# Probing natural language models of source code

Breandan Considine, Xujie Si, Jin Guo
{breandan.considine,xujie.si,jguo}@mail.mcgill.ca
McGill University

## ABSTRACT

Neural networks are increasingly capable of performing end-to-end document comprehension and retrieval. Training these models, however, is equivalent to reading every document in the dataset and tends to produce models which are (1) uninterpretable, (2) disregards user intent and (3) modification is expensive and requires retraining. We would like to produce a model which can retrieve contextually relevant search results based on user intent, available at inference time, and whose query model can be interpreted and modified by the user. The model used to express intent is a symbolic automaton: the model reads the user's context and produces a default query, which the user can modify to filter various types of results.

**ACM Reference Format:**
Breandan Considine, Xujie Si, Jin Guo. 2018. Probing natural language models of source code. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

What is a language model? Disregarding meaning, a language model is no more than a statistical model of symbolic information. Natural languages differ from mechanical languages in a few key ways. Natural languages are linear: almost all human languages have a fixed maximum recursion depth governed by biological and cognitive factors. Mechanical languages, by contrast, and are typically nonlinear due to their metalinguistic properties. Thus, natural language models cannot represent a mechanical language without significantly constratining their expressiveness.

One might argue – since natural language models are effectively linear – a language model with a large enough working memory, given enough data, should be able to model language fragments whose description length fits inside their working memory. Indeed, recent literature has shown that natural language models are surprisingly adept at modeling idioms in source code. However, we would expect such models to struggle with fragments whose information complexity exceeds its working memory (e.g. trees and graphs whose average MDL stretch the limit).

What is the difference between working memory, i.e. reasoning, and long-term memory, i.e. knowledge? Long term memory is information stored in the learned parameters, i.e. the topological structure of the network – as a model *learns*, this information is passively encoded in the topological structure. Long-term mememories must be conserved by the data distribution. Distributional shift tends to erase past memories, a phenonomena known as catastrophic forgetting.

In contrast, short-term memories are dynamical patterns of *activity* which must be actively consereved by the network topology. This second mechanism, called working memory, is a dynamical process with a much smaller information footprint, but allows a network to reason about incoming signals and adapt to previously unseen scenarios without memorization. To be passed on to successive layers, working memories must be actively consereved by the network topology. Both learning and reasoning are a forms of message passing over a graphical structure: learning shapes the graph topology, and reasoning propagates signals through it.

Models which cannot fit a language into working memory must learn maximal-length framgents and use long-term memory to fill in the gaps – i.e. by memorizing transitions between fragments in long-term memory. If this is the case, we would expect to find transitions that are locally consistient, but globally nonsensical. For instance, can source code models learn balanced parenthesis? Balanced intermingled parenethesis? `{([])}` Other algebraic datastructures (e.g. bush, imbalanced trees, etc.)? What are the limit languages that can be learned by natural langauge models of source code (e.g. Transformers)?

Early work in program learning realized the importance of graph-based representations [1], however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal [4], graphical [13] or other user interface to explore the space of valid programs, however do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph [9] from source code, but still require an explicit parser to form the initial graph and adaption to settings where style, terminology and document structure vary remains a challenge.

Furthermore, unlike prior work, our work considers the surrounding project and related artifacts. Instead of parsing their contents explicitly, which may be computationally intractable, we allow the agent to construct the graph organically by exploring the filesystem, then extract a query.

## 2 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.


*Woodstock '18, June 03–05, 2018, Woodstock, NY*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Our goal then, is to identify equivariant representations captured by the langauge model (LM). To do so, we generate program transformations (PTs) and measure vectorial equivariance in terms of 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 simple mutation testing tool that generates source transformations from each of these three categories.

Imagine that we have a stochastic language model that takes a sequence and emits a distribution over sequences:

$LM : Seq\langle Word\rangle \to Dist\langle Seq\langle Word\rangle\rangle$

But how do we obtain LM in the first place? A good language metamodel then, would produce language models that tend to generalize, or align with user preferences:

$LMM : Dist\langle Seq\langle Word\rangle\rangle \to \texttt{LM}$

For many computational languages, the number of models that explain our data far outnumber the size of the dataset, e.g. depending on the language complexity, there can be an exponential, or super-exponential number of models that exactly describe the data. If we wish to consider models that approximate the data, the number of models is effectively infinite. How do we form a distribution over models we cannot even enumerate? For small trees, it may be possible to perform beam-search, but for any reasonably sized query, we simply cannot enumerate all possibilities.

Furthermore, models typically average across all authors, needs, intents and situations. Language model designers are forced to compress human needs into the model training algorithm, which does not work very well. Effectively, we select a higher-order model which produces models that tend to align with human-selected priors. Since these priors are often hard to translate, there must be more human input during the design process. Most of these preferences are incorporated implicitly during the meta model design phase in the dataset selection and feature design.

$LMM : (Intent) \times (TrainingData) \to LM$

Rather than push all the needs and intent into the model training, we argue that these needs should be incorporated into the model at inference time. This is essentially the goal of meta learning, to factorize these probabilities into separate inputs.

$LM : Seq\langle Word\rangle \times (Intent) \to Dist\langle Seq\langle Word\rangle\rangle$

If we want to be able to make better code assistants, we need to incorporate the preferences of the human at runtime. To develop more reusable and human adaptive models, we need a way to train a model that can incorporate the user's preferences at runtime. Instead of trying to anticipate the users needs a priori, we need to be able to produce a model that can take some input from the user (say a query) and emit an answer:

$LM : (Seq\langle Word\rangle) \times (Query) \to Dist\langle Seq\langle Word\rangle\rangle$

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 four or five different language models, and compare their zero-shot generalization on synthetically-generated patterns. We generate code snippet pairs corresponding to manually-identified idioms, and measure inter-rater agreement between language models. Each model rates the similarity between manually-designed code snippets which vary along specific dimensions. Each model represents a human rater, which compares two code snippets for similarity.

We fetch a dataset of 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<Location<F, O>>` of `String` queries to file-offset pairs. We also encode CodeBERT [5] sequence embeddings to substrings `knnIndex: Vector -> Location<F, O>` using a Hierarchial Navigavble Small World Graph [12] (HNSWG).

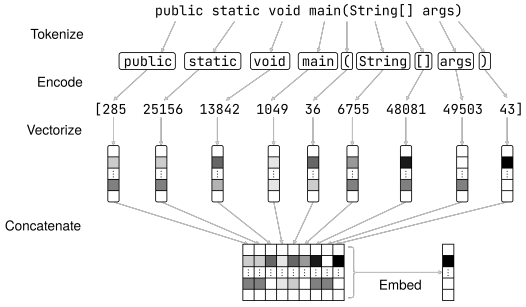


**Figure 1: CodeBERT takes a unicode sequence and emits a vector sequence, which we accumulate into a single vector.**

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 § 5, 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.

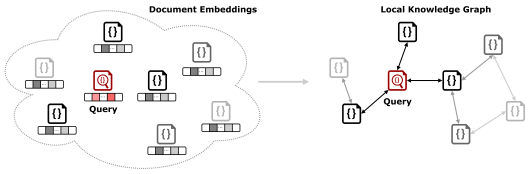


**Figure 2: To compute our query neighborhood, we traverse the HNSWG up to max depth $d$, i.e. $d = 1$ fetches the neighbors, $d = 2$ fetches the neighbors-of-neighbors, and $d = 3$ fetches the neighbors-of-neighbors-of-neighbors.**

For a given query and context, we first compute the context embedding and using a distance metric, fetch the k-nearest neighboring documents in latent space, forming the depth-1 nodes in our graph, then repeat this procedure recursively, pruning with a beam search heuristic based on the total path length in latent space. This procedure is depicted in Fig. 2.

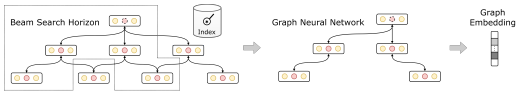


**Figure 3: Unlike language models which directly learn the data distribution, our model is designed to query an unseen database only available at test time. The model scores and selectively expands a subset of promising results within the database using a beam search heuristic, then runs message passing over the resulting graph to obtain the final task prediction.**

Consider the depth-1 beam search procedure (Fig. 4): We can use either vector search or keyword search to perform the context expansion, although vector search has the disadvantage of needing to rebuild the vector embedding index at each step during training. Keyword search is comparatively cheaper, as it only needs to build once for each repo on MiniGithub.

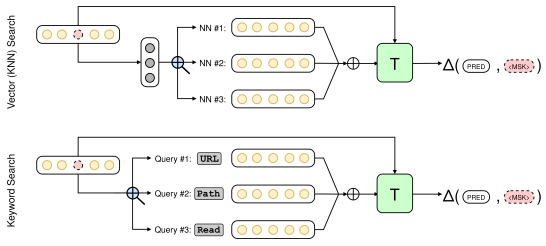


**Figure 4: In the vector search procedure, we compute an embedding for the masked source context, then retrieve the k-nearest sequence embeddings in our database. In the keyword search, we compute an n-hot vector to select the keywords, and retrieve the nearest matching sequences in our index. In both cases, we embed the results, feed them into a transformer, compute a loss between the predicted output and the masked token, and backpropagate.**

Once the beam search procedure is complete, we have a graph representing the neighborhood of the nearest neighboring code fragments in the parent project. This forms a so-called *virtual knowledge graph*, with nodes decorated by the CodeBERT sequence embeddings and edges decorated with the direction vector between documents. We then run $p$ rounds of message passing to obtain graph embedding or *neighborhood summary vector* (Fig. 3).

## 3 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 [11] | X | X | X |
| GraphCodeBERT [6] | X | X | X |
| CodeBERT-small [5] | X | X | X |
| RoBERTa-Java [10] | X | X | X |
| Copilot[2] | 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 performence 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 [7] 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.

## 4 EVALUATION

We evaluate on the following loss function, adapted from [8]. Given a set of code and context pairs, $\mathbf{F} = (\mathbf{c}_i, \mathbf{d}_i)_{i=1}^{N}$, where $\mathbf{S} \sim_{i.i.d.} \mathbf{R}_d$:

$$\mathcal{L} = \frac{1}{N} \sum_{i}^{N} \log \left( \frac{\langle \varphi(\mathbf{c}_i), \varphi(\mathbf{d}_i) \rangle}{\sum_{j \neq i}^{N} \langle \varphi(\mathbf{c}_j), \varphi(\mathbf{d}_i) \rangle} \right) \tag{1}$$

In other words, we minimize the distance between true code and context pairs, while maximizing the distance between distractors $(\mathbf{c}_j, \mathbf{d}_i)_{i \neq j}$.

## 5 RESULTS

Our preliminary results compare distance metrics (Fig. 5), explore embedding quality (Fig. 6) and visualize the synthetic knowledge graphs (Fig. 7).

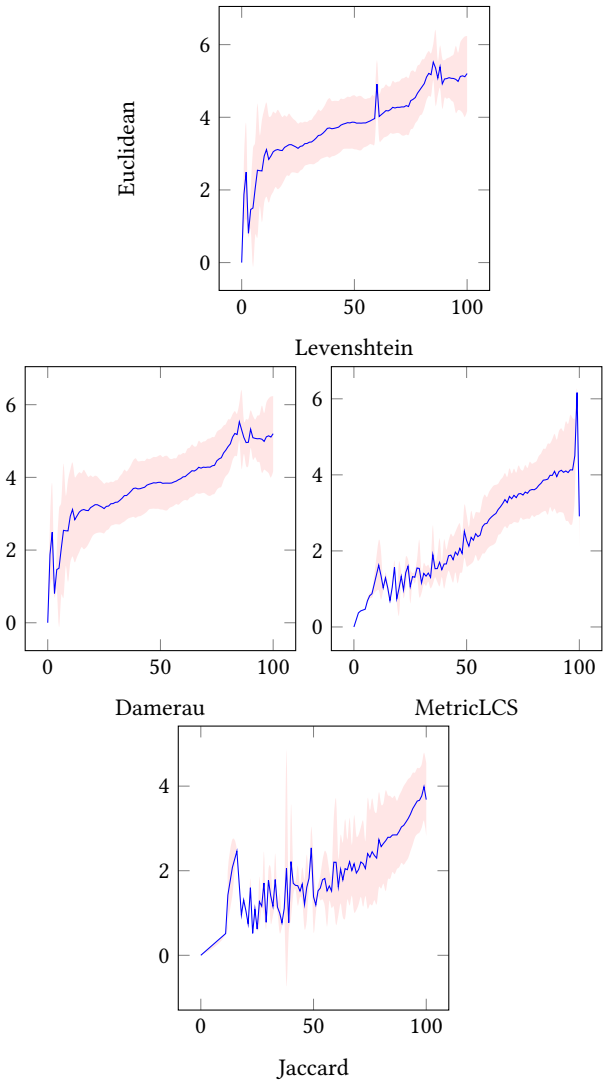


**Figure 5: CodeBERT latent space distance correlates with string edit distance.**

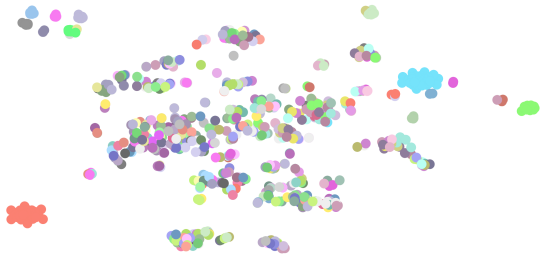

**Figure 6: Reduced dimensionality TNSE embeddings colored by line length.**

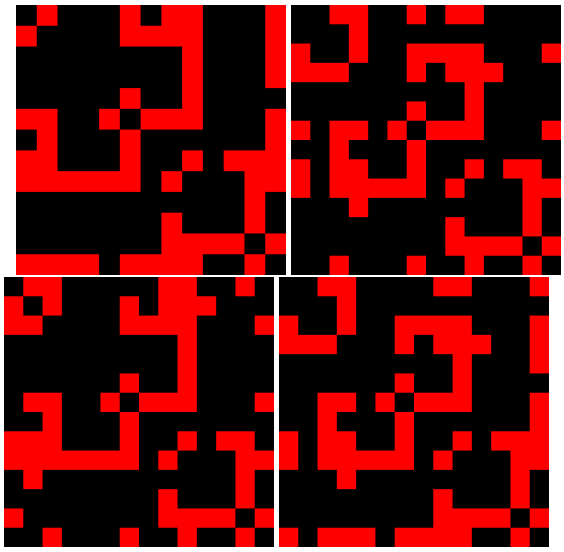

**Figure 7: Example graphs sampled by beam search with $d = 10$, $w = 5$.**

## REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, Will Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[3] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. *arXiv preprint arXiv:1906.04604*, 2019.

[4] 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.

[5] 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.

[6] 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.

[7] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[8] Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

[9] 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.

[10] 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.

[11] 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.

[12] Homer Walke, R Kenny Jones, and Daniel Ritchie. Learning to infer shape programs using latent execution self training. *arXiv preprint arXiv:2011.13045*, 2020.