRef]
3. Anand, S.; Burke, E.K.; Chen, T.Y.; Clark, J.; Cohen, M.B.; Grieskamp, W.; Harman, M.; Harrold, M.J.; McMinn, P.; Bertolino, A.; et al.. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.* **2013**, *86*, 1978–2001. [CrossRef]
4. Shamshiri, S.; Just, R.; Rojas, J.M.; Fraser, G.; McMinn, P.; Arcuri, A. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 201–211. doi:10.1109/ASE.2015.86. [CrossRef]
5. Hindle, A.; Barr, E.T.; Gabel, M.; Su, Z.; Devanbu, P. On the naturalness of software. *Commun. ACM* **2016**, *59*, 122–131. [CrossRef]
6. Allamanis, M.; Barr, E.T.; Devanbu, P.T.; Sutton, C.A. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* **2018**, *51*, 81. [CrossRef]
7. Luong, T.; Pham, H.; Manning, C.D. Effective Approaches to Attention-based Neural Machine Translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, Lisbon, Portugal, 17–21 September 2015; pp. 1412–1421. [CrossRef]
8. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. Bleu: A Method for Automatic Evaluation of Machine Translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 7–12 July 2002; pp. 311–318. [CrossRef]
9. Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; Nakamura, S. Learning to generate pseudo-code from source code using statistical machine translation (T). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 574–584. [CrossRef]
10. Ling, W.; Blunsom, P.; Grefenstette, E.; Hermann, K.M.; Kočiský, T.; Wang, F.; Senior, A. Latent Predictor Networks for Code Generation. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 599–609. [CrossRef]
11. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API Learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 631–642. [CrossRef]
12. Miceli Barone, A.V.; Sennrich, R. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In Proceedings of the Eighth International Joint Conference on Natural Language Processing, Taipei, Taiwan, 27 November–1 December 2017; pp. 314–319.
13. Sennrich, R.; Haddow, B.; Birch, A. Improving Neural Machine Translation Models with Monolingual Data. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 86–96.

14. Allamanis, M.; Tarlow, D.; Gordon, A.D.; Wei, Y. Bimodal Modelling of Source Code and Natural Language. In Proceedings of the 32nd International Conference on International Conference on Machine Learning, Lille, France, 6–11 July 2015; pp. 2123–2132.
15. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing Source Code using a Neural Attention Model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; pp. 2073–2083. [CrossRef]
16. Yao, Z.; Weld, D.S.; Chen, W.P.; Sun, H. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In Proceedings of the 2018 World Wide Web Conference, Lyon, France, 23–27 April 2018; pp. 1693–1703. [CrossRef]
17. Yin, P.; Deng, B.; Chen, E.; Vasilescu, B.; Neubig, G. Learning to mine aligned code and natural language pairs from stack overflow. In Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, 28–29 May 2018; pp. 476–486. [CrossRef]
18. Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed.; Prentice Hall PTR: Upper Saddle River, NJ, USA, 2008.
19. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to Sequence Learning with Neural Networks. In Proceedings of the 27th International Conference on Neural Information Processing Systems, Montreal, QC, Canada, 8–13 December 2014; pp. 3104–3112.
20. Bahdanau, D.; Cho, K.; Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv* **2014**, arXiv:1409.0473. Available online: https://arxiv.org/abs/1409.0473 (accessed on 15 February 2019).
21. Zaremba, W.; Sutskever, I.; Vinyals, O. Recurrent Neural Network Regularization, 2014. *arXiv* **2014**, arXiv:1409.2329. Available online: https://arxiv.org/abs/1409.2329 (accessed on 15 February 2019).
22. Cho, K.; van Merrienboer, B.; Bahdanau, D.; Bengio, Y. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In Proceedings of the SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014; pp. 103–111.
23. Kuhn, T. A Survey and Classification of Controlled Natural Languages. *Comput. Linguist.* **2014**, *40*, 121–170. [CrossRef]