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, featuring prominently in recent work on neural code completion [1]. Yet from a developer's 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 real-time suggestion. This client-server architecture can be seen as a black-box or *extensional* function from a mathematical perspective. How can we evaluate the behavior of neural language models in this setting? The role of automated testing becomes evident.

First conceived in the software testing literature, metamorphic testing [2] 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 infinite amount 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 invariants exist for source code?

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/1122445.1122456

Source code in semantically or even syntactically-equivalent programs has many degrees of freedom: two authors implementing the same function may select different variable names or cosmetic features, such as whitespaces, diagnostic statements or comments. One would expect neural code completion for a programming language semantically invariant under those changes to share the same invariance: *cosmetically-altered code snippets should not drastically change the model's predictions.* For example, code completion in Java should be invariant to variable renaming, whitespace placement, extra documentation or similar transformations.

One can make this statement a little more formal. Given a neural code completion model `ncc: Str→Str`, a list of code snippets, `snps: List<Str>`, a masking procedure, `msk: Str→Str`, a single cosmetic transformation, `ctx: Str→Str`, and a code completion metric, `mtr: (Str, Str)→Float`, we would expect the average completion accuracy to remain approximately the same regardless of whether the transformation was applied, i.e.,

```
fun ctxVariance(ncc, snps, msk, ctx, mtr) = abs(
  zip(snps, snps | msk | ncc) | mtr | average -
  zip(snps | ctx, snps | ctx | msk | ncc) | mtr | average
)
```

where `|` maps a function over a sequence, and `zip` zips two sequences into a sequence of pairs. We will use this notation throughout the paper. We assume `snps` and `msk` are fixed, and evaluate three different neural code completion models, five different transformations and two different metrics. Results are reported in

Do language models learn to recognize patterns in source code? If so, what kinds? A longstanding research area in software engineering tries to extract idioms from large software repositories. Likewise, a longstanding goal of neural program synthesis is compositional generalization. Both currently require designing a feature representation that "selects" for a specific kind of invariance — for example, structural similarity selects for name-invariance, and semantic similarity captures structural invariance.

In our work, we explore the extent to which recent language models trained on source code learn regularities they were not explicitly designed to represent. We analyze three different language models, and compare their zero-shot generalization on synthetically-generated patterns. We generate code snippet pairs corresponding to manually-designed variance, and measure the effect on various language models. Each model is evaluated on a set of code snippets exposed to synthetic transformations and predicts a masked token.

Our goal then, is to identify equivariant representations captured by the language model (LM). To do so, we generate program transformations (PTs) and measure vector equivariance in four rewriting categories:

(1) Syntactic - can the LM detect syntactically invalid PTs? (e.g. syntax corruption, imbalanced parenthesis)
(2) Structural - can the LM detect syntactically valid, but structurally altered PTs? (e.g. use before declaration, permuted argument order)

(3) Semantic - can the LM detect structurally valid, but semantically altered PTs? (e.g. constant modification, operator substitution and order of operations)
(4) Equivalence - can the LM detect semantically valid but rewritten PTs? (e.g. semantically valid rewrites)

To generate these SCTs, we implement a source code transformation tool which generates synthetic variants of naturally-occurring source code snippets from each of these three categories and evaluates the model for completion accuracy. We have implemented the following source code transformations (SCTs):

(1) **Synonym renaming**: rename variables with synonyms
(2) **Peripheral code**: introduce dead code to source code
(3) **Statement reordering**: swap independent statement order
(4) **Document generation**: add synthetic documentation
(5) **Loop bounds fuzzing**: change loop bounds conditions
(6) **Permute argument order**: scramble method arguments
(7) **Literal mutation**: mutate contents of primitive types

For each SCT, we mask various tokens in the code snippet and compare the model's ability to fill in the correct token before and after the transformation is applied. Our goal is to measure how sensitive the pretrained model is to each type of SCT.

We also generate some documentation, using the same model. These additions represent a natural language comment which a human might have written. Below are two examples of Javadocs generated by our document synthesizer using GraphCodeBERT:

```
/**
 * Write a single byte to the output stream.
 */
public void write(int b) throws IOException {
  if (!buffer.hasRemaining()) {
    throw new IOException("IPPacketOutputStream buffer is full");
  }
  buffer.put((byte) b);
  buffer.flip();
  sink();
  buffer.compact();
}

/**
 * receive packet from local socket
 */
public int receive(byte[] packet) throws IOException {
  int r = localSocket.getInputStream().read(packet);
  if (GnirehtetService.VERBOSE) {
    Log.v(TAG, "Receiving packet: " +
    Binary.buildPacketString(packet, r));
  }
  return r;
}
```

For each token in a string, CodeBERT emits a length-768 vector, so a line with n tokens produces a matrix of shape $\mathbb{R}^{768\times(n+1)}$, the first column of which contains the final hidden layer activations. We concatenate the CodeBERT sequence '`<s>`' and using the source tokens, encode and vectorize the encoded sequence using the vocabulary, then take the first row as our sequence embedding as depicted in Fig. 1. In § ??, we compare various distance-metrics to fetch the nearest sequence embeddings in our database and compare precision and recall across various types of distance metrics.

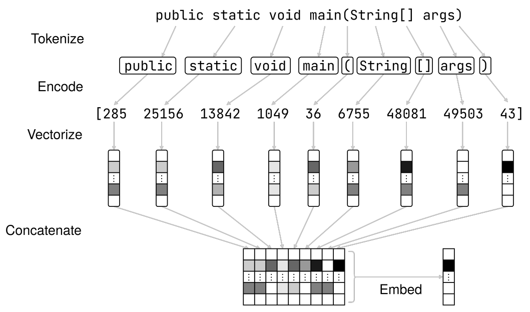

Figure 1: CodeBERT takes a unicode sequence and emits a vector sequence, which we accumulate into a single vector.

2 RESULTS

Model	IRA	QDL	P/R
CodeGPT [6]	X	X	X
GraphCodeBERT [4]	X	X	X
CodeBERT-small [3]	X	X	X
RoBERTa-Java [5]	X	X	X
Copilot[1]	X	X	X

Table 1: Experiments table for comparing pretrained LM embeddings on source code snippets. IRA: Iterrater agreement. QDL: query description length, P/R: Precision/recall

3 DISCUSSION

We examine in our work three primary questions. Given a set of (1) source code (2) a set of SCTs, (3) a set of slicing and tokenization methods, how do these factors affect accuracy on the masked code completion task. We sample a the top 100 most-starred Java repositories from GitHub organizations with over 100 forks and between 1 and 10 MB in size. From these, we extract a set of Java methods using the heuristic described in §A.

4 CONCLUSION

REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[7] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

A SLICING PROCEDURE

We describe below a simple heuristic for extracting method slices in well-formed source code using a Dyck counter. [1] A common coding convention is to prefix functions with a keyword, followed by a group of balanced brackets and one or more blank lines. While imperfect, we observe this pattern can be used to slice methods in a variety of langauges in practice. A Kotlin implementation is given below, which will output the following source code when run on itself:

```
fun String.sliceIntoMethods(kwds: Set<String> = setOf("fun ")) =
  lines().fold(-1 to List<String>(0)) { (dyckCtr, methods), ln ->
    if (dyckCtr < 0 && kwds.any { it in ln }) {
      ln.countBalancedBrackets() to (methods + ln)
    } else if (dyckCtr == 0) {
      if (ln.isBlank()) -1 to methods else 0 to methods.put(ln)
    } else if (dyckCtr > 0) {
      dyckCtr + ln.countBalancedBrackets() to methods.put(ln)
    } else -1 to methods
  }.second

fun List<String>.put(s: String) = dropLast(1) + (last() + "\n$s")

fun String.countBalancedBrackets() = fold(0) { sum, char ->
  val (lbs, rbs) = setOf('(', '{', '[') to setOf(')', '}', ']')
  if (char in lbs) sum + 1 else if (char in rbs) sum - 1 else sum
}

fun main(args: Array<String>) =
  println(args[0].sliceIntoMethods().joinToString("\n\n"))
```

[1]https://en.wikipedia.org/wiki/Dyck_language

B ANALYSIS OF INTERNAL STRUCTURE

Our preliminary results compare distance metrics (Fig. 2), explore embedding quality (Fig. 3) and visualize the synthetic knowledge graphs (Fig. 5).

In the figure below, we compute the distance between pairs of random code snippets in CodeBERT latent space and find a positive correlation with various string distance metrics.

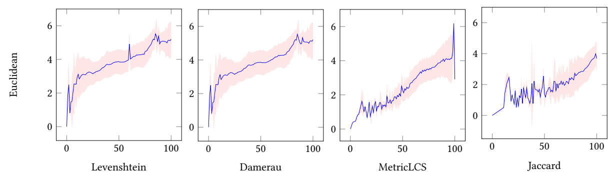

Figure 2: CodeBERT latent space vs. string edit distance.

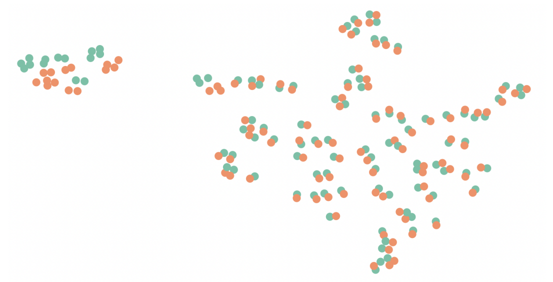

Figure 3: TSNE-embedded code snippets before (green) and after (orange) synonym renaming was applied.

We fetch a dataset of Java and Kotlin repositories sampled repositories on GitHub, containing a mixture of filetypes representing source code and natural language artifacts. From each repository, we index all substrings of every line in every file using a variable height radix tree producing a multimap of `kwdIndex: String → List<Lo` of `String` queries to file-offset pairs. We also encode CodeBERT [3] sequence embeddings to substrings `knnIndex: Vector → Location<F,` using a Hierarchial Navigavble Small World Graph [7] (HNSWG).

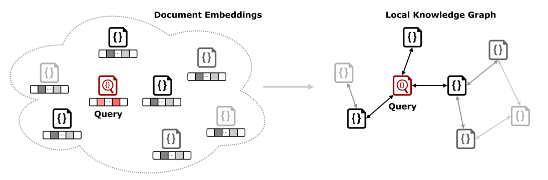

Figure 4: To compute our query neighborhood, we traverse the HNSWG up to max depth d, i.e. $d = 1$ fetches the neighbors, $d = 2$ fetches the neighbors-of-neighbors, and $d = 3$ fetches the neighbors-of-neighbors-of-neighbors.

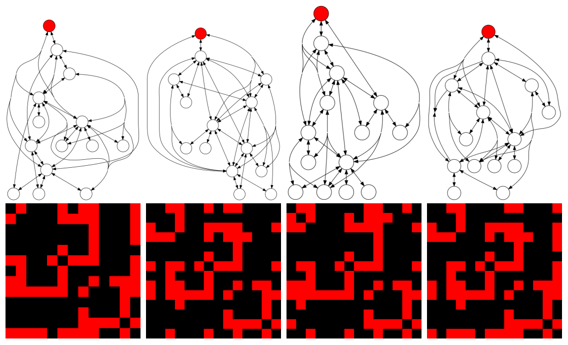

Figure 5: Virtual knowledge graph constructed by searching Euclidean nearest-neighbors in latent space. Edges represent the k-nearest neighbors, up to depth-n (i.e., k=5, n=3).