Exploring the Naturalness of Buggy Code with Recurrent Neural Networks

Jack Lanchantin JJL5SW@VIRGINIA.EDU
University of Virginia, Department of Computer Science

Ji Gao EMAIL@COAUTHORDOMAIN.EDU
University of Virginia, Department of Computer Science

Abstract

Statistical language models are powerful tools which have been used for many tasks within natural language processing. Recently, they have been used for other sequential data such as source code. (Ray et al., 2015)

1. Introduction

Natural language is inherently very well understood by humans. There are certain linguistics and structures associated with natural language which make it fluid and efficient. These repetitive and predictive properties of natural language make it easy to exploit via statistical language models. Although the actual semantics are very much different, source code is also repetitive and predictive. Some of this is constrained by what the compiler expects, and some of it is due to the way that humans construct the code. Regardless of why it is predictable, it has been shown that code is accommodating to the same kinds of language modeling as natural language (Hindle et al., 2012).

The language modeling task is defined as estimating the probability of a sequence of words (or tokens). Formally, given a sequence of tokens S a language model attempts to estimate the probability of S occurring in the language via the following equation:

$$P(S) = P(s_1) \prod_{i=2}^{N} P(s_t | s_1, s_2, ..., s_{t-1})$$

Where the conditional probabilities $P(s_t | s_1, s_2, ..., s_{t-1})$ model the probability of token s_t occurring given all previous tokens $s_1, s_2, ..., s_{t-1}$.

Proceedings of the 32^nd International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

From a distribution such as a language model, we can measure the entropy, or amount of uncertainty of a given sequence S. Using this metric, we can determine particular sequences which are "unnatural" with respect to the language.

Recently, (Ray et al., 2015) showed that it is possible to predict buggy lines of code based on the entropy of the line with respect to a code language model. In this work, the authors proposed a $cachelanguagemodel$, which is an extension of an $n - gram$ language model to handle local regularities in a piece of code which is being examined for bugs. They provide extensive experimentation to show that entropy of code can successfully be used to determine buggy lines similar to, and in some cases better than previous state-of-the-art bug localization tasks.

The main drawback of their paper is that they use a typical $n - gram$ language model, which struggles to handle long term dependencies due to the computation cost of a large n. There has been much recent work which has shown that recurrent neural network models are able to model languages with long term dependencies much better than previous techniques (Graves, 2013; Sutskever et al., 2014; Sundermeyer et al.; Karpathy et al., 2015).

$$P_{ngram}(s_t | s_1, s_2, ..., s_{t-1}) = P(s_t | s_{t-n+1}, ..., s_{t-1})$$

2. Related Work

2.1. Bug Dection

For software bug detection, there are two main areas of research: bug prediction, and bug localization.
Bug prediction, or statistical defect prediction, which is concerned with being able to predict whether or not there is a bug in a certain piece of code, has been widely studied in recent years (Catal & Diri, 2009). With the vast amount of archived repositories in websites such as github.com, there are many opportunities for finding bugs in code.
Static bug finders (SBFs), automatically find where in code

a bug is located. SBFs use properties of code to indicate locations of bugs. There has been a wide array of recent work in this area (Rahman et al., 2014), which use many pattern recognition techniques to find bugs. As noted in (Ray et al., 2015), using SBFs and using the entropy of language models are two very different approaches to achieve the same goal. The main goal of their work is to compare the effectiveness of language models vs SBFs for the same task of classifying buggy lines.

2.2. Natural Language Processing

There have been many works in NLP which use language models for sequence to sequence tasks, such as word completion, machine translation, and many others. There are also a variety of models which use word embeddings trained on language models in order to perform other tasks such as sequence classification (e.g. sentiment analysis).

Separately, there are a variety of models which do not use language models, but do sequence classifications using pattern recognition techniques. These would be similar to SBF techniques in software engineering. However, to the best of our knowledge, there have not been any works in NLP which use entropy from neural language models to classify sequences.

3. Methods

3.1. Recurrent Neural Network Language Model

Recurrent neural networks (RNNs) are models which are particularly well suited for modeling sequential data. At each time step t, an RNN takes an input vector $\mathbf{x_t} \in \mathbb{R}$ and a hidden state vector $\mathbf{h_{t-1}} \in \mathbb{R}^m$ and produces the next hidden state $\mathbf{h_t}$ by applying the following recursive operation:

$$\mathbf{h_t} = f(\mathbf{W}\mathbf{x_t} + \mathbf{U}\mathbf{h_{t-1}} + \mathbf{b}) \tag{1}$$

Where $\mathbf{W} \in \mathbb{R}^{m\times n}, \mathbf{U} \in \mathbb{R}^{m\times m}, \mathbf{b} \in \mathbb{R}^m$ are the learnable parameters of the model, and f is an element-wise nonlinearity. The parameters of the model are learnt via backpropagation through time. Due to their recursive nature, RNNs can model the full conditional distribution of any sequential distribution. However, RNNs suffer from what is referred to as the ”vanishing gradient” problem, where the gradients of time steps far away either vanish toward 0, or explode toward infinity, thus making the optimization of the parameters difficult.

To handle the exploding gradients problem, (Hochreiter & Schmidhuber, 1997) proposed Long Short-term Memory (LSTM), which can handle long term dependencies by using “gating” functions which can control when information

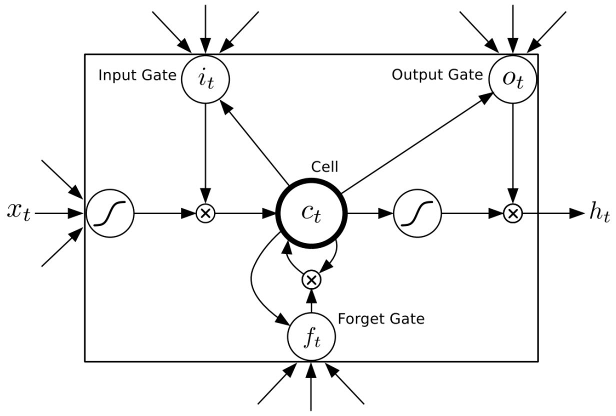

Figure 1. An LSTM Module

is written to, read from, and forgotten. Specifically, LSTM “cells” take inputs $\mathbf{x_t}, \mathbf{h_{t-1}}$, and $\mathbf{c_{t-1}}$, and produce $\mathbf{h_t}$, and $\mathbf{c_t}$ in the following way:

$$\begin{aligned}
\mathbf{i_t} &= \sigma(\mathbf{W^i x_t} + \mathbf{U^i h_{t-1}} + \mathbf{b^i}) \\
\mathbf{f_t} &= \sigma(\mathbf{W^f x_t} + \mathbf{U^f h_{t-1}} + \mathbf{b^f}) \\
\mathbf{o_t} &= \sigma(\mathbf{W^o x_t} + \mathbf{U^o h_{t-1}} + \mathbf{b^o}) \\
\mathbf{g_t} &= tanh(\mathbf{W^g x_t} + \mathbf{U^g h_{t-1}} + \mathbf{b^g}) \\
\mathbf{c_t} &= f_t \odot c_{t-1} + i_t \odot g_t \\
\mathbf{h_t} &= o_t \odot tanh(c_t)
\end{aligned}$$

Where $\sigma()$ and $tanh()$ are element-wise sigmoid and hyperbolic tangent functions. $\odot$ represents and element-wise multiplication. $\mathbf{i_t}$, $\mathbf{f_t}$, and $\mathbf{o_t}$ are referred to as the input, forget, and output gates, respectively. An overview of an LSTM module can be seen in Figure 1 It is the gating mechanisms that allow the LSTM to remember long term dependencies, which are especially important in code where a certain line may depend on code many lines back.

Using LSTMs in modeling source code is arguable more important than in the natural language modeling case. It has been shown that although computationally expensive, modest sized $n - gram$ models can model languages well, and can predict the next word well. However, this is due to the fact that natural language is fairly local. Most words (or characters) do not heavily depend on words farther than about 20 back. In the source code case, it is drastically different. For example, a function could be 20 lines (and thus $>$ about 200 tokens) long. The characters at the end of the function are heavily dependent on the characters at the beginning of the function, and possibly even lines before the function. Recently, it was shown by (Karpathy et al., 2015) that we can easily generate source code which appears to be written by a human, from a source code language model. This is most impressive because in the examples shown, the code is well indented, the braces and brackets are correctly nested, and even commented correctly. This is not something that can be achieved by looking at the previous n characters.

3.2. Experimental Setup

4. Results

5. Threats to Validity

6. Conclusion

References

Catal, Cagatay and Diri, Banu. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 837–847. IEEE, 2012.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Karpathy, Andrej, Johnson, Justin, and Li, Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

Rahman, Foyzur, Khatri, Sameer, Barr, Earl T, and Devanbu, Premkumar. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 424–434. ACM, 2014.

Ray, Baishakhi, Hellendoorn, Vincent, Tu, Zhaopeng, Nguyen, Connie, Godhane, Saheel, Bacchelli, Alberto, and Devanbu, Premkumar. On the" naturalness" of buggy code. *arXiv preprint arXiv:1506.01159*, 2015.

Sundermeyer, Martin, Schlüter, Ralf, and Ney, Hermann. Lstm neural networks for language modeling.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.