How sensitive is neural code completion to syntactic variance?

Anonymous Author(s)

ABSTRACT

Neural language models hold great promise as tools for computer-aided programming, but questions still remain as to their reliability and the consequences of overreliance on those tools. 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 thus needed to understand their behavior on programming-related tasks. In this work, we develop a methodology for systematically evaluating neural language models using simple source code transformations such as synonymous renaming, intermediate logging, and independent statement reordering. Applying these transformations to natural code snippets, we evaluate three state-of-the-art models, CodeBERT, GraphCodeBERT and RobertA-Java, which exhibit varying degrees of robustness. Our approach is implemented and released as a modular and extensible toolkit for evaluating the performance of neural code completion models.

ACM Reference Format:
Anonymous Author(s). 2022. How sensitive is neural code completion to syntactic variance?. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Statistical language models play an increasingly synergetic role in software development, most prominently in code completion [1]. Yet from a user's perspective, the behavior of these models is opaque: partially completed source code written in 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 apparent.

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 certain *invariants* or *metamorphic relations* are known, one can generate an effectively unlimited amount of data by selecting and recombining synthetic transformations. For example, computer vision models should be invariant to shift, scale and rotation, so given a small dataset of labeled images, we can apply these transformations to obtain far more data for training or evaluation. Do 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

One would expect code completion models to exhibit invariance to certain refactorings. For example, code completion should be invariant to synonym variable renaming or reordering dataflow-independent statements.

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) **Dead code**: introduce dead code to source code
(3) **Statement reordering**: swap non-interfering statement order
(4) **Loop bounds fuzzing**: change loop bounds conditions
(5) **Permute argument order**: scramble method arguments
(6) **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.

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 [7]	X	X	X
GraphCodeBERT [4]	X	X	X
CodeBERT-small [3]	X	X	X
RoBERTa-Java [6]	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

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 [5] 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

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] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14. IEEE, 2017.

[6] 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.

[7] 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.

A SLICING PROCEDURE

We describe below a simple heuristic for extracting method slices in well-formed source code using a Dyck counter. 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. 1), explore embedding quality (Fig. 2) and visualize the synthetic knowledge graphs (Fig. 3).

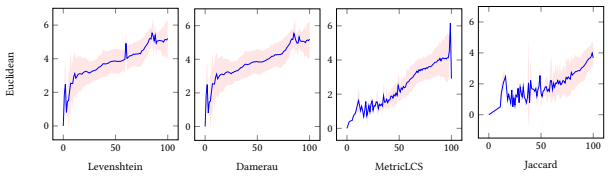

Figure 1: CodeBERT latent space distance correlates with string edit distance.

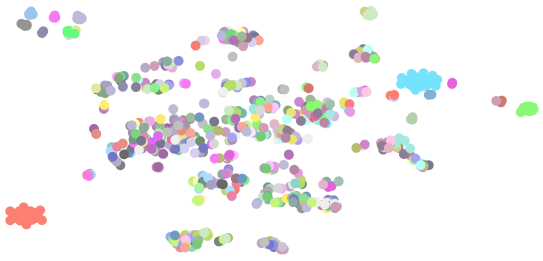

Figure 2: Reduced dimensionality TNSE embeddings colored by line length.

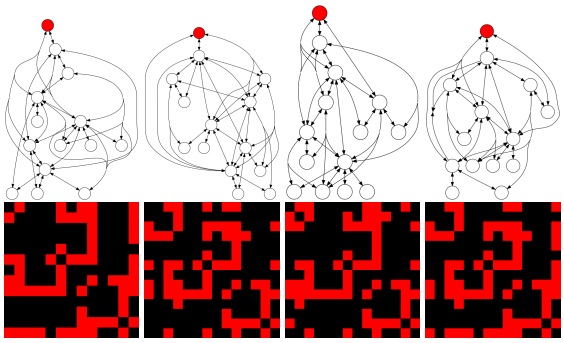

Figure 3: Virtual knowledge graph constructed by visiting nearest-neighbors in latent space.