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, **??** 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 [**?**]. 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 [**?**] 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

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 whose semantics are 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: String→String`, a set of code snippets, `snips: Set<String>`, a single cosmetic transformation, `ctx: String→String`, and a code completion metric, `d: (String, String)→Float`, we would expect the model's accuracy to remain unchanged after the transformation was applied. Specifically,

2 METHOD

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 the appendix

Code transformations...

How to evaluate source code transformations:

Comparing different types of transformations

Combining different transformations or bounds?

Single vs. multiple transformation benefits (cite testing literature?)

Code characteristics: different categories of code based on... selection: slicing, tokenization.. language: java, kotlin size: method length complexity: halstead, cyclometric

3 METHOD

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;
}
```

4 EXPERIMENTS

In this work, we attempt to understand the relationship between entities in a software project. Our research seeks to answer the following questions:

(1) Which contexts in a software project share mutual information?
(2) To what degree can we claim the model has learned to:
    (a) Locate contextually relevant artifacts within a software project?

Model	IRA	QDL	P/R
CodeGPT [?]	X	X	X
GraphCodeBERT [?]	X	X	X
CodeBERT-small [?]	X	X	X
RoBERTa-Java [?]	X	X	X
Copilot[?]	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

For each of these models (available on HuggingFace), we sample a set of code snippets from our synthetic dataset, and compare how well they agree on each task.

In contrast with classical code completion models which only require a file-local context, our method is designed to navigate an entire project. In the following experiments, we compare completion accuracy with a vanilla sequence prediction model, as well as an AST-structured sequence prediction model trained from a corpus of Java projects on the same task.

We hypothesize that by jointly learning to choose locations in a project over which to attend while solving a downstream task, such as masked sequence completion, our model will produce a feature representation capable of locating and extracting information from semantically relevant contexts. We evaluate our hypothesis both qualitatively and quantitatively.

In our first set of experiments, we try to understand what is shared between sequences mapped to similar latent vectors. Do similar sequences share salient keywords? Are those keywords relevant to the task?

In our second experiment, we try to measure the information gain from including and excluding various filetypes through ablation. For graphs containing filetypes which include Markdown or Java, what kinds of information do these resources provide and which categories are most salient?

In our third and final set of experiments, we compare performance across hyperparameters. Does contextual expansion lead to better task performance for a given sequence prediction task? By relaxing edge-construction criteria and increasing hyperparamers such as beam search budget, we would expect corresponding task performance to increase.

If our hypothesis is correct, the virtual knowledge graph will span both natural language and source code artifacts. If so, this would provide evidence to support the broader hypothesis [?] that documentation is a useful source of information. In addition to being useful for the prediction task itself, we anticipate our model could also be used for knowledge graph extraction and suggest semantically relevant code snippets to developers.

5 EVALUATION

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"))
```

B ANALYSIS OF INTERNAL STRUCTURE

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

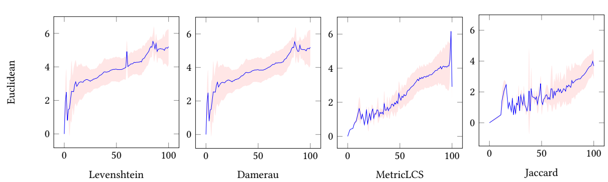

Figure 1: CodeBERT latent space distance correlates with string edit distance.

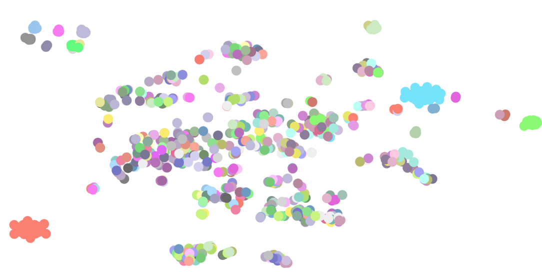

Figure 2: Reduced dimensionality TNSE embeddings colored by line length.

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

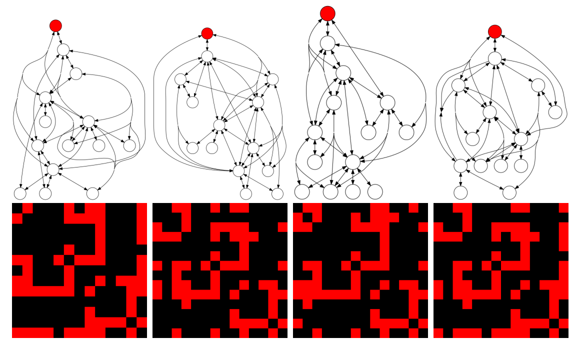

Figure 3: Virtual knowledge graph constructed by visiting nearest-neighbors in latent space.