Language Models for Proofs

Anonymous Author(s)

ABSTRACT

Proofs play a key role in reasoning about programs and verification of properties of systems. Mechanized proof assistants help users in developing proofs and checking the consistency of proofs. These systems allow users to express the proofs in the proof language developed by the systems. In this paper, we analyze proofs written by two different proof assistants (Coq and HOL) to investigate if proofs follow a language model. More specifically, whether there are recurring patterns in the proofs. Moreover, since Coq and HOL are based on different theoretical foundations, our results determines if different theoretical foundations can influence the naturalness of proofs. Our results show ....

ACM Reference format:
Anonymous Author(s). 2018. Language Models for Proofs. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, Florida, United States, 4–9 November, 2018 (ESEC/FSE 2018),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Fix:1

1 INTRODUCTION

Proofs have played a key role in discovering and capturing knowledge since the dawn of mathematics. The power of formal logic to express complex proofs into multiple provably correct steps has been unlocked for computer science by languages such as Prolog (for first-order logic) and, more recently, by interactive frameworks for higher-order logic languages that can automatically establish many simpler theorems [1]. Such frameworks have been used to proof tens of thousands of theorems, each of which can in turn be used as dependencies in other theorems. Formal verification of program properties is greatly facilitated by proof languages as well; languages like Coq [4] have been used to proof many program properties, even verifying an entire C compiler (CompCert [9]).

Much like other programming languages, writing proofs is also tedious and time-consuming. Real-world proofs often consist of many steps and depend on a wide vocabulary of other theorems. To make matters worse, whereas most programming languages will compile programs that may fail on some inputs (i.e. run-time errors), thus allowing more incremental development, proofs must be universally sound. Developer assistance tools can be of great help given these constraints. At present, this predominantly comes in the form of automated sub-theorem proving (for instance, Coq includes keywords like `auto` and `intuition` [4]). Other languages have benefited from automated code completion [12], fault localization [11] and rewriting [2]. All of these can be unlocked by language models, which capture and utilize typical repetitive patterns in bodies of text, including code. If such patterns exist in typical proofs as well, we may bring a substantial range of assistance tools within reach of proof writing and hopefully simplify this process. In this work, we establish that this is indeed the case.

ESEC/FSE 2018, 4–9 November, 2018, Lake Buena Vista, Florida, United States

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

2 LANGUAGE MODELS

Language models judge the fluency of a text by comparing its statistics to those observed in a large corpus of real-world expressions. For instance, an n-gram model may collect occurrence statistics of sequences of n adjacent words from a collection of newspaper articles and compare these to sequences that occur in a new article. If the new article is relatively unpredictable given the past data, this indicates that it its style of writing is atypical, possibly containing errors. Vice versa, a language model may be queried for suggestions in a context; if it adequately "understands" the context, it should recommend useful completions of e.g. a word or sentence.

In practice, n-gram models are often used as baseline models because they are fast to estimate and remarkably accurate. They accomplish this by only considering contexts of $n-1$ tokens, where n is typically around 5, which tends to be the most informative bit of context. This does ignore many long-range dependencies, however, so Deep learning models, based on recurrent neural networks, have more recently come to dominate the state-of-the-art in many natural language modeling tasks. These models abstract context into opaque latent states that theoretically allows them to carry dependencies over long distances (though practically they tend to be more limited). In source code, both types of models have proven highly successful: in general, tokens (e.g. punctuation, keywords, identifiers) in programming languages are more predictable than those in natural languages. n-gram models can especially benefit from locality (e.g. tokens in the same file), allowing them to outperform deep learning models in some settings [6].

We use both (dynamic) n-gram and recurrent neural network models. Specifically, we use SLP-Core[1] to implement n-gram models, both plain and with file-cache components. We implement recurrent neural network language models in CNTK,[2] using a fairly typical architecture with 300-dimensional embeddings and two 650-dimensional, GRU activated hidden layers, with a residual connection. We apply drop-out regularization (50% likelihood of disabling a hidden neuron at training time) and batch-normalization to the second hidden layer. The network is trained across 10 epochs with an initial learning rate of 0.002 per token that is decayed by half every epoch starting epoch 5. On account of the small vocabularies, we found performance to be substantially better when the model was allowed to back-propagate gradients across more time-steps; typically the recurrence is only "unfolded" for 20 to 50 words, but

[1] https://github.com/SLP-team/SLP-Core
[2] https://cntk.ai/

we achieved substantially better results at 100 steps. Because the corpora are fairly large (HOL's is much larger than typical for code) but the vocabularies are small, We also used relatively large minibatch sizes, to 10 thousand for Coq and 20 thousand for HOL.

We fix the vocabulary for the deep learning model by treating all tokens that are seen only once at training time as a generic "unknown" token. Please note that we leave the vocabulary open for the *n*-gram models unless otherwise specified and shall account for this discrepancy in our analysis; please see related refer to Hellendoorn & Devanbu for a discussion on how the necessity of a closed vocabulary artificially inflates the neural network's performance [6].

2.1 Metrics

Cross-entropy is our primary metric (typically just called entropy) because it is commonly reported in language modeling work. This is an information-theoretic metric, which captures the predictability of a text in terms of the number of additional bits needed to transfer each token to a receiver if they had access to the same language model. Specifically, it averages the negative log-likelihood across all tokens in the corpus, which reflects their geometric mean predictability.[3] Lower values thus imply higher predictability, with the optimal entropy score being 0, at which point a model experiences zero "surprisal" at observing any next token. Typical values for natural languages lie in the range of 5 - 8 bits (where 5 bits is currently the domain of both very large corpora and neural networks) and values for code are in the range of 1-4 bits [6, 7]. Entropy serves well as a general indication of predictability, but may not perfectly reflect predictability between languages. This is because it is averaged per token: more verbose languages (like Java) will tend to use more (predictable) syntax-related tokens and may achieve lower per-token entropies than less verbose languages (like Haskell). If the goal is to compare between languages, which ours is not, cumulative entropy for equivalent functionality might be a better indicator, but parallel corpora needed to evaluate such a metric are very rare.

Prediction accuracy allows us to approximate how useful a code completion tool could be for a language by recommending a list of tokens in every context. Typically, accuracy is reported based on where the correct item appears in an ordered list of suggestions. Top-1 accuracy captures how often the correct item is also the top-ranked item only (we will mainly report this metric); top-5 and top-10 accuracies allow the suggestion to appear anywhere in that respective range to be counted correct and thus implicitly assume that a user will look at those top-*k* items before choosing a completion or typing the token themselves. Often, the most simple tokens are also the most predictable, so that top-*k* accuracy scores may be deceptive; it is unlikely that a programmer will ask for a 1-character completion. We will also report the relative *character savings* at top-1/5/10: the percentage of characters in the file that a programmer would save (not have to type) when the correct item is in the top-*k* suggestions.

[3]This is more stable than the arithmetic mean given the wide range of probabilities that may be encountered.

Figure 1: An example of a COQ proof

Figure 2: An example of HOL proof

Table 1: Statistics of the datasets considered

Dataset	# files	# tokens	# unique tokens
Coq	3,589	9,817,678	37,813
HOL	11,410	107,576,235	1,837

3 CORPORA

To capture the types of proofs that are commonly written, we use two major data sources consisting of Coq proofs, and higher-order logic proofs.

3.1 Coq

Coq is an automated theorem prover based on Calculus of Inductive Construction(CIC) that is widely used for reasoning about programs. Proofs in CIC are lambda expressions typed with propositions. When the propositions removed, the result is a lambda expression that expresses its associated algorithm.

1 Through Curry-Howard isomorphism [**?**] the types systms of Coq allows propositions to be expressed as types. In CIC, each proof is a lambda expression typed with proposition, and removing propositions, will result in a lambda expression that describes the algorithm.

Figure 1

3.2 Higher Order Logic

Figure 2

A large corpus of HOL proofs was released as part of a machine learning challenge to choose the correct next step in a proof; this dataset is called HolStep [8]. In the original challenge, the authors collect real proofs, each starting with some (typically simple) dependencies and followed by a number of "positive" (real) and "negative" (fake) steps. The original proof contained the positive steps only, and the task of a machine learner in this challenge was to decide which steps were real and which were fake. The authors registered some reasonable success rates (ca. 80%) using convolutional neural networks, suggesting that there is indeed something "natural" about real proof steps, although the type of negative examples may be a substantial confound.[4] We are not interested in the challenge task itself, but instead use the positive steps from their proofs as training data. This gives us 10 thousand training proofs (with nearly 99 million tokens) and 1,410 test proofs (with ca. 8.5 million tokens),

4 RESULTS

We present the entropy results for our various models in Table 2. In this section, we compare the predictability (and aspects of it) of both proof languages to other programming languages.

[4]We found negative examples to be significantly more repetitive than positive ones; such inherently different characteristics may allow deep neural networks to distinguish between real and fake steps without any insight into the actual proof

Table 2: Entropies (in bits) and top-1 prediction accuracy on the proof corpora. Character savings gives an indication of typing effort reduction through the fraction of characters that would not need to be typed by choosing the top-1 completion if correct.

	Entropy		Top-1 Acc.		Char. savings	
	Coq	HOL	Coq	HOL	Coq	HOL
n-gram	3.52	2.49	55.7%	55.5%	46.9%	56.1%
n-gram+cache	2.28	1.28	70.2%	78.8%	63.0%	80.8%
RNN	3.03	1.72	56.6%	68.6%	46.5%	67.9%
RNN w/o UNK*	–	–	56.2%	68.6%	45.6%	67.9%

*RNN for which suggesting the generic unknown token is not considered accurate

4.1 Performance Characteristics

Looking at the per-token entropy values, both Coq and HOL proofs are relatively predictable compared to natural languages, where *n*-gram entropies tend to be around 6 to 8 bits per token. Coq's entropy of 3.52 bits/token is quite similar to typical entropies for Java [7]. HOL's entropy is substantially lower, although we caution the reader against assigning too great an importance too the difference for several reasons: (1) HOL's vocabulary is much smaller (and its training corpus larger), (2) the entropy is measured per-token, so that more verbose languages tend to score lower (e.g. Java is less entropic than Python by this measure), and (3) we are not comparing the entropy of these two languages, just aiming to understand their characteristics in the larger context of programming language "naturalness".

The RNN model outperforms the plain *n*-gram model by a sizable margin, but is outperformed by an *n*-gram model with a cache component, which is in line with previous results on modeling code [6, 15]. Its prediction accuracy, however, is about the same as the plain *n*-gram model's. This matches insights from prior work, which found that the RNN mainly achieves low average entropy by being very confident in correct predictions (thus achieving effectively zero entropy on those), whereas the *n*-gram model is more "reserved" due to smoothing. As a result, the entropy gain of the RNN over the plain *n*-gram model does not translate into any useful improvements in prediction accuracy.

Completion accuracy on Coq shows artifacts of inflation relative to actual character savings: less typing effort is saved than suggested by the top-1 accuracy alone across all models. This means that relatively short tokens are more predictable when offering completions on those is unlikely to save a developer effort. However, the difference is relatively small, possibly because very few long (over 10 characters) tokens were found in the Coq data, whereas other programming languages will often contain many more such tokens. Performance on the HOL dataset did not show a discrepancy at all, in fact yielding slightly better character savings than prediction accuracy. The (few) relatively long tokens in HOL proofs are references to dependencies use by the proof and are apparently relatively predictable. This suggests promise for a code completion tool for HOL proofs.

4.2 Vocabulary

The vocabulary of both proof systems is remarkably low, much lower than both natural languages and programming languages considering their corpus sizes. For instance, a 16M token Java corpus in prior work had a vocabulary of ca. 200,000 tokens [6]. Part of the reason for this appears to be that variables in proofs follow highly repetitive naming conventions, almost exclusively `f0`, `b0`, `b1` etc. Besides these, the vocabulary contains a small number of syntactic tokens (e.g. `"c/\"`, `"c~"` in HOL), but most of the vocabulary innovation comes from references to other theorems that are used as dependencies, which tend to have more meaningful names. Here, Coq proofs appear to be substantially less repetitive than HOL proofs.

Relatively small vocabularies tend to imply higher predictability, but we do note that this by no means explains the low entropy values *in ipso*. Even a vocabulary of a thousand tokens would yield an entropy of ca. 10 bits if those tokens were used at random. And the popular PTB English dataset has a vocabulary of just 10,000 words but a best entropy of over 6 bits. Instead, the vocabulary size may justify some of the lower entropy values of HOL compared to Coq, but a substantial inherent repetitiveness is responsible for the overall low entropy values.

The deep neural network uses a closed vocabulary when training and testing, meaning some tokens are treated as generic "unknowns". As a result, it uses a vocabulary of 24,427 tokens for the Coq data and 1,759 for the HOL data. This necessarily inflates its performance somewhat as these unknown tokens are fairly common and easy to predict while replacing tokens that are rare and hard to predict. Following Hellendoorn & Devanbu, we include an additional row in Table 2 that instead gives no reward for predicting the unknown token [6]. As can be seen this effect is quite mild in our data; prediction accuracy on Coq decreases somewhat (especially in terms of character savings because new tokens tend to be longer), but not as much as in prediction of Java in prior work. Performance on HOL does not change significantly at all because of its naturally almost completely closed vocabulary. Thus, proof languages may be a domain where deep neural network's difficulty learning new tokens is a very mild hinderance. However, the RNNs are still not able to achieve the accuracy that the *n*-gram models with explicit cache component can.

4.3 Locality

Both Coq and HOL entropies benefit substantially from a cache component: it more than halves surprisal in the both,[5] yielding an especially impressive improvement in HOL considering the already low entropy. Programming languages have been shown to exhibit a high degree of locality in their patterns, which is especially present in locally typical use of identifiers (e.g. only within one file, package or project) [6, 13]. However, HOL's vocabulary is already extremely small, implying that there is instead a high degree of locality in the organization of the arguments in the proofs themselves. This is further reinforced by comparison with a 1-gram cache (a cache that only stores tokens, not sequences): this model achieves an entropy of only 2.41 bits/token, which is substantially worse than what can be achieved by caching whole sequences.

[5] every bit of entropy is a factor two increase in geometric mean probability

Both datasets see a substantial increase in prediction accuracy and (even more so) relative character savings of the top-1 predictions. Since the longest tokens in most proofs are references to dependencies (other theorems), the cache component appears to successfully pick up on repeated use of those dependencies, which should be helpful for a developer in a code completion setting.

5 IMPLICATIONS

Premise Selection

Learning to guide proof [10]
HOLstep [8]
Guiding SMT solvers [5]

Representation of Proofs

Graph Representation [14] of HOL.
Dependency Graph for Coq[3].

REFERENCES

[1] HOL Interactive Theorem Prover. https://hol-theorem-prover.org/#about.

[2] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 281–293.

[3] Bertot, Y., Pons, O., and Pottier, L. Dependency graphs for interactive theorem provers. *Rapport de recherche, INRIA* (2000).

[4] Coquand, T., and Huet, G. The Coq Proof Assistant. https://coq.inria.fr/.

[5] Graham-Lengrand, S., and Färber, M. Guiding smt solvers with monte carlo tree search and neural networks. *AITP 2018* (2018).

[6] Hellendoorn, V. J., and Devanbu, P. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 763–773.

[7] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on* (2012), IEEE, pp. 837–847.

[8] Kaliszyk, C., Chollet, F., and Szegedy, C. Holstep : a Machine Learning Dataset Higher - Order Logic Theorem Proving. *ICLR* (2017), 1–12.

[9] Leroy, X., et al. The compcert verified compiler. *Documentation and userâĂŹs manual. INRIA Paris-Rocquencourt* (2012).

[10] Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. Deep Network Guided Proof Search. In *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning Deep* (2017), vol. 46, pp. 85—-105.

[11] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 428–439.

[12] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Acm Sigplan Notices* (2014), vol. 49, ACM, pp. 419–428.

[13] Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 269–280.

[14] Wang, M., Tang, Y., Wang, J., and Deng, J. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems* (2017), pp. 2783–2793.

[15] White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on* (2015), IEEE, pp. 334–345.