Unit Test Case Generation with Transformers

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, Neel Sundaresan
Microsoft
Redmond, WA, USA
Email: {mitufano, dadrain, alsvyatk, shade, neels}@microsoft.com

***Abstract*—Automated Unit Test Case generation has been the focus of extensive literature within the research community. Existing approaches are usually guided by the test coverage criteria, generating synthetic test cases that are often difficult to read or understand for developers.**

In this paper we propose ATHENATEST, an approach that aims at generating unit test cases by learning from real-world, developer-written test cases. Our approach relies on a state-of-the-art sequence-to-sequence transformer model which is able to write useful test cases for a given method under test (*i.e.,* focal method). We also introduce METHODS2TEST– the largest publicly available supervised parallel corpus of unit test case methods and corresponding focal methods in Java, which comprises 630k test cases mined from 70k open-source repositories hosted on GitHub. We use this dataset to train a transformer model to translate focal methods into the corresponding test cases.

We evaluate the ability of our model in generating test cases using natural language processing as well as code-specific criteria. First, we assess the quality of the translation compared to the target test case, then we analyze properties of the test case such as syntactic correctness and number and variety of testing APIs (*e.g.,* asserts). We execute the test cases, collect test coverage information, and compare them with test cases generated by EvoSuite and GPT-3. Finally, we survey professional developers on their preference in terms of readability, understandability, and testing effectiveness of the generated test cases.

***Index Terms*—Automated Software Testing, Deep Learning**

I. INTRODUCTION

Software testing is widely acknowledged as one of the most critical, challenging, and expensive phases of the software development lifecycle. Technology companies are constantly looking into ways to deliver their software faster, without sacrificing its quality and correctness. To succeed, these companies often rely on continuous integration and delivery of software, which allows for fast and reliable deployment of software into production. In this context, automated testing represents a fundamental piece of the pipeline, providing developers with the confidence they need to iterate quickly, and integrate new features without regressions.

Unit testing lays as the foundational basis of the testing pyramid, beneath integration and end-to-end testing [1]. This prominent visual metaphor intends to provide a guidance on the adequate amount of effort that should be allocated for each of the testing layers. Thus, the largest amount of tests should be at the unit test layer, where individual units of software (*e.g.,* a single method) are tested in isolation to ensure that they behave as intended.

Unit Test frameworks, such as JUnit [2], offer an environment and APIs that facilitate writing and executing repeatable test cases. JUnit provides methods such as *assertions* which support the developers in checking conditions, outputs, or states in a software program, assessing its expected behavior. Several other frameworks have been built on top of JUnit, such as Cactus [3] and TestnNG [4]. Others can be integrated with JUnit to support different scenarios or testing methodologies, such as Mockito [5], which allows mocking of objects by replacing functionalities with dummy implementations that emulate real code, focusing the testing on the method under test.

On top of these frameworks, researchers have proposed several techniques that aim to automate the generation of unit test cases. EvoSuite [6], Randoop [7], and Agitar [8] are among the most popular and widely used examples of such techniques. EvoSuite relies on an evolutionary approach based on a genetic algorithm to generate unit test cases, targeting code coverage criteria such as branch and line coverage. Specifically, it introduces mutants (*i.e.,* modified versions of methods or classes under test) and iteratively generates assert statements to kill such mutants. During this process, EvoSuite minimizes the number of asserts while trying to maximize the number of detected mutants. Randoop is a different automated test generation tool that relies on feedback-directed random testing, a technique that uses execution traces to guide the selection of method sequences which are then checked against a set of user-specified contracts (*i.e.,* user-specified program logic).

A major weakness and criticism of these approaches is related to the poor readability and understandability of the generated test cases [9], [10], which clearly appear as machine-generated code. Other studies have highlighted different limitations of these automation tools, such as unsatisfactory code quality [11]–[13], poor fault-detection capability [14], and the inability to adequately meet the software testing needs of industrial developers [15], [16]. These limitations stem from the fact that these approaches mainly focus on code coverage as unique objective, disregarding other factors that may be relevant for developers.

Deep learning techniques have shown the potential of learning from real-world examples, and have been employed in several software engineering tasks, such as code completion [17], automated patch generation [18], [19], comment generation [20], and many others [21]. Recent advancements in transformer models, such as OpenAI GPT-3 [22], have made headlines and shown impressive results in realistic text generation and question answering tasks.

In this paper, we present an approach that aims to ***learn from developer-written test cases how to generate correct and readable tests.*** Our approach relies on a large sequence-to-sequence transformer model pretrained both on English and

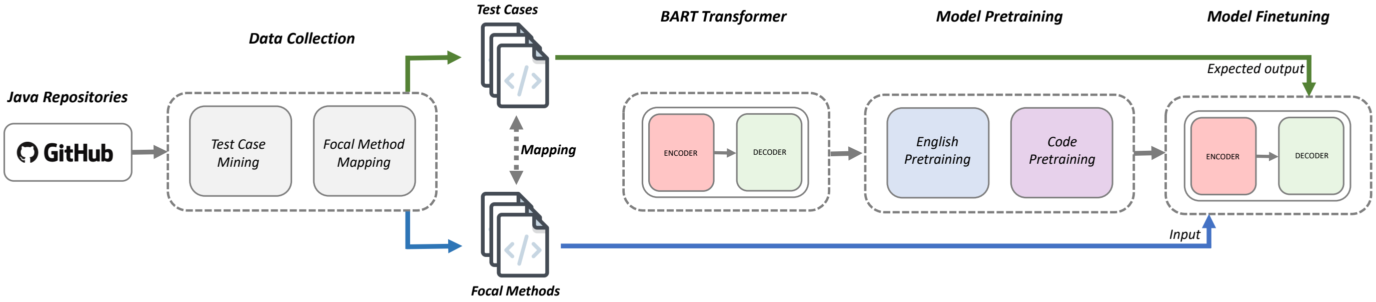


Fig. 1. Overview of AthenaTest – We mine test cases from GitHub and map them to the corresponding focal methods, which we collect in Methods2Test, then pretrain a BART Transformer model on both English and Source Code corpora, finally we finetune the model on the unit test case generation task.

Java source code, then finetuned on the task of generating unit test cases. For this task, we mine thousands of real-world test cases and map them to the corresponding focal methods, then use this parallel corpus for training and evaluation.

To summarize, our contributions are as follows:

- AthenaTest: an automated test case generation approach based on a sequence-to-sequence transformer model. The approach is able to generate syntactically correct test cases that invoke a variety of testing APIs. The generated test cases have comparable test coverage w.r.t. EvoSuite and they are preferred by professional developers in terms of readability, understandability, and testing effectiveness. These test cases appear to be: (i) *realistic* – similar to developer-written test cases; (ii) *accurate* – correctly asserting the expected behavior of a focal method; (iii) *human-readable* – readable and understandable code, with good variable and method names.
- Methods2Test: the largest publicly available[1] parallel corpus of test cases mapped to the corresponding focal methods [23]. This dataset enlists 630k mapped test cases, extracted from 70k open source Java projects.

II. Approach

Figure 1 provides an overview of our approach. Starting with a dataset of Java open-source projects obtained from GitHub, we mine test cases and map them to the corresponding focal methods (Sec. II-A). Next, we finetune a transformer model (Sec. II-B), which has been pretrained on English and source code corpora (Sec. II-C), for the task of generating unit test cases (Sec. II-D).

A. Data Collection

The goal of this stage is to mine test cases and their corresponding focal methods (*i.e.,* the method tested by the test case) from a set of Java projects. We select a 70k sample of all the public GitHub Java repositories with at least ten stars that have been updated within the last five years and are not forks.

First, we parse each project to obtain classes and methods with their associated metadata. Next, we identify each test class and its corresponding focal class. Finally, for each test case within a test class, we map it to the related focal method obtaining a set of mapped test cases.

Parsing: We parse each project under analysis with the `tree-sitter` parser [24]. During the parsing, we automatically collect metadata associated with the classes and methods identified within the project. Specifically, we extract information such as method and class names, signatures, bodies, annotations, and variables.

Find Test Classes: In this stage, we identify all the test classes, which are classes that contain a test case. To do so, we mark a class as a test class if it contains at least one method with the `@Test` annotation. This annotation informs JUnit that the method to which it is attached can be run as a test case.

Find Focal Classes: For each test class we aim to identify the focal class which represents the class under test. To this aim, we employ the following two heuristics, in sequence:

- *Path Matching*: best practices for JUnit testing suggests placing code and corresponding test cases in mirrored folder structure. Specifically, given the class `src/main/java/Foo.java` the corresponding JUnit test cases should be placed in the class `src/test/java/FooTest.java`. Our first heuristic tries to identify the folder where the focal class is defined, by following the path of the test class but starting with the `src/main` folder (*i.e.,* production code).
- *Name Matching*: the name of a test class is usually composed of the name of the focal class, along with a "Test" prefix or suffix. For example, the test case for the class `Foo.java` would probably be named `FooTest.java`. Thus, following the path matching heuristic, we perform name matching to identify the focal class by matching the name of the test case without the (optional) "Test" prefix/suffix.

Find Focal Method: For each test case (*i.e.,* method within a test class with the `@Test` annotation) we attempt to identify the corresponding focal method within the focal class. To this aim, we employ the following heuristics:

- *Name Matching*: following the best practices for naming classes, test case names are often similar to the corresponding focal methods. Thus, the first heuristic attempts to match the test cases with a focal method having a name that matches, after removing possible `Test` prefix/suffix.
- *Unique Method Call*: if the previous heuristic did not identify any focal method, we compute the intersection

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

between (i) the list of method invocations within the test case and (ii) the list of methods defined within the focal class. If the intersection yields a unique method, then we select the method as the focal method. The rationale behind this approach is as follows: since we have already matched the test class with the focal class (with very high confidence heuristics), if the test case invokes a single method within that focal class, it is very likely testing that single method.

Mapped Test Cases: The result of the data collection phase is a set of mapped test cases, where each test case is mapped to the corresponding focal method. It is important to note that we discard test cases for which we were not able to identify the focal method using our heuristics. We designed these heuristics to be based on testing best practices, and obtain a correct mapping with very high confidence. This allows us to train our model on test cases that follow best practices, and likely excluding test cases that have been automatically generated.

We collect a total of 631,131 mapped test case pairs. We remove duplicate pairs and split the dataset in train (80% - 462,093 pairs), validation (10% - 57,761 pairs), and test (10% - 57,755 pairs) sets. The set of mapped test cases will be used to train our model to generate a test case given the focal method. We publicly release the dataset METHODS2TEST [23].

B. BART Transformer

ATHENATEST is based on a BART transformer model. BART [25] is a denoising autoencoder which utilizes the standard sequence-to-sequence transformer architecture from [26], substituting ReLUs with GeLU activation functions.

We select the BART model architecture because it facilitates finetuning for the downstream translation task of test case generation, providing a more advanced set of noising transformations, which include token masking, token deletion, infilling and statement permutation. The model is pretrained by corrupting documents and optimizing the cross-entropy loss between the decoder's output and the original input sequence.

We pretrain the BART large model architecture, which has 12 encoder layers and 12 decoder layers. The model is trained in mixed-precision, using Adam stochastic optimization procedure with $\epsilon = 10^{-6}$, and $\beta_1 = 0.9$, $\beta_2 = 0.98$ optimizer parameters; we apply inverse square root learning rate schedule with the base learning rate of 0.0001, a warmup period of 5000 update steps, and local gradient accumulation with a frequency of 4 update steps.

C. Pretraining

We employ two pretraining stages: English Pretraining, where we perform semi-supervised pretraining on a large corpus of English text, and Code Pretraining, where the model is pretrained on Java source code.

English Pretraining: In this stage we pretrain a model in a semi-supervised fashion on a large corpus of English text, with the goal of learning semantic and statistical properties of natural language. The pretraining is performed for 40 epochs on 160GB of English text extracted from books, Wikipedia, and news articles [27].

BART is trained in an unsupervised manner. Given corrupted text, its objective is to reconstruct the original text. The particular type of noise used in this work involves masking 30% of all tokens, with masks covering spans of tokens with lengths following a Poisson distribution parameterized by $\lambda = 3$, as well as permuting all sentences.

Code Pretraining: In this stage we pretrain a model on source code corpus written in Java language, with the goal of learning syntax and properties of source code.

We collect this code corpus dataset by crawling all public, non-fork Java repositories on GitHub with at least 50 stars. We then deduplicate at the file-level using a hash function. After filtering for permissive licenses and filtering out based on heuristics like the fraction of non-ASCII characters, we are left with 25GB of training data from the 26,000 repositories. For pretraining validation, we use the 239 test Java repositories from the CodeSearchNet [28], which comprise 600MB.

A similar pretraining strategy to English pretraining is employed. The source code files are corrupted by deleting 20% of all tokens independently and rotating half of all documents. This pretraining is performed for 10 epochs.

D. Finetuning

In this stage we finetune a model on the task of generating unit test cases for a given method. Specifically, we represent this task as a *translation* task, where the source is a focal method (*i.e.,* the method we would like to test), and the target is the corresponding test case originally written by a software developer.

The finetuning training is performed using the collected mapped test cases (Sec. II-A), where a mapped test case mtc_i can be seen as a pair $mtc_i = \{tc_i, fm_i\}$ comprising the test case tc_i and the corresponding focal method fm_i. The finetuning process is a translation task, with a training objective to learn the mapping $fm_i \rightarrow tc_i$ as a conditional probability $P(tc_i|fm_i)$.

During training, we use the cross entropy loss and the Adam optimizer, monitoring the loss on the validation set for early stopping. We use shared vocabulary embeddings between Encoder and Decoder for optimization reasons [26], [29] and because our input and output language is the same (*i.e.,* Java source code).

E. Model Variants

At the end of these stages, we obtain four different variants of the model, based on the level of pretraining performed:

- *BART_Scratch*: a model which has not been pretrained on any corpus but directly finetuned on the test case generation task.
- *BART_English*: a model which has been pretrained on the English corpus and then finetuned for the test case generation task.
- *BART_Code*: a model pretrained on the source code corpus, then finetuned on the test case generation task.

- *BART_English+Code*: a model pretrained first on English and further pretrained on source code corpus, then finetuned on the test case generation task.

III. Experimental Design

The goal of our empirical study is to determine if our approach can generate accurate and useful unit test case given a method. Our experiments aim at answering the research questions described in the following paragraphs.

RQ$_1$: Can our models learn to generate Unit Test Cases? To address this research question we investigate whether our models can learn to generate unit test cases as a translation task from the method under test. With this aim, we analyze several metrics related to the learning and generation process, and compare the model variants in order to select the best model.

Learning - Validation Loss: We begin by observing the validation loss during model training. A low validation loss means the model is effectively learning meaningful representations during training and is able to *generalize* how to generate test cases on a different set of input methods (*i.e.,* validation set). Specifically, we analyze three key metrics: (i) the initial validation loss during finetuning, which indicates the impact of the pretraining process; (ii) the best validation loss, which highlights the model achieving the best performance; (iii) the number of steps needed to reach the best validation loss, as a measure of how fast the finetuning process converges.

Translation - BLEU Score: We test the best checkpoints (*i.e.,* model checkpoint with lowest validation loss after training) of the model variants on the test set. Specifically, we test the models using a beam width of five, and compute the BLEU score of the predicted translations, comparing each translation to the target test case wrote by the developer. We rely on the BLEU score since it represents a common metric used for evaluating neural machine translation tasks. We do not compute additional translation metrics (*i.e.,* ROUGE) since the translation quality is not the focus of this paper, but only a proxy metric to select the best model. Further analyses will be investigating the quality of the generated test cases.

Generation - Perfect Predictions: We analyze the predictions of the models (*i.e.,* generated test cases) and compare them to target test case. We consider a prediction to be a perfect prediction if it matches tokens-by-tokens the target test case. Specifically, we compute the top-5 accuracy using the top-5 predictions generated by each of the models. Note that this analysis shall be considered only as a preliminary investigation, since there could be many non-perfect predictions that represent correct and effective test cases.

At the end of this research question we select the model variant that achieves the best performances. This model will be the core of AthenaTest, and further analyzed in the following research questions.

RQ$_2$: What is quality of the generated Test Cases? In this research question we further analyze the test cases generated by the model selected in RQ$_1$. The focus of this analysis is to scrutinize the generated model's predictions looking for specific properties that unit test cases should have.

Syntactic Correctness: We begin by verifying that the sequence of tokens generated by the model represents a syntactically correct source code method conforming to the Java specifications. To this aim, we parse all the predictions generated by the model using a Java parser, which determines the syntactic correctness.

Testing APIs: For a method to be considered as a test case, it needs to exhibit some basic properties, such as:

- *Test Annotation*: the test case should declare the `@Test` annotation.
- *Focal Method Invocation*: to properly test a focal method, the test case should invoke the focal method.
- *Testing APIs*: the test case should check the proper behavior of the focal method using testing APIs, such as assert statements and mocking methods. Specifically, we consider two testing framework APIs: JUnit Assert APIs (*e.g.,* `assertTrue`, `assertEqual`) as well as the Mockito Framework APIs (*e.g.,* `mock`, `verify`). We chose these testing framework for their popularity and applicability in many different contexts and domains. We plan to incorporate more domain-specific testing frameworks, such as Selenium [30] or REST Assured [31] in future work.

We check the compliance to these properties using a Java parser, extracting annotations and method calls. We also compare the distribution of testing APIs between the original test cases and the ones generated by the model.

RQ$_3$: How does our approach compare to EvoSuite and GPT-3? The goal of this research question is to provide a preliminary, quantitative, and qualitative comparison between the test cases generated by our model and those generated by two alternative approaches: EvoSuite and GPT-3. We chose these two approaches as representative of two different classes of techniques: (i) evolutionary-based automated test case generation; (ii) transformer-based language models.

EvoSuite: EvoSuite [6] is a widely known tool that automatically generates unit tests for Java software. EvoSuite uses an evolutionary algorithm to generate JUnit tests, targeting code coverage criteria. Specifically, it introduces mutants and iteratively generates assert statements to kill such mutants. During this process, EvoSuite minimizes the number of asserts while trying to maximize the number of detected mutants.

GPT-3: Generative Pre-trained Transformer 3 (GPT-3) is an autoregressive language model introduced by OpenAI [22]. GPT-3 is a transformer decoder-only architecture having 175 billion trainable parameters. It has been pre-trained on the Common Crawl dataset [32] constituting nearly a trillion words, an expanded version of the WebText [33] dataset, two internet-based books corpora (Books1 and Books2), and English-language Wikipedia. GPT-3 has demonstrated an impressive task-agnostic few-shot performance on text generation, translation and question-answering, as well as cloze tasks. The few-shot learning assumes an extended context supplied to the model during inference as a task description, and requires no gradient updates.

Experiment's Design: In this experiment, we aim at assessing two main qualities of the generated test cases: (i) *correctness* – tests that accurately assert the behavior of the focal method; (ii) *code coverage* – number of lines and conditions covered by the test cases.

For this comparison we select a small but reproducible testbed using defects4j [34]. We rely on defects4j since it provides a reliable infrastructure to generate, compile, execute, and evaluate test cases. Specifically, we select Lang-1-f, which represents the fixed version of the first bug in the defects4j collection belonging to the project Apache Commons Lang [35]. Note that this project is not included in our finetuning dataset used to train the model. We generate unit test cases for all the public methods of the class impacted by the bug, `NumberUtils`, using our model, EvoSuite, and GPT-3. Next, we compile and execute the test cases and manually assess their correctness. Specifically, to be defined as correct, the test case needs not only to be able to execute and pass, but also requires to specify at least one assert that is semantically accurate w.r.t the focal method. Subsequently, we compute test coverage using defects4j (which, in turn, relies on Cobertura [36]) singularly for each unit test case generated by the three approaches.

EvoSuite - Generation: To generate test cases with EvoSuite, we use the defects4j built-in command `gen_tests.pl -g evosuite -p Lang -v 1f`. This command invokes EvoSuite test generation on the first fixed revision of Lang, which will generate test cases for the class affected by the bug (*i.e.,* `NumberUtils`). We let EvoSuite generate test cases for 500 seconds ($\sim$ 8 minutes). Then, we test every unit test case generated and select the best test case for each focal method.

GPT-3 - Generation: To generate test cases with GPT-3 we rely on few-shot learning. Specifically, we provide two examples of input focal method and corresponding test case taken from the training set, then feed one of the public methods in the `NumberUtils` class, and expect GPT-3 to answer with the corresponding test case.

We use the OpenAI APIs and `davinci-msft` serving endpoint to perform inference on the model. We experiment with two different sets of prompts (*i.e.,* focal methods and test cases) from the supervised training set for our target downstream task as conditioning, varying the sampling temperature parameter from 0.1 to 0.9 with 0.1 increments (*i.e.,* the higher the temperature, the more risky/creative are the outputs). We generate ten candidate output sequences for each focal method, selecting the best test case for each focal method. Note, we fall back to one-shot learning if the examples and the current focal method exceed the maximum sequence length for GPT-3 (*i.e.,* 2048 tokens), which happened only once.

ATHENATEST- *Generation:* We feed each focal method within the class to our trained model. We sample the top-10 predictions of the model for each focal method, and select the single best prediction for each focal method. When we construct the input to the model, we also add the class information (`public class <focal_class> {<focal_method>}`), to inform the model about the class that contains the focal method. Note, this is exactly the same input we used for GPT-3, except for the task description that GPT-3 needs.

We are aware that this represents only a small-scale preliminary evaluation, however, given the significant manual effort assessing the correctness, we believe this is an important first step. We discuss this in the threats to validity section.

RQ$_4$: Do developers prefer ATHENATEST's test cases over EvoSuite'? In this research question we aim at analyzing the developers' perspective and preferences regarding test cases. In particular, we are interested in the developers' view of different aspects of test cases, such as readability, understandability, and testing effectiveness.

To this aim, we designed a survey with developers where we show them a focal method under test and two alternative test cases: one generated with ATHENATEST, and the other with EvoSuite. We then ask the developers three questions, soliciting them to rely on their personal preferences when evaluating these factors:

- Q$_1$: Which test case is more readable and understandable?
- Q$_2$: Which test case is testing the method more appropriately?
- Q$_3$: Which test case would you prefer to have in your project?

The first two questions are designed to evaluate two different factors, namely understandability and testing effectiveness of the test cases. These questions can be answered by choosing: (i) Test Case A; (ii) Test Case B; (ii) Equally (*i.e.,* same degree of understandability and testing effectiveness). The third question is designed to break possible ties, and ask for overall preference between the two test cases (choose A or B). This will provide some clues on whether developers prefer one factor over the other.

The survey consists of two background questions, asking about Java and JUnit experience, followed by 14 testing scenarios to review. Each scenario is formed by a focal method, and two test cases (one from ATHENATEST, the other from EvoSuite), randomly assigned with label A or B. The 14 focal methods have been selected from the experiment in RQ$_3$ and all the test cases selected are compilable and correct. We simply instruct the developer to answer the questions based on their personal preferences, without providing any clues on which test case was generated by our approach.

IV. Experimental Results

In this section we report and discuss the results of our empirical study.

RQ$_1$: Can our models learn to generate Unit Test Cases?

Learning - Validation Loss: Figure 2 shows the cross-entropy loss on the validation set during training for the four model variations. We note a substantial gap between the model without pretraining (*BART_Scratch*) compared to the models with English (*BART_English*), source code (*BART_Java*) and both (*BART_English+Java*) pretraining. Comparing the English only and the English+Java models, the additional pretraining on source code has three evident effects: (i) lower initial loss

Fig. 2. Validation Loss during finetuning

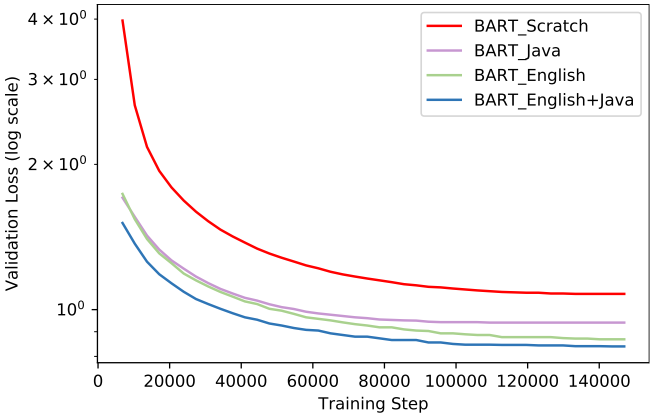


TABLE I
Intrinsic Evaluation Metrics

Metric	*BART_Scratch*	*BART_Java*	*BART_English*	*BART_English+Java*
BLEU4	33.33	34.04	34.57	**34.87**
Validation Loss	1.08	0.94	0.87	**0.84**

(1.74 versus 1.51); (ii) lower best loss (0.87 versus 0.84); (iii) faster convergence (∼20k training steps earlier).

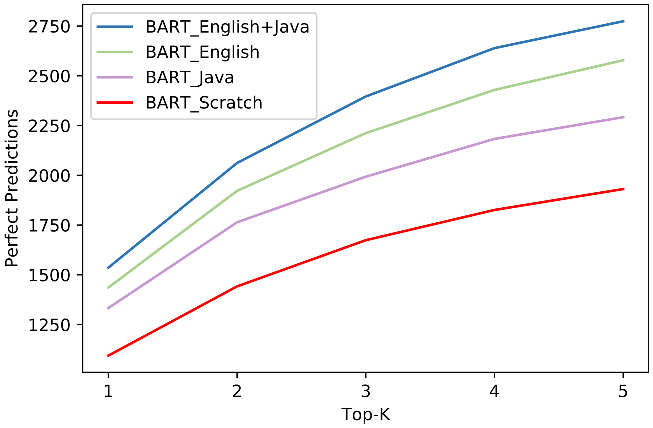
Translation - BLEU Score: Table I reports the BLEU score obtained by the models on the test set. Also in this case, the model with both English and code pretraining (*BART_English+Java*) achieves the best score (34.87). The differences in the validation loss are also confirmed in the gaps of BLEU score among the model variants.

Generation - Perfect Predictions: Figure 3 reports the top-k accuracy for our four model variations. The x-axis represents the k value, ranging from 1 to 5, indicating the number of predictions considered for each input (*i.e.,* for $k = 1$, only the top single prediction is considered), while the y-axis indicates the number of perfect predictions in the test set. The model *BART_English+Java* achieves the best results, with more than 1,500 perfect predictions when only a single translation is suggested. The number of perfect predictions increases when more candidates are generated, reaching 2,750 perfect test cases when considering the top-5 candidates generated by the model. These values represent between 2.7% and 4.9% of the test set. It is important to note that this represents a significant underestimate of the potentially correct test cases generated by our models.

Finally, we investigated whether these perfect predictions are novel or simple repetition of the test cases seen in the training set. We found that more than 85% of the perfect predictions are indeed novel test cases. Overall, when considering all the predictions (not only the perfect ones), only ∼5% of the models' predictions are found in the training set. This result shows that the models are able to generalize to different focal methods, and not simply memorizing test cases.

Thus, we select the model *BART_English+Java* as the core of AthenaTest.

Fig. 3. Top-K Accuracy Results



Summary for RQ$_1$. Pretraining on both English and source code has a significant positive effect on the task of generating Test Cases. The model *BART_English+Java* achieves the best validation loss, BLEU score, and top-k accuracy.

RQ$_2$: What is quality of the generated Test Cases?

Syntactic Correctness: The model generates syntactically correct Java methods for 84% of the top predictions in the test set. We manually investigated the reasons behind the syntactic errors for some of the predictions, and found that they were mostly due to truncated sequences when generating long test cases. We devised a simple approach that attempts to recover these predictions by deleting the last truncated statement, and adding a closing parenthesis. With this simple approach, the syntactic correctness reaches 95%. These results show that our approach is able to generate syntactically correct Java methods in most of the cases, and with simple post-processing it achieves extremely high levels of correctness. Furthermore, an incorrect prediction could be replaced with another prediction generated by the model (on the same focal method) using beam search or sampling.

Testing APIs: The model generates methods that declare the `@Test` annotation in 99.99% of the cases, correctly learning the JUnit standard for test cases. Furthermore, 94.9% of the generated test cases invoke the correct focal method which is supposed to test.

Figure 4 shows the distribution of testing API calls within each test cases in the test set, both for the original test cases and for the predictions of the model. From the boxplot we can notice that the two distributions have the same quartiles with, on median, one testing API call in each test case. Note that outliers are not reported in this figure. The mean (shown as a red triangle) indicates that the original test cases tend to contain slightly more testing APIs compared to the ones generated by the model.

Figure 5 shows the breakdown distribution of the top-16 testing API found in the test set. These include JUnit APIs such as `assertEquals` and Mockito APIs such as `mock` and `verify`. The plot clearly shows that the generated test cases

Fig. 4. Testing APIs Distribution

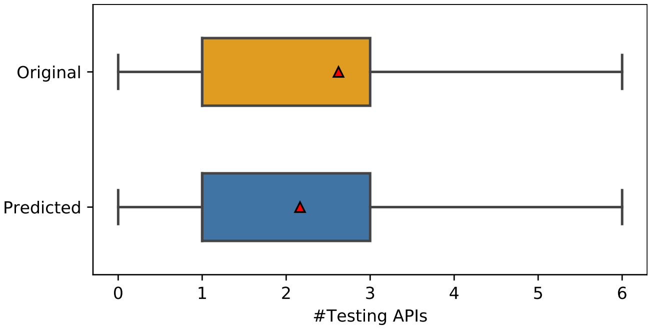
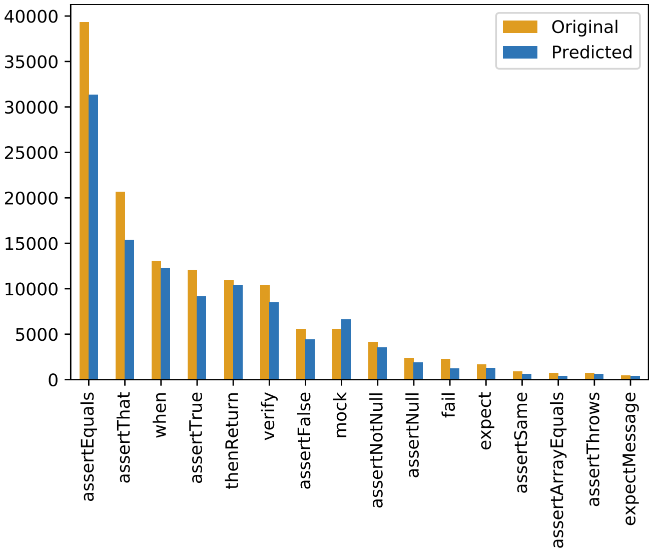


Fig. 5. Testing APIs Breakdown Distribution



invoke a variety of different testing APIs, closely following the distribution of the original test cases. However, we do observe a gap between the number of APIs in the original and predicted test cases. In our future work we plan to incorporate techniques to augment the number of assert statements in the test cases.

We conclude this research questions with qualitative examples of test cases generated by the model. Figure 6 shows the focal method `review` belonging to the class `CSConference`. This example was written by one of the authors with the goal of demonstrating the generalizability of our model on *novel* and *unusual* input methods (*i.e.,* not belonging to the training distribution). Figure 6 also shows three examples of generated test cases selected from the top-10 predictions of the model. In the test case #1 the model creates a new instance of `Submission`, followed by an assert statement that contains the focal method invocation. The model generates a correct method invocation passing the correct number of arguments and types. The assert statement interestingly inspects the reason of the `Response` by matching the same string used within the focal method. In the test case #2 the model first creates a realistic instance of a paper submission, then it invokes the focal method saving its response, and finally asserts that the response is of type `Acceptance`. Finally, in test case #3 the model relies on mocking APIs from Mockito to mock the `Submission` class, as well as using assert statements on the response. These examples showcase the ability of our model in generating realistic test cases, which resemble test cases written by developers.

Summary for RQ$_2$. Our approach generates syntactically correct test cases that conform to the Unit Test Case standards. The generated test cases invoke a variety of Testing APIs, following a similar distribution to the original test cases.

Focal Method

```
public class CSConference {
public static Response review(Submission paper, List<Reviewer> revs) {
    if(paper.citations < 50){
        return new Rejection("Discuss related works");
    }
    if(!paper.hasComparison()){
        return new Rejection("Compare with state of the art");
    }
    return new Acceptance("Congrats!");
}}
```

Model Generated Test Case #1 - Asserts

```
 @Test public void conference() {
    Submission paper = new Submission();
    assertTrue(CSConference.review(paper, new ArrayList<Reviewer>()).getReason()
        .equalsIgnoreCase("Discuss related works"));
 }
```

Model Generated Test Case #2 - Asserts

```
 @Test public void mentionedStates() {
    Submission paper = new Submission("Yang et al. 2012",
        "Elements of Information Sciences", 61, 3);
    Response response = CSConference.review(paper, new ArrayList<Reviewer>());
    assertTrue(response instanceof Acceptance);
}
```

Model Generated Test Case #3 - Mockito & Asserts

```
@Test public void reference() {
    Submission paper = mock(Submission.class);
    when(paper.getNumberOfCitations()).thenReturn(3);
    when(paper.citations()).thenReturn(60);
    when(paper.hasComparison()).thenReturn(true);
    Response response = Conference.review(paper, new ArrayList<Reviewer>());
    assertEquals(new Acceptance("Congrats!"), response);
    when(paper.citations()).thenReturn(99);
    response = Conference.review(paper, new ArrayList<Reviewer>());
    assertEquals(new Rejection("Discuss related works"), response);
}
```

Fig. 6. Examples of Generated Test Cases

RQ$_3$: How does our approach compare to EvoSuite and GPT-3? Table II reports the results of our test coverage analysis comparing EvoSuite, GPT-3, and AthenaTest on the class `NumberUtils` of Lang-1-f. The table reports the absolute (and percentage) line and condition coverage at class-level, for each of the 18 unique public methods in the class (without considering overloading), marking in bold the best coverage value. From the results in Table II we can notice: (i) EvoSuite was able to successfully test all the methods; (ii) GPT-3 correctly tested only 6 out of 18 methods; (iii) AthenaTest generated correct test cases for all the methods except 2, while achieving the best coverage in most cases.

For GPT-3 we explored several sampling temperatures, and settled on the 0.5 value which appeared to provide good diversity of the samples while still generating realistic code. We found that, in most of the cases where GPT-3 was not able to generate a correct test case, it generated code that only invoked the focal method without correctly asserting its behavior. However, in those 6 cases reported in the table, we

TABLE II
TEST COVERAGE ANALYSIS

Focal Method	EvoSuite		GPT-3		ATHENATEST	
	Lines	Conditions	Lines	Conditions	Lines	Conditions
`toInt(String, int)`	21 (5.6%)	1 (0.3%)	-	-	**23 (6.1%)**	**2 (0.6%)**
`toLong(String, long)`	**20 (5.3%)**	**1 (0.3%)**	-	-	**20 (5.3%)**	**1 (0.3%)**
`toFloat(String, float)`	20 (5.3%)	1 (0.3%)	-	-	**22 (5.9%)**	1 (0.3%)
`toDouble(String, double)`	**20 (5.3%)**	**1 (0.3%)**	-	-	**20 (5.3%)**	**1 (0.3%)**
`toByte(String, byte)`	20 (5.3%)	1 (0.3%)	-	-	**23 (6.1%)**	**2 (0.6%)**
`toShort(String, short)`	20 (5.3%)	**1 (0.3%)**	-	-	**22 (5.9%)**	**1 (0.3%)**
`createFloat(String)`	20 (5.3%)	1 (0.3%)	-	-	**21 (5.6%)**	**2 (0.6%)**
`createDouble(String)`	20 (5.3%)	1 (0.3%)	-	-	**21 (5.6%)**	**2 (0.6%)**
`createInteger(String)`	**20 (5.3%)**	**1 (0.3%)**	-	-	-	-
`createLong(String)`	20 (5.3%)	1 (0.3%)	20 (5.3%)	1 (0.3%)	**21 (5.6%)**	**2 (0.6%)**
`createBigInteger(String)`	28 (7.5%)	**8 (2.4%)**	**30 (8.7%)**	7 (2.1%)	20 (5.3%)	1 (0.3%)
`createBigDecimal(String)`	**22 (5.9%)**	**3 (0.9%)**	-	-	**22 (5.9%)**	**3 (0.9%)**
`min(long[])`	**27 (7.2%)**	**6 (1.8%)**	26 (6.9%)	5 (1.5%)	22 (5.9%)	2 (0.6%)
`min(int, int, int)`	22 (5.9%)	**2 (0.6%)**	**23 (6.1%)**	**2 (0.6%)**	22 (5.9%)	**2 (0.6%)**
`max(float[])`	**28 (7.5%)**	**7 (2.1%)**	-	-	-	-
`max(byte, byte, byte)`	**23 (6.1%)**	**2 (0.6%)**	21 (5.6%)	**2 (0.6%)**	22 (5.9%)	**2 (0.6%)**
`isDigits(String)`	20 (5.3%)	1 (0.3%)	**23 (6.1%)**	**5 (1.5%)**	**23 (6.1%)**	**5 (1.5%)**
`isNumber(String)`	44 (11.7%)	29 (8.6%)	-	-	**51 (13.6%)**	**41 (12.1%)**

found the test cases to be correct and readable code, and sometimes also obtaining the best coverage. While GPT-3 achieved the lowest overall performances of the three, we would consider this still a positive result for GPT-3, given the fact that it was not finetuned on test case generation.

Regarding our approach, ATHENATEST was able to generate correct test cases for all the methods except 2. We found that in those two instances, while the approach was generating syntactically correct code, none of the top-10 predictions were correct test cases. For example, for the method `createInteger`, our approach generated the following assert: `assertEquals(0, NumberUtils.createInteger("0"));` which is an ambiguous reference to `assertEquals`. This could have been solved by casting the int value to Integer. Overall, the results indicate that ATHENATEST is able to generate correct test cases with adequate test coverage, often achieving better coverage than EvoSuite.

We now provide a qualitative comparison of the test cases generated by the three approaches. Figure 7 shows the generated test cases for the focal method `createFloat`. EvoSuite creates a test case that assert that the return value of the method is null, when providing a null string as input, covering the first condition in the focal method. GPT-3 creates a test case that simply invokes the focal method multiple times (limited in the figure), with correct arguments, but without asserting the correct behaviour of the method. ATHENATEST generated a test case that checks (i) the focal method correctly creates a the float 1.2; (ii) the focal method return null on a null string. Specifically, it covers both conditions of the focal method. We can also notice that the generated test case has a very idiomatic name `testCreateFloat` (similar to GPT-3), compared to EvoSuite's `test044`.

Figure 8 shows the test cases for the focal method `isDigits`. EvoSuite's test case checks whether the empty string is correctly identified as not being a numerical digit. GPT-3 accurately assert the behavior of the method by testing a string containing only digits (*e.g.,* "100") and one that contains a non-digit character (*e.g.,* "1a"). Our approach, generates a test case which tests six different strings, one of which only contains digits, while the other five contains also non-digit characters, for example a negative number (*e.g.,* "-123") as well as comma-separated numbers (*e.g.,* "1,2,3"). Interestingly, in this case, combining EvoSuite and ATHENATEST would lead to better overall code coverage.

Focal Method

```java
public static Float createFloat(final String str) {
        if (str == null) {
            return null;
        }
        return Float.valueOf(str);
}
```

EvoSuite

```java
@Test(timeout = 4000)
public void test044()  throws Throwable  {
   Float float0 = NumberUtils.createFloat((String) null);
   assertNull(float0);
}
```

GPT-3

```java
@Test public void testCreateFloat() {
    NumberUtils.createFloat("1.0");
    NumberUtils.createFloat("1.0f");
    NumberUtils.createFloat("1.0E+1");
    NumberUtils.createFloat("1.0E-1");
    ...
```

ATHENATEST

```java
@Test public void testCreateFloat() {
     assertEquals(new Float(1.2f), NumberUtils.createFloat("1.2"));
     assertNull(NumberUtils.createFloat(null));
 }
```

Fig. 7. Test Cases Generated for `createFloat`

Focal Method

```java
public static boolean isDigits(final String str) {
        if (StringUtils.isEmpty(str)) {
            return false;
        }
        for (int i = 0; i < str.length(); i++) {
            if (!Character.isDigit(str.charAt(i))) {
                return false;
            }
        }
        return true;
}
```

EvoSuite

```java
@Test(timeout = 4000)
public void test092()  throws Throwable  {
    boolean boolean0 = NumberUtils.isDigits("");
    assertFalse(boolean0);
}
```

GPT-3

```java
@Test public void testIsDigits() {
    assertTrue(NumberUtils.isDigits("100"));
    assertFalse(NumberUtils.isDigits("1a"));
}
```

ATHENATEST

```java
@Test public void isDigits() {
     assertTrue(NumberUtils.isDigits("123"));
     assertFalse(NumberUtils.isDigits("-123"));
     assertFalse(NumberUtils.isDigits("1a2"));
     assertFalse(NumberUtils.isDigits("a0b-2"));
     assertFalse(NumberUtils.isDigits("abcax"));
     assertFalse(NumberUtils.isDigits("1,2,3"));
}
```

Fig. 8. Test Cases Generated for `isDigits`

Compared to EvoSuite, our approach is able to generate correct test cases which obtain comparable (if not better) test coverage for most of the focal methods. ATHENATEST's test cases appear to be more readable and similar to developers-written code w.r.t. those generated by EvoSuite. Our approach outperforms GPT-3 in terms of number of correct test cases generated. While GPT-3 was not finetuned on the test case generation task, it is a substantially larger model (117 billion parameters) compared to our transformer-based model (450 million parameters).

Summary for RQ$_3$. Our approach generates test cases that accurately test the focal methods and obtain comparable test coverage w.r.t. EvoSuite, as well as outperforming GPT-3. These test cases appear to be similar to developer-written test cases with readable and understandable code.

RQ$_4$: Do developers prefer AthenaTest's test cases over EvoSuite'? We received responses from 12 Microsoft developers, none of them involved in this work. All the developers had Java experience (4 with one year or less, 7 with 1-3 years, 1 with 4 or more years), 8 of them claimed to have JUnit experience.

Figure 9 reports the answers to the three survey questions in a likert-style plot, where the y-axis represents the testing scenario instance, and the x-axis the number of responses for EvoSuite (in red, towards left), for AthenaTest (in blue, towards right), and neutral answer (middle green).

Regarding Q$_1$, we found that 61% of the responses favored AthenaTest's test cases in terms of readability and understandability, while in 29% of the cases the developers thought both test cases were equally readable, and only in 10% of the cases they preferred EvoSuite's.

For Q$_2$, 70% of the responses selected AthenaTest's test cases as testing the focal method more appropriately than EvoSuite's counterpart. In 12% of the cases they were deemed as equally appropriate, and only in 18% the developers preferred EvoSuite's test case.

Finally in Q$_3$, when asked to choose which test case they preferred overall, they overwhelmingly elected AthenaTest's test cases, in 82% of the cases, and only 18% EvoSuite.

Interestingly, we found that in 12 instances ($\sim$7%), developers picked one test case in Q$_1$ and the other test case in Q$_2$. A deep dive in these cases revealed that developers mostly preferred AthenaTest test cases in terms of readability, but EvoSuite in terms of testing effectiveness.

Summary for RQ$_4$. Developers prefer test cases generated by AthenaTest over those generated by EvoSuite, in terms of readability, understandability, and testing effectiveness.

V. Discussion & Future Work

Our preliminary evaluation shows encouraging results in many different aspects. Our approach is able to generate syntactically correct test cases that conform to the test case standards and invoke a variety of testing APIs. While further analyses should be performed, this preliminary evaluation shows that the generated test cases appear to be (i) *realistic* – similar to developer-written test cases; (ii) *accurate* – correctly asserting the expected behavior of a focal method; (iii) *human-readable* – readable and understandable code, with good variable and method names.

We believe this work represents a stepping stone towards a new category of automated test case generation tools, shifting away from coverage-guided approaches towards models that aim at code understanding. These learning approaches have the potential of generating natural test cases that better integrate with the existing code base, and do not appear like *machine-written* code.

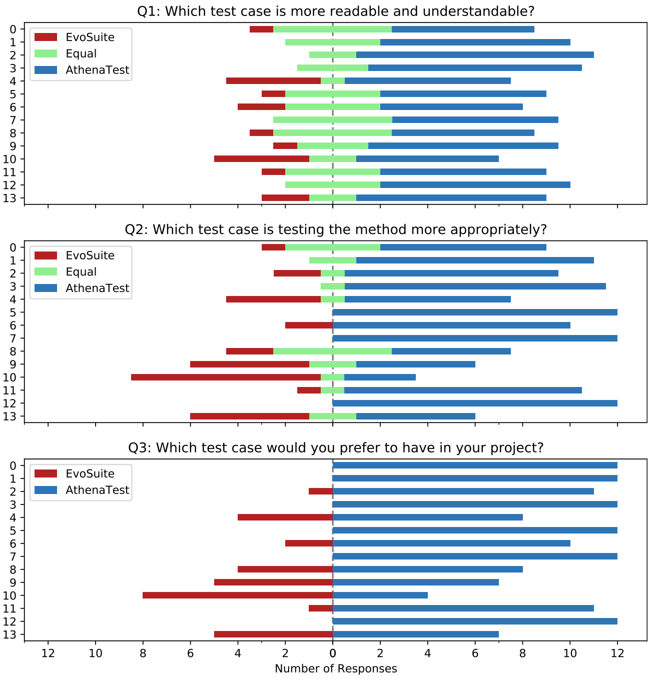


Fig. 9. Survey results with professional developers

During our manual investigation of the generate test cases, we also observed several weaknesses and pitfalls of the model, which we will discuss in this section. These weaknesses serve us as inspiration for future work, with the goal of improving our model.

A. Focal Class Context

When providing only the focal method as input, the model is forced to perform a series of reasonable guesses on the composition of the focal class. Specifically, the model needs to guess the correct way to instantiate the class' object, and potentially invoking auxiliary methods and attributes of the class, while testing the focal method. This results in unfeasible model's predictions due to incorrect object instantiation (*e.g.,* incorrect number, order, or type of arguments) and use of reasonable yet undefined methods or attributes within the focal class.

We aim to address this problem in the upcoming version of our approach by providing additional focal class context during training and inference of the model. Specifically, the input to the model will include, along with the focal method, also the following context:

- *Class constructor*: the class name and constructor (or lack there of, in case of singleton) will inform the model on the expected installation of the class;
- *Class Attributes*: the public class attributes will allow the model to generate test cases that can inspect and assert the value of the object's attributes;
- *Methods' Signatures*: the list of public methods' signatures in the focal class can be used by the model to set-up the

testing environment and inspect the result (*e.g.,* using getters and setters).

Furthermore, semi-supervised pretraining on the projects where the model will be used to generate test case, could help the model to familiarize with the code base and be more accurate when generating statements and method calls.

B. Testing Frameworks

Numerous testing frameworks are available for Java developers which aim at supporting domain-specific applications or different testing scenarios and methodologies. Our current approach does not take into consideration the specific testing framework used by the developer, thus could propose a test case using a different testing API which is not being used in the current project.

In our future work we plan to train our model to support multiple testing frameworks, and allow the developer to specify the particular testing APIs to be used. This could be achieved using control codes (*i.e.,* special reserved keywords) to inform the model about the particular testing APIs used in the test case, both during training and inference.

C. Deployment

Deployment of large neural models to production represents a major engineering challenge. In this section, we discuss the possible deployment scenario in Visual Studio Code IDE backed by the Azure cloud compute.

We propose to design the ATHENATEST system as a two-layer service, consisting of the server-side inference module and the client-side unit test case provider module. With the model size exceeding 100 MB, the cloud-based deployment is the only viable option, which also offers control over the hardware setup and can guarantee resource availability. Introducing the client-side unit test case provider module would allow to minimize the inference time for the best user experience. The server-side module is deployed as a containerized web application to Azure Kubernetes Service [37] listening on a HTTPS endpoint. It processes completion requests and returns the model output, which is implemented in PyTorch.

VI. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation and are mainly related to the measurements we performed. In our context, the threat arises by training our models on potentially noisy data, specifically, low quality test cases and incorrect mapping between focal methods and tests. We attempt to mitigate this threat by relying on safe and accurate heuristics to mine test cases and focal methods, following best practices.

Internal validity threats concern factors internal to our study that could influence our results. The performance of our approach depends on the hyperparameter configuration and pretraining process. We did not perform hyperparameter search since these large models require substantial training time, however, we reuse configurations suggested in the literature. We experiment with different pretraining stages and report the results of our experiments.

Threats to *external validity* concern the generalization of our findings. In this paper the threat arises in RQ_3, given the small-scale evaluation, we cannot claim generalizability of the results. We clearly state that this represents a preliminary evaluation and more experiments should be conducted to assess the quality of our approach. We also acknowledge the fact that additional analyses should be performed to evaluate the fault detection capability of the generated test cases. We are actively working on addressing these limitations in our continuing work.

VII. Related Work

Our work is related to several existing approaches in the area of automated software testing. In particular, there is a class of approaches that aims at generating tests cases, such as Evosuite [6], Randoop [7], and Agitar [8]. The main differentiating factor between these techniques and our approach is the learning component. ATHENATEST is based on transformer model which aims at learning, from developer-written test cases, the best practices on how to write readable and accurate test cases. On the other hand, most of the existing techniques in the literature rely on handcrafted rules or heuristics to generate test cases, optimizing towards code coverage.

Several existing works in the literature have proposed deep learning based approaches for software engineering tasks, such as code completion [17], automated patch generation [18], [19], comment generation [20], and many others [21]. While we share with these approaches the process of learning from examples, we also introduce significant novelty in this process. Specifically, we are among the first to train large, state-of-the-art sequence-to-sequence transformer models applied to software engineering tasks. Additionally, we pretrain these models on both English and source code showing the benefits of both pretrainings on the generation of test cases.

Our work is also related to a broad set of literature on transfer learning [32], unsupervised language model pretraining [38], [39], and denoising pretraining [25], [40], [41]. In this paper, we extend these ideas to source code as a language, combining English and source code pretraining modes, fine-tuning on a downstream translation task from the automated software engineering domain. We compare this approach to the task-agnostic few-short learning approach introduced in GPT-3 [22]. We find and discuss certain limitations of the few-shot learning approach as compared to finetuning using translation task.

VIII. Conclusion

In this paper we presented ATHENATEST, an approach that aims at generating unit test cases by learning from real-world, developer-written test cases. Our approach relies on a sequence-to-sequence transformer model which was pretrained both on English and Java source code, then finetuned on the task of generating test cases given a method under test. We train the model using a supervised parallel corpus of 630k test cases and corresponding focal methods in Java, which we publicly release as METHODS2TEST [23].

Our evaluation shows that ATHENATEST is able to generate syntactically correct test cases that invoke a variety of testing APIs. We compiled and executed these test cases, comparing them with EvoSuite and GPT-3, finding that we achieve comparable or better test coverage. Finally, in a study with professional developers, we found that they prefer ATHENATEST's test cases in terms of readability, understandability, and testing effectiveness.

REFERENCES

[1] M. Cohn, *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[2] "Junit," https://junit.org.

[3] "Apache jakarta cactus," http://jakarta.apache.org/cactus.

[4] "Testng," https://testng.org.

[5] "Mockito," https://site.mockito.org.

[6] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[7] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[8] Agitar, "Utilizing Fast Testing to Transform Java Development into an Agile, Quick Release, Low Risk Process," http://www.agitar.com/, 2020.

[9] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.

[10] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–3483.

[11] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2016, pp. 5–14.

[12] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 130–141.

[13] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[14] G. H. Pinto and S. R. Vergilio, "A multi-objective genetic algorithm to test data generation," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, vol. 1. IEEE, 2010, pp. 129–134.

[15] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[16] S. Shamshiri, "Automated unit test generation for evolving software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 1038–1041.

[17] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[18] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[19] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[20] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[21] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," *arXiv preprint arXiv:2002.05800*, 2020.

[22] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[23] Microsoft, "methods2test," https://github.com/microsoft/methods2test, 2020.

[24] "Tree-sitter," http://tree-sitter.github.io/tree-sitter.

[25] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[27] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[28] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[29] O. Press and L. Wolf, "Using the output embedding to improve language models," *CoRR*, vol. abs/1608.05859, 2016. [Online]. Available: http://arxiv.org/abs/1608.05859

[30] A. Bruns, A. Kornstadt, and D. Wichmann, "Web application tests with selenium," *IEEE software*, vol. 26, no. 5, pp. 88–91, 2009.

[31] "Rest assured," http://rest-assured.io, 2020.

[32] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2019.

[33] A. Gokaslan and V. Cohen, "Openwebtext corpus," http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[35] "Apache common lang," https://commons.apache.org/proper/commons-lang, 2020.

[36] "Cobertura," https://cobertura.github.io/cobertura, 2020.

[37] Microsoft, "Azure kubernetes service," https://azure.microsoft.com/en-us/services/kubernetes-service, 2020.

[38] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2018. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[39] ——, "Language models are unsupervised multitask learners," 2019.

[40] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[41] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692