How robust is neural code completion to cosmetic variance?

Anonymous Author(s)

ABSTRACT

Neural language models hold great promise as tools for computer-aided programming, but questions remain over their reliability and the consequences of overreliance. In the domain of natural language, prior work has revealed these models can be sensitive to naturally-occurring variance and malfunction in unpredictable ways. A closer examination of neural language models is needed to understand their behavior in programming-related tasks. In this work, we develop a methodology for systematically evaluating neural code completion models using common source code transformations such as synonymous renaming, intermediate logging, and independent statement reordering. Applying these synthetic transformations to a dataset of handwritten code snippets, we evaluate three SoTA models, CodeBERT, GraphCodeBERT and RobertA-Java, which exhibit varying degrees of robustness to cosmetic variance. Our approach is implemented and released as a modular and extensible toolkit for evaluating code-based neural language models.

ACM Reference Format:
Anonymous Author(s). 2022. How robust is neural code completion to cosmetic variance?. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Neural language models play an increasingly synergetic role in software engineering, and are featured prominently in recent work on neural code completion [2]. Yet from a development perspective, the behavior of these models is opaque: partially completed source code written inside an editor is sent to a remote server, which returns a completion. This client-server architecture can be seen as a black-box or *extensional* function. Could there be a way to evaluate the behavior of neural language models in this setting?

First conceived in the software testing literature, metamorphic testing [3] is a concept known in machine learning as *self-supervision*. In settings where labels are scarce but invariant to certain groups of transformation or *metamorphic relations*, given a finite labeled dataset, one can generate an effectively limitless quantity of synthetic data by selecting and recombining those transformations. For example, computer vision models should be invariant to shift, scale and rotation: given a small dataset of labeled images, we can apply these transformations to generate much larger training or validation set. Could similar kinds of transformations exist for code?

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

Recent work in neural language modeling has shown impressive progress in long-range sequence prediction, starting with self-attention [12], to the BERT [6] and RoBERTa [10] architectures, now widely available in neural code completion models such as CodeBERT [7] and GraphCodeBERT [8]. However these models have known limitations, such as their sensitivity to adversarially-constructed noise in the input space [11]. Recognizing the risk this issue poses for code completion, recent work has explored adversarial robustness for source code [1, 14]. Due to the clear distinction between syntax and semantics in programming languages, one can more precisely reason about semantically admissible perturbations, a task which is considerably more difficult in natural languages due to the problem of semantic drift. Similar research has been undertaken [2, 4, 13] to characterize the families of computational languages neural language models can recognize in practice.

Our work builds on this literature from an engineering standpoint: we explore the extent to which neural code completion models generalize to plausible cosmetic variation. Applying synthetic transformations to naturally-occurring code snippets, we compare the robustness of these models on two downstream tasks: code and document completion. Whereas prior work has explored similar probing tasks on both natural language and source code, they are typically adversial and may admit a much wider class of transformations. Our work is treats the model as a black box and considers a very narrow class of transformations. We have identified three high-level categories of source code transformations:

(1) **Syntactic**, which may be valid (i.e., parsable) or invalid (e.g., typos, imbalanced parentheses, unparsable code)
(2) **Semantic**, either preserving (functional code clones) or altering (e.g., disequal constant or expression rewriting)
(3) **Cosmetic**, e.g., variable renaming, independent statement reordering, extra documentation, dead code, or logging

In contrast with syntactic or semantic transformations, cosmetic transformations are semantically identical, syntactically valid and only superficially alter syntactic structure. We show that even in this highly restrictive space of transformations, source code has many degrees of freedom: two authors implementing the same function may select different variable names or other cosmetic features, such as whitespaces, diagnostic statements or comments. One would expect neural code completion for programming languages semantically invariant under those changes to share the same invariance: *cosmetically-altered code snippets should not drastically change the language model's predictions.* Yet our results suggest that even in this narrow space of transformations, SoTA neural code completion models exhibit sensitivity to cosmetic changes.

In addition to its empirical value, our work provides a modular and extensible framework of software tools and unit tests for evaluating black-box code completion models and constructing similar benchmarks. Our toolkit is open source and may be obtained at the following URL: https://anonymous.4open.science/r/cstk-1458

2 METHOD

Our goal is to measure the robustness of SoTA neural code completion models on natural code snippets exposed to various cosmetic transformations. To do so, we first construct one SCT from each of the following five categories of cosmetic changes:

(1) **Synonym renaming**: renames variables with synonyms
(2) **Peripheral code**: introduces dead code to source code
(3) **Statement reordering**: swaps independent statements
(4) **Permute argument order**: scrambles method arguments

Ideally, these SCTs would be implemented using a higher-order abstract syntax (HOAS) to ensure syntactic validity, however for the sake of simplicity, we implemented the transformations using a set of ad-hoc regular expressions (regexes). While somewhat clumsy for more complex SCTs, we observed that regex-based pattern matching can reliably perform cosmetic treatments like renaming and linear statement reordering without much difficulty. Specifically, we have implemented our SCTs as follows:

(1) The `renameTokens` SCT substitutes each CamelCase subword in the most frequent user-defined token with a uniformly-sampled lemma from its WordNet hypernym ego graph up to three hops away, representing an alternately-chosen (e.g., variable or function) name of similar meaning.
(2) The `addExtraLogging` SCT adds intermittent print statements in linear chains of code, with a single argument synthesized by the code completion model for added variation. More generally, this can be any superfluous statement which does not change the runtime semantics.
(3) The `swapMultilineNo` SCT swaps adjacent lines of equal scope and indentation which share no tokens in common. Although this SCT may occasionally introduce semantic drift in imperative or effectful code, it ideally represents an alternate topological sort on the dataflow graph.
(4) The `permuteArgument` SCT performs a Fisher-Yates shuffle on the arguments of a user-defined function of dyadic or higher-arity, representing an alternate parameter order of some function outside the JDK standard library.

Idempotent SCTs (i.e., snippets which remain unchanged after the SCT is applied) are considered invalid and discarded. Although (3) and (4) may produce semantic variants, strictly speaking, we cannot rule out the possibility that any of the aforementioned SCTs are semantically-preserving due to the inherent complexity of source code analysis, however we have manually validated their admissibility for a large fraction of cases. A more principled macro system would help to alleviate these concerns, however a framework for rewriting partial Java code snippets with flexible error recovery is, to the best of our knowledge, presently unavailable.

Our framework is capable of evaluating both code completion and document synthesis using the same end-to-end strategy. In principle, this measurement can be done inside the model's latent space using a sensitivity margin or via some metric on the raw data. In practice, we decided to focus on the input domain and consider two metrics: masked token completion accuracy for code completion and ROUGE-synonym score for document synthesis.

For code completion, we uniformly sample and mask N individual tokens from both the original and transformed code snippet for evaluation. We then collect the model's highest-scoring predictions for each mask location, and average the completion accuracy on the original and transformed code snippet, recording the relative difference in accuracy before and after transformation.

1.a) Original method	1.a) Synonymous Variant

`public void flush(int b) {`
`  buffer.write((byte) b);`
`  buffer.compact();`
`}`
	`public void flush(int b) {`
	`  cushion.write((byte) b);`
	`  cushion.compact();`
	`}`

2.a) Multi-masked method	**2.b) Multi-masked variant**

`public void <MASK>(int b) {`
`  buffer.<MASK>((byte) b);`
`  <MASK>.compact();`
`}`
	`public void <MASK>(int b) {`
	`  cushion.<MASK>((byte) b);`
	`  <MASK>.compact();`
	`}`

3.a) Model predictions	**3.b) Model predictions**

`public void output(int b) {`
`  buffer.write((byte) b);`
`  buffer.compact();`
`}`
	`public void append(int b) {`
	`  cushion.add((byte) b);`
	`  cushion.compact();`
	`}`

The model correctly predicted $\frac{2}{3}$ masked tokens in the original snippet, and $\frac{1}{3}$ after renaming, so the relative accuracy is $\frac{\frac{2}{3}-\frac{1}{3}}{\frac{2}{3}} = \frac{1}{2}$.

Similarly, in the case of document synthesis, we mask a naturally-occurring comment and autoregressively synthesize a new one in its place, then compare the ROUGE-scores of the synthetic documents before and after transformation. In the following example, we apply the `renameTokens` SCT, then mask the comment on line 3 and autoregressively sample tokens from the decoder to generate a two synthetic comments, before and after applying the SCT.

```
//-------------------------------------------------------------
// 1.) Original method with ground truth document
//-------------------------------------------------------------
  public void testBuildSucceeds(String gradleVersion) {
      setup( gradleVersion );
      // Make sure the test build setup actually compiles
      BuildResult buildResult = getRunner().build();
      assertCompileOutcome( buildResult, SUCCESS );
  }

//-------------------------------------------------------------
// 2.) Synthetic document before applying SCT
//-------------------------------------------------------------
  public void testBuildSucceeds(String gradleVersion) {
      setup( gradleVersion );
      // build the tests with gradletuce compiler
      BuildResult buildResult = getRunner().build();
      assertCompileOutcome( buildResult, SUCCESS );
  }

//-------------------------------------------------------------
// 3.) Synthetic document after applying renameTokens SCT
//-------------------------------------------------------------
  public void testBuildSucceeds(String gradleAdaptation) {
      setup( gradleAdaptation );
      // build the actual code for test suite generation
      BuildResult buildResult = getRunner().build();
      assertCompileOutcome( buildResult, SUCCESS );
  }
```

Initially, we seeded the document completion using `//<MASK>` and applied a greedy autoregressive decoding strategy, recursively sampling the softmax top-1 token and subsequently discarding all malformed comments. This strategy turns out to have a very high rejection rate, due to its tendency to produce whitespace or unnatural language tokens (e.g., greedy decoding can lead to sequences like `// ///// //` or temporarily disabled code, e.g., `// System.out.println("debug")`). A simple fix is to select the highest-scoring prediction with natural language characters. By conditioning on at least one alphabetic character per token, one obtains more coherent documentation and rejects fewer samples. It is possible to construct a more sophisticated natural language filter, however we did not explore this idea in great depth.

Our experimental architecture is to our knowledge, unique, and merits some discussion. The entire pipeline from data mining to preprocessing, evaluation and table generation is implemented as a pure functional program in the point-free style. The evaluation pipeline can be described succintly as follows. Given a code completion model `cc: Str→Str`, a list of code snippets, `snps: List<Str>`, a masking procedure, `msk: Str→Str`, an SCT, `sct: Str→Str`, and a single metric over code snippets, `mtr: (Str, Str)→Float`, we measure the average relative discrepancy before and after applying `sct` to `snps`:

```
fun evaluate(cc, snps, msk, sct, mtr) = Δ(
  zip(snps, snps | msk | cc) | mtr | average,
  zip(snps | sct, snps | sct | msk | cc) | mtr | average
)
```

where `|` maps a function over a sequence, and `zip` zips two sequences into a sequence of pairs. We assume `snps` and `msk` are fixed, and evaluate three neural code completion models across five different SCTs, various complexity code buckets, and two separate metrics, representing downstream tasks (i.e., code and document synthesis). Those results are reported in § 3.

Using our framework, it is possible view the marginals of a rank-n tensor, representing an n-dimensional hyperdistribution formed by the Cartesian product of all variables under investigation (e.g. code complexity, metric, task, model). During evaluation, we sample these independent variables uniformly using a quasirandom sequence to ensure entries are evenly populated. We then record the first and second moments of the dependent variable of interest (e.g. average relative ROUGE-score discrepancy) using a sketch-based historgram. Results are continuously delivered to the user, who may preview 2D marginals of any pair and watch the error bounds grow tighter as additional samples are drawn. This feature is useful when running on preemptible infrastructure and can easily be parallelized to increase the experiment's statistical power or explore larger subspaces of the experimental design space.

3 EXPERIMENTS

Our dataset consists of a hundred of the highest-starred Java repositories hosted by GitHub organizations with over 100 forks and between 1 and 10 MB in size. This ensures a diverse dataset of active repositories with reasonable quality and stylistic diversity.

We evaluate three state-of-the-art pretrained models (GraphCodeBERT, CodeBERT and RoBERTa) on two downstream tasks (code completion and document synthesis). While the number of samples may vary per model and per bucket, we provide the same wall clock time (180 minutes) and hardware resources (NVIDIA Tesla V100) to each model. The number of code snippets each can evaluate in the allotted time vary depending on the architecture, but in each case, the significant figures have mostly converged.

Below, we report the average relative decrease in completion accuracy after applying the SCT in the column to a code snippet whose Dyck-1 complexity is reported in the row heading, plus or minus the variance, with a sample size reported in parentheses.

CodeBERT

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
10-20	-0.13 ± 0.094 (42)	0.040 ± 0.329 (156)	0.208 ± 0.348 (359)	0.033 ± 0.082 (15)
20-30	-0.26 ± 0.189 (112)	0.137 ± 0.299 (312)	0.116 ± 0.338 (542)	-0.01 ± 0.202 (82)
30-40	-0.29 ± 0.224 (62)	0.098 ± 0.264 (163)	0.185 ± 0.335 (329)	0.081 ± 0.109 (73)
40-50	-0.27 ± 0.222 (74)	0.142 ± 0.373 (138)	0.092 ± 0.357 (232)	0.043 ± 0.208 (82)
50-60	-0.09 ± 0.295 (66)	0.041 ± 0.282 (130)	0.120 ± 0.267 (335)	0.014 ± 0.181 (136)
60-70	-0.21 ± 0.244 (60)	0.020 ± 0.280 (108)	0.161 ± 0.252 (179)	-0.02 ± 0.211 (98)
70-80	-0.11 ± 0.384 (24)	0.071 ± 0.343 (55)	0.081 ± 0.376 (79)	-0.03 ± 0.356 (73)
80-90	-0.12 ± 0.325 (42)	0.080 ± 0.363 (70)	0.035 ± 0.429 (97)	-0.04 ± 0.350 (75)
90-100	-0.04 ± 0.307 (37)	0.214 ± 0.291 (52)	0.218 ± 0.293 (70)	0.075 ± 0.226 (69)

GraphCodeBERT

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
10-20	-0.31 ± 0.204 (21)	0.201 ± 0.384 (114)	0.137 ± 0.374 (276)	0.166 ± 0.055 (6)
20-30	-0.18 ± 0.155 (93)	0.130 ± 0.317 (247)	0.034 ± 0.323 (438)	-0.03 ± 0.209 (71)
30-40	-0.19 ± 0.233 (48)	0.174 ± 0.209 (149)	0.198 ± 0.340 (288)	-0.04 ± 0.108 (64)
40-50	-0.22 ± 0.256 (61)	0.093 ± 0.346 (94)	0.078 ± 0.346 (181)	-0.08 ± 0.282 (63)
50-60	-0.26 ± 0.228 (59)	0.064 ± 0.259 (139)	0.099 ± 0.280 (323)	-0.06 ± 0.185 (135)
60-70	-0.21 ± 0.207 (45)	0.058 ± 0.251 (85)	0.122 ± 0.249 (170)	-0.00 ± 0.224 (82)
70-80	-0.39 ± 0.319 (17)	0.126 ± 0.417 (46)	0.072 ± 0.335 (69)	-0.11 ± 0.315 (63)
80-90	-0.00 ± 0.339 (37)	-0.00 ± 0.294 (64)	0.056 ± 0.340 (85)	-0.01 ± 0.295 (69)
90-100	-0.13 ± 0.291 (29)	0.209 ± 0.386 (49)	0.035 ± 0.342 (67)	0.011 ± 0.254 (61)

RoBERTa

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
10-20	-0.42 ± 0.175 (122)	0.296 ± 0.406 (277)	0.387 ± 0.349 (704)	0.0 ± 0.0 (12)
20-30	-0.33 ± 0.172 (168)	0.258 ± 0.338 (460)	0.302 ± 0.288 (838)	-0.04 ± 0.145 (101)
30-40	-0.23 ± 0.188 (107)	0.084 ± 0.261 (313)	0.224 ± 0.311 (604)	0.031 ± 0.172 (142)
40-50	-0.05 ± 0.237 (118)	0.183 ± 0.291 (249)	0.254 ± 0.268 (412)	-3.07 ± 0.098 (155)
50-60	-0.06 ± 0.239 (108)	0.085 ± 0.253 (259)	0.246 ± 0.242 (510)	-0.00 ± 0.138 (203)
60-70	-0.03 ± 0.196 (80)	-4.31 ± 0.282 (171)	0.174 ± 0.273 (291)	-0.02 ± 0.240 (144)
70-80	0.124 ± 0.409 (35)	0.062 ± 0.253 (97)	0.174 ± 0.338 (132)	-0.01 ± 0.235 (107)
80-90	-0.06 ± 0.394 (43)	0.053 ± 0.350 (94)	0.225 ± 0.359 (132)	-0.00 ± 0.296 (103)
90-100	0.118 ± 0.341 (47)	0.064 ± 0.347 (77)	0.294 ± 0.339 (95)	0.124 ± 0.309 (88)

Sample size varies across SCTs because not all SCTs modify all snippets, and we discard all code snippets which remain unchanged after transformation, which results in column imbalance.

Our results support the relative model rankings as reported by prior literature: RoBERTa ≪ CodeBERT < GraphCodeBERT. As we can see, RoBERTa is considerably more sensitive to cosmetic variance than CodeBERT and GraphCodeBERT, which are relatively close contenders. In all cases, the `swapMultilineNo` and `permuteArgument` SCTs exhibit a more detrimental effect than the other SCTs. Unexpectedly, it appears that token renaming tends to improve completion accuracy across all models on average.

Below are the ROUGE scores we collected for document synthesis, using the ROUGE-synonym metric. Like above, we report the average relative difference in ROUGE-synonym scores alongside their variance and sample size for each SCT, complexity bucket and model. Although the sample sizes are smaller and less conclusive, we observe a similar trend across SCTs emerging.

GraphCodeBERT

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
60-70	NaN ± NaN (0)	-0.25 ± 0.281 (3)	3.922 ± 129.3 (8)	-0.29 ± 0.036 (2)
70-80	NaN ± NaN (0)	-0.66 ± 0.0 (1)	0.947 ± 3.621 (6)	3.689 ± 17.37 (3)
80-90	5.833 ± 0.0 (1)	-0.66 ± 0.0 (1)	-0.02 ± 0.720 (7)	-0.45 ± 0.595 (3)
90-100	5.417 ± 0.0 (1)	3.179 ± 22.20 (7)	3.746 ± 13.65 (12)	2.551 ± 0.0 (1)
100-110	NaN ± NaN (0)	0.040 ± 0.909 (7)	1.156 ± 8.423 (13)	-0.54 ± 0.539 (7)

CodeBERT

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
110-120	0.071 ± 0.0 (1)	-0.56 ± 0.068 (2)	2.345 ± 28.16 (6)	-0.95 ± 0.0 (1)
120-130	-0.81 ± 0.0 (1)	0.307 ± 0.036 (2)	1.549 ± 12.76 (5)	NaN ± NaN (0)
130-140	-0.33 ± 0.005 (2)	-0.02 ± 0.137 (6)	2.131 ± 8.023 (9)	20.66 ± 0.0 (1)
140-150	-1.0 ± 0.0 (1)	-0.24 ± 0.304 (3)	0.239 ± 1.016 (6)	-0.51 ± 0.0 (1)

RoBERTa

Complexity	renameTokens	swapMultilineNo	permuteArgument	addExtraLogging
30-40	NaN ± NaN (0)	NaN ± NaN (0)	3.142 ± 14.87 (2)	1.4 ± 0.0 (1)
40-50	-0.69 ± 0.037 (2)	0.644 ± 19.80 (13)	0.350 ± 12.70 (39)	-0.53 ± 0.864 (9)
50-60	NaN ± NaN (0)	1.653 ± 9.808 (14)	1.671 ± 26.44 (85)	3.194 ± 75.10 (8)
60-70	NaN ± NaN (0)	0.186 ± 1.067 (15)	2.107 ± 19.51 (82)	0.218 ± 4.887 (8)
70-80	NaN ± NaN (0)	9.666 ± 183.9 (8)	2.515 ± 56.38 (48)	0.200 ± 3.070 (10)
80-90	0.210 ± 1.319 (4)	3.016 ± 33.88 (15)	1.774 ± 46.96 (55)	4.167 ± 49.65 (14)
90-100	-0.35 ± 0.075 (3)	0.520 ± 1.556 (21)	2.435 ± 39.08 (54)	0.407 ± 2.483 (18)
100-110	2.645 ± 13.31 (4)	0.613 ± 5.410 (19)	2.900 ± 96.60 (57)	1.348 ± 14.02 (27)

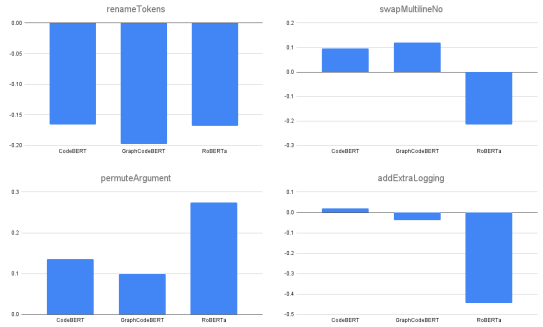

Figure 1: Average relative difference across all models.

In both cases, we observe a clear trend across method complexity: SCTs in low complexity code have a larger effect on completion quality than similar transformations in high-complexity code. We hypothesize this phenomenon can be explained by the fact that the same transformation can have a comparatively larger effect on a shorter code snippet than a longer one, which contains more contextual information and is thus more stable to minor perturbations.

Examining the source code transformations, we notice that renaming can have a significant effect on document synthesis. As the model frequently copies tokens from the source code to the document and vis-versa, renaming tends to have a deleterious effect on document quality. Likewise, swapping multiline statements appears to have a significant negative effect on document quality.

4 CONCLUSION

The work described herein is primarily an empirical study, but also showcases a framework and a systematic approach to evaluating neural code completion models. It offers a number of advantages from a software engineering standpoint: due to its functional implementation, it is efficient, parallelizable and highly modular, allowing others to easily reuse and extend our work with new benchmarks.

Despite its simplicity, the regex-based SCT approach has some shortcomings. Although regular expressions are easy to implement and do support rudimentary transformations, they are a crude way to manipulate source code. In order to generate semantically valid transformations, one must really use full-fledged term rewriting system, such as higher-order abstract syntax or some kind of parser-combinator. Several options were evaluated, including OpenRewrite, TXL [5], Refal [9] et al., but their features were ultimately found wanting (e.g., poor error recovery) and the complexity of using them (e.g., parsing, API integration) proved too burdensome.

Our SCTs can be viewed as "possible worlds" in the tradition of modal logic: the original author plausibly could have chosen an alternate form of expressing the same procedure. Although we are unable to access all these worlds, we can posit the existence and likelihood of some, and given a dataset of alternate code snippets, begin to probe the completion model's predictions.

One intriguing avenue for future work would be to consider combinations of source code transformations. This would vastly expand the cardinality of the validation set, enabling us to access a much larger space of possible worlds, albeit potentially at the risk of lower semantic admissibility, as arbitrary combinations of SCTs can quickly produce invalid code. This presents an interesting engineering challenge and possible extension to this work.

Although we currently only use average mutlimask accuracy and ROUGE-synonym metrics, it would be useful to incorporate various other metrics such as mean average precision (MAP), mean reciprocal rank (MRR), and normalized discounted cumulative gain (NDCG). In addition to their utility as yardstick for evaluating model robustness, these metrics can be used to retrain those same models, a direction we hope to explore in future work.

Finally, one could imagine using the code completion model itself to generate code for testing the same model. We have implemented this functionality to a limited extent in the `addExtraLogging` SCT, in which the model synthesizes a single token to log, and the `insertComment` SCT, where the model inserts a short comment. While this approach could be a useful way to generate additional training data, it would would require careful monitoring and post-processing to avoid introducing unintended feedback loops.

Neural language models hold much promise for improved code completion, however complacency can lead to increased reviewer burden or more serious technical debt if widely adopted. While trade secrecy may prevent third-party inspection of pretrained models, users would still like some assurance of the model's robustness to naturally-occurring variance. Our work helps to address this use case by treats the model as a black box: it does not require access to the model parameters or training data.

Our contributions in this work are twofold: we show that SoTA neural language models for source code, depsite their effectiveness on long-range seqeunce prediction tasks are unpredictable in the presence of cosmetic noise. We also describe a systematic approach and open source implementation of a new software toolkit for evaluating code completion models.

REFERENCES

[1] Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

[2] Mark Chen et al. Evaluating large language models trained on code, 2021.

[3] TY Chen, J Feng, and TH Tse. Metamorphic testing of programs on partial differential. *Information and Computation*, 121(1):93–102, 1995.

[4] Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code. *arXiv preprint arXiv:2010.07987*, 2020.

[5] James R Cordy. TXL-a language for programming language tools and applications. *Electronic notes in theoretical computer science*, 110:3–31, 2004.

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

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[8] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

[9] Ruten Gurin and Sergei Romanenko. The Refal Plus programming language. 1991.

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019.

[11] Lichao Sun, Kazuma Hashimoto, Wenpeng Yin, Akari Asai, Jia Li, Philip Yu, and Caiming Xiong. Adv-BERT: BERT is not robust on misspellings! Generating nature adversarial samples on BERT. *arXiv preprint arXiv:2003.04985*, 2020.

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

[13] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. *arXiv preprint arXiv:1805.04908*, 2018.

[14] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. Adversarial robustness of deep code comment generation. *arXiv preprint arXiv:2108.00213*, 2021.