CONTEST: A Unit Test Completion Benchmark featuring Context

Johannes Villmow, Jonas Depoix, Adrian Ulges
RheinMain University of Applied Sciences
Wiesbaden, Germany
{johannes.villmow, adrian.ulges}@hs-rm.de
jonas.depoix@web.de

Abstract

We introduce CONTEST, a benchmark for NLP-based unit test completion, the task of predicting a test's assert statements given its setup and focal method, i.e. the method to be tested. CONTEST is large-scale (with $365k$ datapoints). Besides the test code and tested code, it also features *context code* called by either. We found context to be crucial for accurately predicting assertions. We also introduce baselines based on transformer encoder-decoders, and study the effects of including syntactic information and context. Overall, our models achieve a BLEU score of 38.2, while only generating unparsable code in 1.92% of cases.

1 Introduction

Testing is commonly considered an important part of software development, but it tends to be neglected in practice, as developers find it rather time-consuming and tedious. This has motivated *automated* testing: Approaches such as EvoSuite (Campos et al., 2019) and Randoop (Pacheco and Ernst, 2007) can bootstrap tests with decent code coverage using static code analysis and evolutionary search. However, recent studies (Almasi et al., 2017; Shamshiri, 2015) have found the readability of those generated tests to be subpar, and have – more importantly – found that the above approaches struggle with producing "meaningful" checks that truly assert the code to behave as expected.

Therefore, recent approaches have tackled *AI-based test completion* as a research challenge for NLP-based programming, using encoder-decoder models with bidirectional recurrent networks (Watson et al., 2020) or pre-trained transformers (Tufano et al., 2020b). These models take the test's setup code, together with the targeted method call (focal method) as input and predict the test's assertion. Since this assertion is arguably the test's part which requires the most understanding of the target method, it is also the part where heuristic approaches struggle the most. Here, an AI-based approach can fill this gap and provide the most valuable addition to existing solutions.

We extend this line of research by introducing CONTEST, a new contextual benchmark for automated test completion. CONTEST is based on Github data. Each of its datapoints features a test method, linked with the tested focal method using a fuzzy name matching. Additionally, the test is split into segments using heuristics and static code analysis: We provide the focal method, test setup code executed before the assertions, context methods called within the setup code, and context methods called within the focal method. A manual inspection of samples estimates its ground truth's accuracy of matching the correct focal method at 94%, but we even found the falsely matched methods to be relevant to the test. Summarizing, CONTEST offers the following benefits[1]:

- **Context**: CONTEST includes not only the test code and focal method, but also *context code* called by either of them, including the test setup and recursively called methods from the test file or tested file. Arguably, this context is crucial for fully comprehending a test. Correspondingly, we found it to strongly improve accuracy.
- **Scale**: With $365k$ high-quality datapoints for test completion, CONTEST is, to the best of our knowledge, the largest test completion dataset to date.
- **Rigorous evaluation**: When building CON-

[1]The dataset is available at https://github.com/lavis-nlp/ConTest

Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pages 17–25

TEST, we have carefully avoided bias towards simple cases. Besides common metrics such as BLEU and ROUGE, CONTEST also estimates a code's unparsable rate in an accompanied evaluation package. Finally, we also provide a *project-based split*, which enforces test completion models to generalize to projects unseen in training, a setup which we found to be particularly challenging.

- **Strong baselines**: We provide Transformer-based baseline experiments. These include (1) enriching the input source code using Abstract Syntax Trees (AST), which we found to yield improvements by 4.1 BLEU points, and (2) adding context methods called by either test code or tested code (improvements by 10.1 BLEU points). Our experiments also show that the model struggles to generalize knowledge it has gathered from tests within the same project to tests in other projects (-21 BLEU). Nonetheless, our results indicate that automated test completion is an interesting direction for future research.

2 Related Work

Our work targets the fields of automated software testing and natural language processing. Automated software testing can be divided into test generation approaches, aiming to generate the *complete* test (Tufano et al., 2020a) and test *completion* approaches, aiming to generate meaningful assert statements (Watson et al., 2020). Test completion has been tackled by rule based approaches such as Agitar (Belhumeur et al., 2004), Randoop (Pacheco and Ernst, 2007), and EvoSuite (Campos et al., 2019), which can bootstrap tests with decent code coverage using static code analysis and evolutionary search. However, recent studies (Almasi et al., 2017) have found that these approaches struggle with producing "meaningful" checks that truly assert the code to behave as expected. Furthermore these approaches require handcrafted rules in order to generate assert statements.

Recently, the focus has shifted to approach automatic software testing using natural language processing (NLP) methods. White and Krinke (2018) employ a RNN-based neural machine translation system to generate complete tests, while more recent work (Tufano et al., 2020a) adapts pre-trained transformer models such as BART (Lewis et al., 2019). This and most recent work builds on transformers (Vaswani et al., 2017), which have become ubiquitous in natural language processing for sequence-to-sequence tasks. However, the authors report that the model often had difficulties to correctly initialize the object under test, as it was lacking the context information to do so. Our benchmark CONTEST includes such context information.

Recently, test completion has been tackled with machine translation methods by Watson et al. (2020). The authors propose an RNN-based approach to assert generation called ATLAS, along with a dataset of the same name. The ATLAS dataset consists of 158,096 tests paired with single line assert statements and is, to the best of the authors knowledge, the only available dataset for test completion available. Note that the ATLAS dataset is less than half the size of CONTEST, does not contain any context information, does not provide a project-wise split and in contrast to CONTEST contains only single line assertions. ATLAS has been used in recent studies of Tufano et al. (2020b), where the authors pre-train the transformer-based BART model on a large set of English texts and code artifacts and subsequently fine-tune it to generate assert statements. Applying their model on our dataset is an interesting direction for future research.

3 Dataset

Java has been the most used programming language in 2020, and its unit test framework JUnit has become a widely adopted industry standard (Poirier, 2018). Driven by the vast amount of open-source projects available on GitHub that employ JUnit as a testing framework, we chose to build CONTEST by scraping JUnit projects from GitHub. By limiting the projects to only one testing framework the test structure will stay more consistent throughout the dataset. Furthermore, as JUnit requires annotating tests with `@Test`, such methods can be located easily.

3.1 Data collection

We utilize the publicly available Github BigQuery Dataset (Hoffa, 2016), containing a snapshot with more than 3TB data of 2.8 million open source GitHub repositories[2]. Using BigQuery (Fernandes and Bernardino, 2015), we filter relevant projects

[2]We found using the GitHub API to be infeasible due to its rate limited of 5,000 requests per hour.

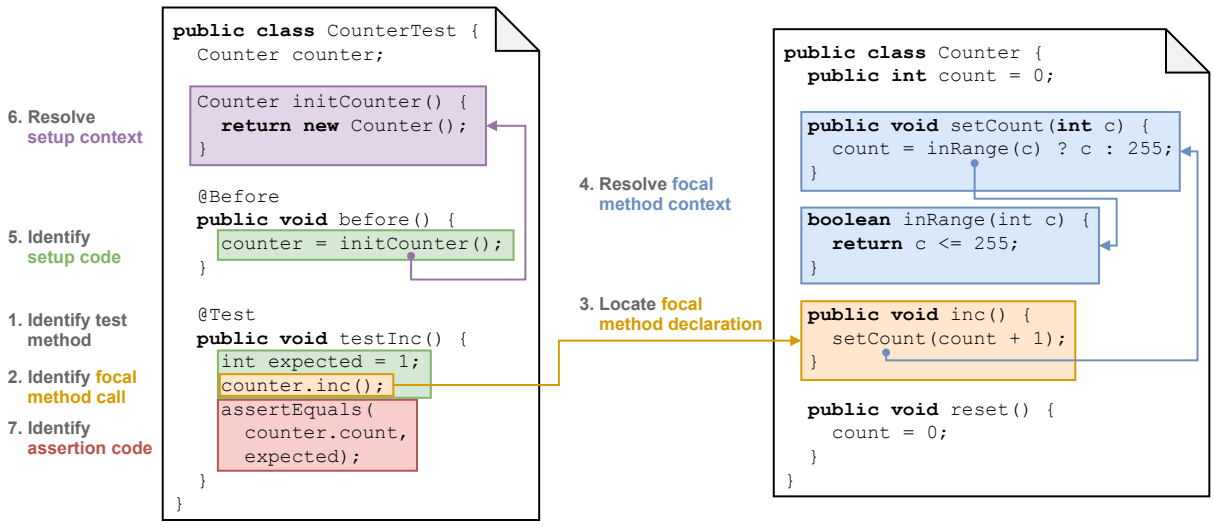

Figure 1: Components of CONTEST and steps taken to build the dataset. The task is to generate assertions (red box) given the partial test and its context. For each test we extract up to four parts: (1) the setup code of the test (green) and also include global JUnit setup code (i.e. the initialization of the `counter`), (2) code of the focal method (orange) matched using fuzzy string matching, and the source code of all additional methods that are called (recursively) in any of the two files (3) in the test setup context (purple, i.e. `initCounter`) or (4) in the focal method (blue, i.e. `setCount`, `inRange`).

by ensuring there is at least one `.java` file that contains the substring `org.junit`, an indication of an import from the JUnit package. Even though GitHub stores large amounts of code, duplicates are a major part of it (Lopes et al., 2017). We therefore exclude forks from the dataset to prevent duplicates, unless they have more than 20 GitHub stars, indicating them being different enough from the original repository to be relevant. We employ additional per sample deduplication in a later stage to further reduce duplicates. Finally, test methods are identified by matching the `@Test` annotation.

3.2 Identifying and locating focal methods

For each test method, we need to identify the test method's so-called *focal method*, i.e. the method that will be tested. Commonly, this method's name (e.g., `inc()`) is similar to the test method's name (e.g., `testInc()`). Therefore, we identify the focal method by ranking method calls within the body of the test method (referred to as *candidates*) by their token similarity to the test's method name or the test's class name[3]: We tokenize the name of the test, the name of the test class and the candidate method's names (from which we exclude JUnit assertions) by splitting them on camel and snake case into sets of tokens. We then lowercase the tokens and remove the token `test`. This way identifiers like `createAllUsers`, `create_all_users`, `CREATE_ALL_USERS`, and `test_createAll-Users` all result in the tokens `create`, `all` and `users`. We compute the following similarity based on two sets of tokens:

$$sim(T, C) := \frac{|T \cap C|}{|T|} \quad (1)$$

with T, C being sets of tokens. Let T_{name}, T_{class} be the set of tokens in the test's name/class and C_i the set of tokens in a candidate i's method name. Every unique candidate is first ranked by computing $sim(T_{name}, C_i)$ and the highest ranked candidate with a score greater than a threshold δ=0.1 chosen as the focal method. If there are multiple candidates with the same score or the score is below δ, we repeat the ranking process with T_{class}. If this process fails we consider the test as unresolvable and drop it from the dataset.

Knowing the name of the focal method, its declaration is located using JavaParser's TypeSolver feature (van Bruggen et al., 2020), which infers the type of a given AST node by analyzing surrounding nodes and the ASTs of imported modules.

In a manual evaluation on 100 randomly selected samples of our dataset we found that 94% were correctly matched and that the false positives were

[3]When matching the test name with all candidates fails, we compare the candidates with the test's class name, as sometimes tests are named by a test scenario (e.g. `testMultipleCalls` could be a test method of class `TestInc`).

also relevant to the test.

3.3 Identifying setup and assertion code

Next, we split the test method into *setup* vs. *assertion code* (green vs. red in Figure 1). Contrary to ATLAS (Watson et al., 2020), we do not only consider calls to JUnit assertion methods (like `assertEquals`) assertion code, but also the code leading up to it. This code could for example contain variable initialization or a surrounding `for`-loop, which is often essential to the assertion logic. To achieve this, all lines of code *following* the *last* focal method call (there can be multiple calls) are considered assertion code, while the preceding code lines are considered setup code.

The code line containing the last focal method call itself is considered part of the setup code (see Figure 1), unless the call happens inside a JUnit assert method (e.g., `assertEquals(counter.inc(),17)`). In this case, it is considered part of the assertion code.

In JUnit the `@Before` or `@BeforeEach` annotation (depending on the JUnit version) declares that code inside this methods should be executed before every single test in that class, or with `@BeforeClass`/`@BeforeAll` to be executed only once before all tests. Therefore, methods annotated as such are also considered part of the setup code in CONTEST.

3.4 Resolving contexts

The most notable novelty CONTEST provides is the addition of the *setup context* (purple in Figure 1) and *focal method context* (blue). We consider a method as part of such a context and thereby relevant to the task, if it is called during setup or in the focal method and belongs to any of the two java classes. Additionally, we consider methods called within the context methods as part of the context, thereby defining it recursively. We resolve those context methods using JavaParser's previously mentioned TypeSolver feature.

Since crucial parts of a focal method's logic may be outsourced to its context, we argue that adding the context of the setup code and focal method to the input of a test completion model is crucial to improve complex focal method understanding. This assumption is also backed by previous work reporting failed predictions due to missing context information (Tufano et al., 2020a).

3.5 Dataset size

Using the approach described in the previous sections 99,015 repositories were downloaded, with a total compressed size of 535GB. In these repositories 6,040,446 test methods in 1,127,415 test classes were found. Of those test methods 2,182,225 have successfully been mapped to their target method in one of 430,934 target classes. On the other hand, 3,858,221 tests could not be mapped, as no target method was found using the heuristics and resolving described in Section 3.2. Of the successfully mapped tests the setup and assertion code was identified for 1,336,360.

Another important distinction in the data is made by the location of the focal method calls. As having focal methods calls within the assertion code would require a model trained to generate assertion to also predict focal method calls, we limit CONTEST to tests were focal method calls only occur within the setup code, thereby limiting it to 845,497 datapoints[4]. After removing duplicates and datapoints with excessively large contexts, the final dataset contains 365,450 samples.

Following common practice (Watson et al., 2020; Tufano et al., 2020a; White and Krinke, 2018), we release CONTEST alongside the data split we used to randomly distribute the dataset into training (80%), validation (10%) and test (10%) data. However, when randomly distributing datapoints into splits, the training and test split are likely to contain tests from the same project, possibly even testing the same class or method. Therefore, we believe it must be carefully investigated whether models trained using these splits exploit information encountered during training, without generalizing beyond projects. To do so, CONTEST contains another project-based split alongside the regular split, where no project has tests in more than one split.

Split	Samples	
	Random	**Project-based**
Train	292,360	290,190
Validation	36,545	38,098
Test	36,545	37,162
Total	365,450	365,450

Table 1: Sizes of the splits in CONTEST

[4]We offer to make the unfiltered version available upon request.

3.6 Evaluation Protocol

Alongside the dataset, we release an evaluation package that can be used to evaluate new models. The evaluation protocol includes tokenization of the predictions using *javalang*. We consider only parsable predictions to be valid. The parsed token sequence is then evaluated using BLEU and ROUGE, whereas we adjust the score $s \in [0, 1]$ of each metric to take unparsable predictions into account:

$$adj(s) := s \cdot (1 - r_{up}) \tag{2}$$

where $r_{up} \in [0, 1]$ is the *unparsable rate*. This scales the metric by the ratio of parsable samples. By scaling BLEU/ROUGE in this manner, a similar effect is achieved to having an unparsable prediction score of 0, i.e. scores are penalized if the model is not able to consistently generate parsable code.

4 Approach

Our approach towards test completion is illustrated in Figure 2. Similar to prior successful work on test completion and test generation (Tufano et al., 2020a), our model utilizes a transformer encoder-decoder architecture. Additionally, we utilize *context code* that is called by either test method or focus method as additional input. Also, we investigate two fundamental ways of encoding source code: (1) in form of linearized abstract syntax trees (ASTs), and (2) a tokenized version of the source code.

Our model pipeline is illustrated in Figure 2: We process two source files containing the test and the tested code by parsing relevant code parts into ASTs, which are combined to the so-called *source tree* describing the whole test. This tree is fed into an encoder-decoder transformer, which produces a assertion's linearized AST. This is compared to the target AST using a the cross entropy loss function.

4.1 Preprocessing

Our model's input consists of four different parts: (1) the setup code of the test (green in Figure 2), (2) code of the focal method which is tested (orange), and the source code of all additional methods in any of the two files that are called (3) in the test setup context (purple) and (4) in the focal method (blue). Note, that the context resolution works recursively, so that a method which is called by a context method will also be included in the context.

Parsing: To encode the syntactic structure of the code, we parse the code snippets into abstract syntax trees (AST)[5]. Every code fragment is represented by a small tree, which we subsequently combine into a larger tree structure by adding a root node. This new tree is referred to as the *source tree* in the following. Note that context methods (3 and 4) – of which multiple ones may exist – are subsumed in separate subtrees beforehand.

Vocabulary: For the ASTs' leaf nodes – which represent identifiers occurring in code, like variable names – we follow common NLP practice (Babii et al., 2019) and tokenize them into fine-grain tokens using Byte Pair Encoding (BPE)[6]. For example an identifier like `getCount` may be splitted into `_get` and `Count`. We replace each leaf node with multiple leaf nodes with the same parent for each token in resulting list.

Linearization: Finally, to use the source tree as an input to the transformer, it is linearized in a form from which it can be decoded back into java source code: We encode each non-terminal node by an opening token `<[NodeType]>` and a closing token `<[/NodeType]>`, and render its child nodes recursively in between. Non-terminal nodes without children are represented by a single self-closing token `<[NodeType]><[/]>`, while value nodes simply result in a token representing their value. The target assertion's AST is encoded using the exact same preprocessing steps. In JavaParser each AST node is represented as an object, for which the corresponding source code can be retrieved by calling its `toString()` method. This enables reconstructing the source code of a linearized AST sequence.

4.2 Model

We train a vanilla transformer encoder-decoder model to perform test completion. For brevity we omit details about the transformer architecture here and refer the reader to Vaswani et al. (2017).

The linearized source trees input tokens form a sequence $\mathbf{x} = (x_1, x_2, x_3, \ldots, x_n)$, which is first processed by a transformer encoder, resulting in a sequence of continuous representations $\mathbf{z}(\mathbf{x})$, or shorter $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_n)$. From this, the autoregressive transformer decoder generates an out-

[5]We parse ASTs using Javaparser (van Bruggen et al., 2020).

[6]More specifically we train a sentencepiece unigram model (Kudo, 2018)

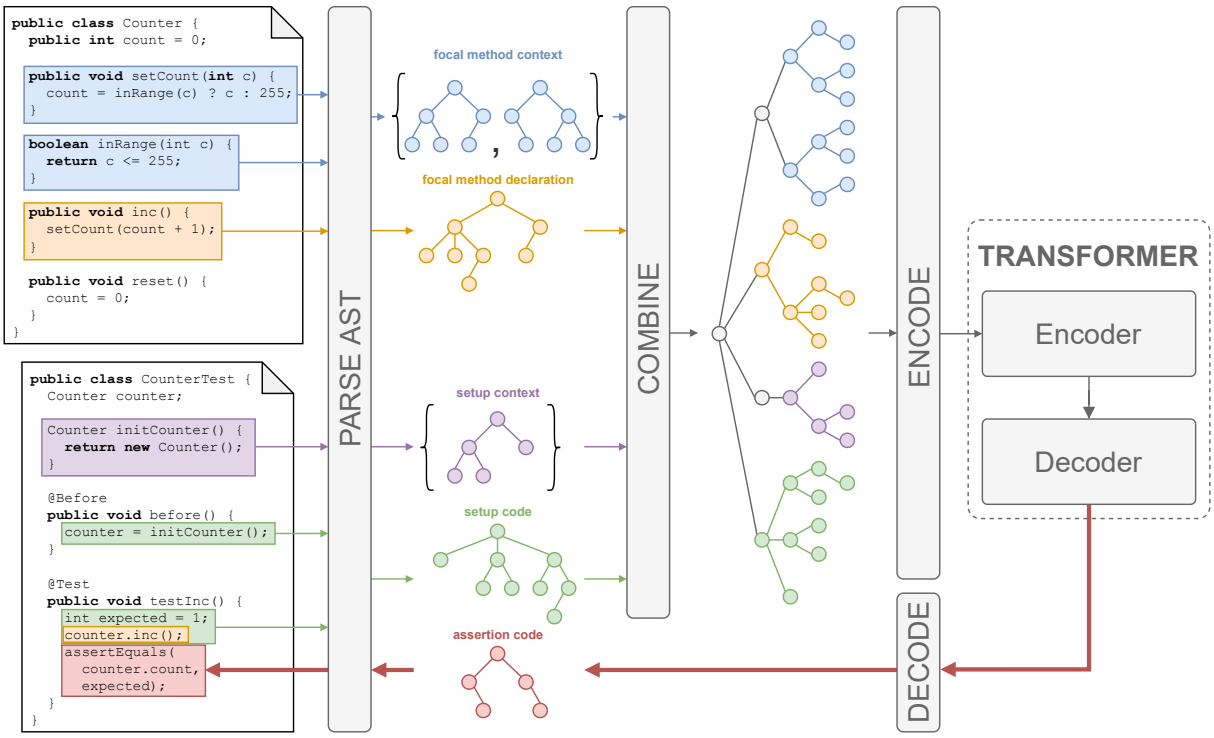

Figure 2: Visualization of our approach. We parse all four input parts into abstract syntax trees, which are then combined into one large *source tree*, that is then linearized and fed into the transformer. With this as input the transformer generates the linearized AST sequence of the target assertions (red). The generated AST is then decoded back into source code.

put token sequence $\mathbf{y} = (y_1, \ldots, y_m)$, i.e. the remainder of the test $\mathbf{x}$ (red in Figure 2). When generating token y_{i+1}, the decoder attends to the whole encoded sequence $\mathbf{z}$ as well as all previously generated symbols $y_1, \ldots, y_i$.

The sequence-to-sequence model is trained using teacher forcing (Williams and Zipser, 1989) by maximizing the conditional probability of the output sequence given the input, i.e. $p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{m} p(y_i|y_{\leq i-1}, \mathbf{z})$ with the cross entropy loss.

5 Experiments

We evaluate our transformer-based models on CONTEST to investigate the usefulness of contextual information and compare our syntax-based AST encoding with a token-only baseline. Finally, we investigate the effect of generalizing between training and test projects using CONTEST's project-level split.

5.1 Hyperparameters and Setup

We use a standard transformer architecture consisting of 6 transformer layers in both encoder and decoder, with 8 attention heads, 2048-dimensional feed-forward layers, $d = 512$ dimensional token embeddings, and a dropout rate of 0.2. To reduce the amount of parameters, we reuse the token embeddings of the transformer encoder as input and output embedding matrices in the decoder. We train our model using the Adam optimizer (Kingma and Ba, 2014) and an inverse square root learning rate scheduler with a linear warm-up.

Relevant training hyperparameters are optimized using the Optuna framework (Akiba et al., 2019) with the Tree-structured Parzen Estimator (Bergstra et al., 2011) on the validation loss. We report results for the model with the best validation loss. For the *learning rate* we investigate the set $\{0.01, 0.005, 0.001, 0.0001, 0.00001\}$, for *warmup steps* the set $\{500, 1000, 2000, 4000\}$, and for the *batch size* the set of $\{64, 128, 256, 512\}$. For our final evaluations, we generate sequences with greedy sampling as we found this to outperform nucleus sampling (Holtzman et al., 2019) on the validation set.

We train a Byte Pair Encoding with $16k$ subwords on the training set of the dataset and use the same BPE-model throughout all experiments. We implemented our model in PyTorch using existing transformer modules and all experiments are exe-

Model	BLEU	$BLEU_1$	$BLEU_2$	$BLEU_3$	$BLEU_4$	ROUGE-1 F-Measure	ROUGE-2 F-Measure	ROUGE-L F-Measure	Unparsable Rate (%)	Max length exceeded (%)
Linearized Tree	38.19	56.31	42.35	32.85	27.15	73.00	56.43	70.64	1.92	9.06
Linearized Tree (adjusted)	37.34	55.04	41.40	32.12	26.54	70.58	55.16	69.06	1.92	9.06

Table 2: Results of our core model utilizing syntax in terms of linearized abstract syntax trees. Absolute metrics (top) are compared with their adjusted version taking non-parsable outputs into account (bottom, see Equation (2)).

	Model	BLEU	$BLEU_1$	$BLEU_2$	$BLEU_3$	$BLEU_4$	ROUGE-1 F-Measure	ROUGE-2 F-Measure	ROUGE-L F-Measure	Unparsable Rate (%)	Max length exceeded (%)
(1)	Linearized Tree (LT)	**37.05**	**55.55**	**41.22**	**31.70**	**25.98**	**70.24**	**54.39**	**68.64**	1.91	9.41
	Tokenized	32.97	49.63	36.67	28.16	23.06	68.97	53.58	67.49	**1.07**	**3.79**
(2)	Context (LT)	**37.57**	**55.73**	**41.73**	**32.25**	**26.57**	**70.61**	**55.16**	**69.05**	1.91	**9.41**
	No Context (LT)	27.49	46.86	31.80	22.31	17.18	65.14	46.94	63.51	**1.73**	16.71
(3)	Random Split (LT)	**36.46**	**54.88**	**40.58**	**31.14**	**25.47**	**69.31**	**53.46**	**67.72**	1.91	**9.41**
	Project-based Split (LT)	15.42	37.52	20.35	10.89	6.80	55.88	32.97	53.90	**1.11**	30.43

Table 3: We compare – in pairs of two – a transformer operating on a linearized tree (LT) with a variation of itself: (1) one that operates on the actual tokens and not on the tree, (2) trained without contextual information, and (3) trained and evaluated on a project-based split. Note, that the scores for (3) are not directly comparable, as the dataset differs. All scores are *adjusted*.

cuted on a server using an Intel i9-10900X CPU, 128 GB of RAM and four NVIDIA GeForce 2080 Ti GPUs with 11 GB video memory each.

5.2 Results

We present the results of our experiments in Table 2 and Table 3, where we report BLEU and ROUGE scores. For BLEU we report the cumulative BLEU4 (Papineni et al., 2002) score as the main metric, as well as the single n-gram scores ($BLUE_{1-4}$), while for ROUGE we report F-measures for ROUGE-1, ROUGE-2, and ROUGE-L scores (Lin, 2004).

In Table 2 we report the scores of our core model as described in Section 4. We evaluate once with regular scores and once with the adjusted score (compare Section 3.6) and find that the model achieves a decent BLEU. In an ablation study in Table 3 we analyze the parts of our dataset in which we compare – in pairs of two – the best performing model (transformer operating on linearized tree, denoted by "LT") with a variation of itself: (1) a "no-AST" baseline that only takes the actual tokens as input, (2) trained without contextual information, and (3) trained and evaluated on a project-based split. Note, that the scores for (3) are not directly comparable as the dataset differs. The models may fail on different evaluation samples, therefore we only report the *adjusted* metrics here (Equation (2)) and compare only datapoints for which both models were able to generate a parsable prediction. Note that this causes the scores of the best performing model to vary between experiments.

5.2.1 Syntax

We compare the transformer utilizing syntactical information by operating on a linearized tree against a regular transformer trained on tokenized and BPE'd source code. For the tokenized model we tokenize the source code using `javalang` and then apply the same BPE encoding as for the other model. In the tree, each part of the input is represented by a subtree with a unique node label. For the tokenized version we concatenate the sections of the input sequence representing test code, tested code and the respective contexts (compare Section 4.1), whereas each section is prepended by a special marker token. We found that syntax is highly beneficial when generating test assertions, as Table 3 (1) shows that the model utilizing syntax yields an improvement of 4.1 in BLEU compared to the tokenized approach. However, the tokenized approach is able to generate parsable code more often. Due to the shorter sequence length it is also able to generate longer output sequences. Recall that input and output of the AST model is longer because of the tree linearization format (compare Section 4.1).

5.2.2 Context

To evaluate the effect the context information has on the model's performance, another ablation experiment is conducted. We train a variation of the model with inputs in which context parts of the input have been removed. We drop the test setup context (Figure 1, purple) and focal method context (blue). Note that this setup is similar to other test completion datasets (i.e. ATLAS (Tu-

fano et al., 2020a)) as those offer only parts of the test before the assertions (1) and the focal method (2) as input to the model. It is also worth noting that the model trained with context benefits from the additional information and is able to apply this knowledge to improve the quality of its predictions by 10.1 BLEU. This feels natural, as even for a developer it is often hard to understand the functionality of a method without investigating methods called within. Consider the example in Figure 1, in which there is no way of understanding the functionality of `setCount(int c)` that increments until 255 without having access to the method `inRange(int c)` called inside `setCount` which implements this check.

5.2.3 Project-level Splits

The code style within a software project mostly follows certain paradigms, and tests inside the same repository may employ the same coding style. This could be exploited by the model, which then does not need to learn the actual semantics of the code. In a last – and maybe most important – experiment we investigate how well the model is able to generalize to unseen repositories. We argue that this is the most realistic use case, as most code models will be applied to unseen repositories.

We therefore create a different version of our dataset, in which it is split by software projects and train the same model again on the resulting dataset (see Section 3.5). From Table 3 (3) we can see that the model trained on the project-based split fails to generalize across projects. This results in a BLEU score of 15.4, around half of what the model on the random split achieves. We consider this an open challenge and are looking forward to future findings indicating whether pre-trained language models on code can improve generalization. Following previous research we would like to emphasize that project-level splits should be the golden standard if one creates a dataset for machine learning on source code (Alon et al., 2019). However, as most code datasets are build upon GitHub data one should consider test leakage when evaluating large-scale pre-trained language models for code.

6 Conclusion

We have proposed a large-scale benchmark for automatic test completion coined CONTEST. In addition to pairs of test and focal methods, our benchmark uniquely contains context and setup code, offers multiline targets, and defines project-level splits. We have shown in an ablation study that context information appears to be extremely relevant to the task of test completion, and that a sequence-to-sequence transformer baseline struggles with generalizing across projects. Future research could aim to improve cross-project generalization, for example by fine-tune large scale pre-trained language models for code (Feng et al., 2020; Roziere et al., 2021) on CONTEST.

Acknowledgements

This work was funded by German Federal Ministry of Education and Research (Program FHprofUnt, Project DeepCA (13FH011PX6)).

References

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

M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40.

Hlib Babii, Andrea Janes, and Romain Robbes. 2019. Modeling vocabulary for big code machine learning. *arXiv preprint arXiv:1904.01873*.

David Belhumeur, Dale Brenneman, and Ken Pereira. 2004. Agitator for interactive exploratory testing. Accessed: 2021-04-26.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation.

Danny van Bruggen, Federico Tomassetti, and Roger Howell. 2020. Release javaparser - 3.16.1.

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

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

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

Felipe Hoffa. 2016. Github on bigquery: Analyze all the open source code. Accessed: 2021-04-26.

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

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

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*.

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

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

Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28.

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

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

Yolande Poirier. 2018. What are the most popular libraries java developers use? based on github's top projects. Accessed: 2021-04-26.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages.

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

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

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2020b. Generating accurate assert statements for unit test cases using pretrained transformers. *CoRR*, abs/2009.05634.

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

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1398–1409, New York, NY, USA. Association for Computing Machinery.

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

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