Using LSTMs to Model Programming Languages

Brendon Boldt BRENDON.BOLDT@GMAIL.COM
Marist College, 3399 North Rd. Poughkeepsie, New York 12601

Abstract

Recurrent neural networks (RNNs), specifically long-short term memory networks (LSTM) are particularly good at performing next word prediction on natural languages. This research investigates the ability for these same LSTMs to perform next word prediction on programming languages, namely the Java programming language. In order to be fed into the LSTM, Java source code had to undergo a transformation which preserved the logical structure of the source code and removed from the code various specificities such as variable names and literal values. A standard English corpora and four separate Java repositories were then tested with a standard LSTM. Results suggest that LSTM used can more effectively model the Java code than it can English. These findings could be useful in areas such as code prediction in IDEs or in automated code generation.

1. Introduction

Machine learning techniques of language modelling are often applied to natural languages, but many of the techniques used to model natural languages can be applicable to programming languages as well. One such an application of a language model is next-word prediction which can prove very useful for tasks from auto-completion to anomaly detection. There has been research into programming language models which use Bayesian statistical inference (n-gram models) to perform next-word prediction (Allamanis & Sutton, 2013). Yet some of the most successful natural language models have been built using recurrent neural networks (RNNs); their ability to remember data over long sequences makes them particularly apt for word prediction.

Preliminary work as part of the Marist College Honors Program Thesis project.

Specifically, long-short term memory (LSTM) RNNs have further improved the basic RNN model by increasing the ability of an RNN to remember data over a longer sequence of input without the signal decaying quickly (Zaremba et al., 2014). LSTMs are a sequence-to-word language model which means given a sequence of words (e.g., words in the beginning of a sentence), the model will produce a probability distribution describing what the next word in the sequence is. The equation below illustrates the basic structure of a sequence-to-word language model where L is the language model, w_i is the ith word in the sequence, and W_{n+1} is a vector describing the probability distribution describing which word w_{n+1} is.

$$L(w_1, w_2, w_3, \ldots, w_n) = W_{n+1} \tag{1}$$

We are specifically investigating next-statement prediction in method bodies. While other parts of Java source code (e.g., class fields, import statements) do have semantic significance, method bodies make up the functional aspect of source code[1] and most resemble natural language sentences. Just as individual semantic tokens (words) comprise natural language sentences, statements, which can be thought of as semantic tokens, comprise method bodies. Furthermore, the semantics of individual natural language words coalesce to form the semantics of sentence just as the semantics of the statement in a method body form the semantics of the method as a whole. By this analogy, language modelling techniques which operate on sentences comprised of words could apply similarly to method bodies comprised of statements.

2. Tokenizing Java Source Code

We are specifically looking at predicting the syntactic structure of next statement in within Java source code method bodies. The syntactic structure of a complete piece of source code is typically represented in an abstract syntax tree (AST) where each node of the

[1] Functional insofar as method bodies describe that actual behavior of the program.

tree represents a distinct syntactic element (e.g., statement, boolean operator, literal integer). Method bodies are, in particular, comprised of statements which, more or less, represent a self-contained action. Each of these statements is the root of its own sub-AST which represents the syntactic structure of only that statement. For this reason, the statements are the smallest independent, semantically meaningful unit of a method body and are suitable to be tokenized for input into the RNN.

Nguyen et al. (Nguyen & Nguyen, 2015) present a model for syntactic statement prediction called ASTLan which uses Bayesian statistical inference to interpret and predict statements in the form of sequential statement ASTs. While Bayesian statistical inference can be applied to statements directly in their AST form, RNNs operate on independent tokens such as English words. Thus, it is necessary that statement ASTs be flattened into a tokenized form in order to be used in an RNN language model.

2.1. Statement-Level AST Tokenization

The RNN model described in Zaremba et al. (2014) specifically uses space-delimited text strings, hence, when the statement ASTs are tokenized, they must be represented as space-delimited text strings.

To show the tokenization of Java source, take the following Java statement:

```
int x = obj.getInt();
```

The corresponding AST, as given by the Eclipse AST parser, appears in Figure 1.

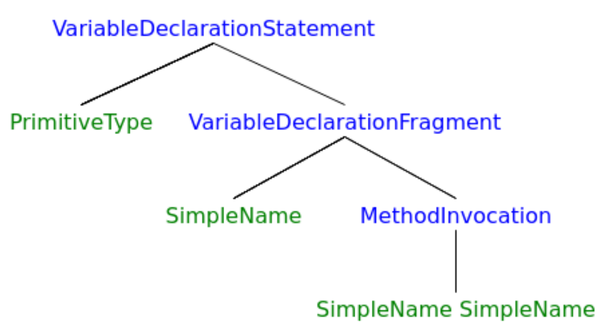

Figure 1. The abstract syntax tree (AST) representation of of the Java statement `int x = obj.getInt();`

This statement, in turn, would be transformed into the following token.[2]

```
_PrimitiveType_VariableDeclarationFr
agment(_SimpleName_MethodInvocation(_
SimpleName_SimpleName)))

_60(_39_59(_42_32(_42_42)))
```

In the actual representation, the AST node names are replaced with integers IDs, but we have included the named version to demonstrate how it fits in with the visual AST. Individual AST nodes are separated by underscores ("_") and parentheses are used to denote a parent-child relationship so that the tree structure of the statement is preserved. In fact, it is possible to recreate the syntax of the original source code from the the tokens; thus, this tokenization is lossless in terms of *syntactical* information yet lossy in other areas (variable and function names and the like are omitted to make the model more general).

2.2. Method-Level Tokenization

Now we can look at an entire method

```
int foo() {
    int x = obj.getInt();
    if (x > 0) {
        x = x + 5;
    }
    return x;
}
```

Each statement in the method body is tokenized just as the single statement was above where the tokens are space delimited. Braces, while not statements, are included (denoted by "{" and "}" to retain the semantic structure of the method body. Note that the return type and parameters are included as the first token with a leading "(" to denote that it is a method signature (no other statement tokens begin with an open paren).

```
(_39_42 { _60(_39_59(_42_32(_42_42)))
 _25(_27(_42_34) { _21(_7(_42_27(_42_
 34))) } _41(_42) }
```

The space-delimited sequence of these tokens forms a "sentence" which directly correlates to the body of a single Java method. These individual tokens will then comprise the vocabulary which the LSTM network uses to train and make predictions with.

[2] `VariableDeclarationStatement` is not included in the tokenized version of the AST since the syntax is adequately represented by starting with the root node's children.

Table 1. Total size of each corpora measured in words. The approximate split between training, validation, and test data is 80%, 10%, and 10% respectively.

Corpus	Size
PTB	1085779
JDK	303560
Guava	259686
ElasticSearch	561697
Spring Framework	526968

2.3. English and Java Source Corpora Used

Similarly to Zaremba et al. (2014), We are using the Penn Treebank (PTB) for the English language corpus as it provides an effective, general sample of the English language. For the Java programming languages, four different corpora were built from the source code of projects (one project is built into one corpus). The Java Development Kit (JDK), Google Guava, ElasticSearch, and Spring Framework. The JDK is a good reference for Java since it is largest implementation of the Java language; the other three projects were selected based on their high popularity on GitHub in addition to the fact they are Java-based projects.

It is important to note that the Penn Treebank does not contain any punctuation while the tokenized Java source contains "punctuation" only in the form of statement body-delimiting curly braces ("{" and "}") since these are integral to the semantic structure of source code.

2.4. Vocabulary Comparison

In addition to preserving the logical structure of the source code when tokenizing it, another goal of the specific method of tokenization was to produce a vocabulary with a frequency distribution similar to that of English (compared against the English corpora used, that is). If the same Java statement tokens appear too frequently, the tokenization might be generalizing the Java source too much such that it loses the underlying semantics. If the statement tokens, instead, all have a very low frequency it would be difficult to effectively perform inference on the sequence of tokens within the allotted vocabulary size.

In all of the Java corpora,, the left and right curly braces comprise 35% of the total tokens present. This a disproportionately high number in comparison to the rest of the tokens, but removing them from the frequency distribution, since they classify as punctuation, gives a more accurate representation of the vocabularies. The adjusted frequency distribution shown in Figure 2 compares the Penn Treebank to the Java Development Kit source code. The rate of occurrence for the highest ranked words is significantly higher in the JDK than in the PTB, but the frequency distributions track closely together beyond the fifth-ranked words.

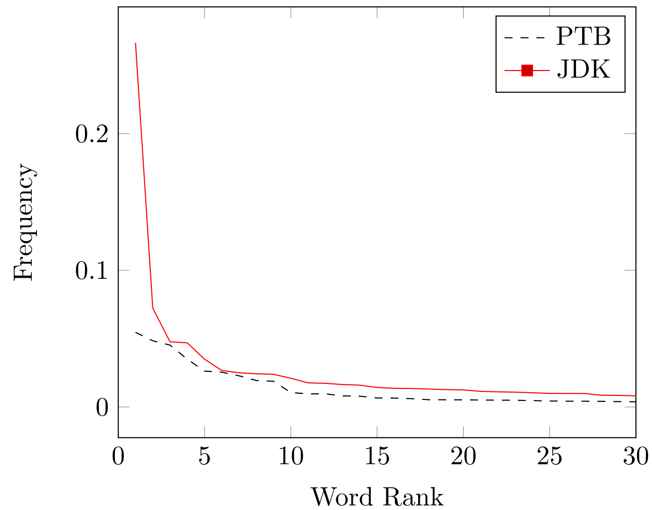

Figure 2. Comparison of English and Java word frequency distributions. The y-axis represents the total proportion of the word with a given rank (specified by the x-axis).

Table 2. Proportion and rank of the metatoken`<unk>`. Proportions and ranks are from the adjusted Java corpora with the left and right curly braces removed.

Corpus	Proportion	Rank
PTB	0.0484	2
JDK	0.0724	2
Guava	0.0476	5
ElasticSearch	0.1618	2
Spring Framework	0.0873	2

Adjusting the four Java corpora in the same way (removing the left and right curly braces) yields similar frequency distributions across all word ranks (see Figure 3).

Another consideration when comparing the English and Java corpora is the prevalence of the metatoken `unk` which denotes a token not contained in the language model's vocabulary. Due to the nature of LSTMs, the vocabulary of the language model is finite; hence, any word not contained in the vocabulary is considered unknown. We specifically used a vocabulary size of $10,000$. A vocabulary size which is too small will fail to represents enough words in the

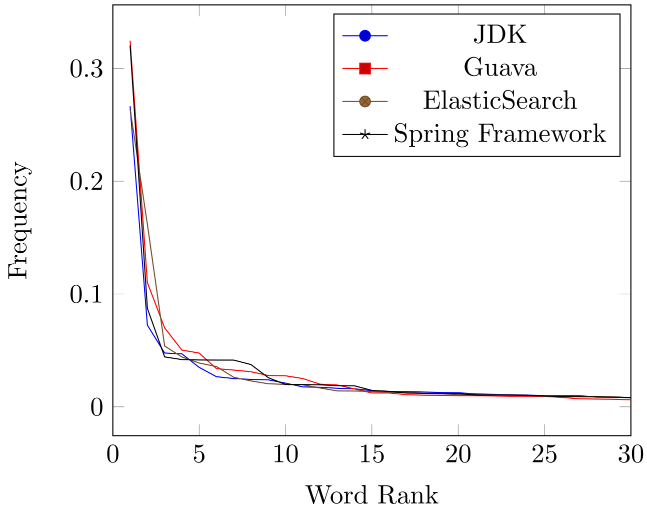

Figure 3. Comparison of Java corpora frequency distributions

corpus; the result is the LSTM seeing a high proportion of the unk metatoken. A vocabulary which is too large increases the computation required during training and inference. The proportion of `<unk>` tokens in both the English and the Java source data sets (save for ElasticSearch) are $< 10\%$ which indicates that the $10,000$ word vocabulary accounts for approximately 90% of the corpus' words by volume. It is important that the Java copora's `<unk>` proportion is not significantly higher than that of the Penn Treebank since that would suggest that $10,000$ is too small a vocabulary size to describe the tokenized Java source code.

3. Language Modelling

In order to make a good comparison between language modelling in English and Java, a model with demonstrated success at modelling English was chosen. The model selected was a long-short term memory (LSTM) neural network, a type of recurrent neural network (RNN), as described in Zaremba et al. (2014). This particular LSTM uses regularization via dropout to act as a good language model for natural languages such as English (Zaremba et al., 2014).

The LSTM's specific configuration was the same as the "medium" configuration described in Zaremba et al. (2014) with the exception that the data was trained for 15 epochs as the validation cost suggested that the model was overfitting past 15 epochs on the data sets tested. Notably, this model contains two RNN layers with a vocabulary size of $10,000$ words.

Each corpora was split into partitions such that 80% was training data and the remaining 20% was split evenly between test and validation data. Perplexity, the performance metric of the LSTM, is determined by the ability of the LSTM to perform sequence-to-word prediction on the test set of that corpus. Perplexity represents how well the prediction (in the form a probability distribution) given by the LSTM matches the actual word which comes next in the sentence. A low perplexity means that the language model's predicted probability distribution matched closely the actual probability distribution, that is, it was better able to predict the next word.

We chose word-level perplexity was chosen as the metric for comparing the language models' performance on the given corpora since it provides a good measurement of the models overall ability to predict words in the given corpus. Perplexity for a given model is calculated by exponentiatiting (base e) the opposite of the mean cross-entropy across all words in the test set.

$$P(L) = \exp\left(\frac{1}{N}\sum_{i=1}^{N} H(L, w_i)\right) \tag{2}$$

Where N is the test data set size, L is language model, w_i is the ith word in the test set, and $H(L, w)$ is the natural log cross-entropy from w to the prediction given by $L(w)$. A lower perplexity represents a language model with better prediction performance. The cross-entropy is calculated by summing the product of the probability of that word appearing (1 for the correct word and 0 for all other incorrect words) and the natural log of output value of LSTM's softmax layer.

$$H(L, w) = \sum_{i=1}^{V} p(w) \ln L(w) \tag{3}$$

Since the probability of all incorrect words is 0, the sum can be reduced to 1 times the the natural log of the probability of the correct word as given by the LSTM.

$$H(L, w) = \ln L_w(w) \tag{4}$$

4. Results

The results of running the LSTM on the data sets is displayed in Table 3. All four Java data sets showed a drastic reduction in perplexity compared to the English data set. This suggests that the LSTM was able to more accurately model the pre-processed Java source code than it could English.

Table 3. Perplexities given by Equation 2.

Corpus	Perplexity
PTB	85.288
JDK	21.808
Guava	18.678
ElasticSearch	11.397
Spring Framework	11.318

5. Conclusion

The pre-processed Java code represents a very general and cursory representation of the original code as it does not include anything such as variable names or variable types. Future research along these lines could account for information such as variable types, literal values, operator values, etc. Additionally, other machine learning methods like a naive Bayesian classifier could be paired with the LSTM to predict variable names as well as the syntactic structure of the next statement. It would also be beneficial to compare the modelling of Java with other programming languages or to train the model across multiple repositories in one language.

References

Allamanis, Miltiadis and Sutton, Charles. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pp. 207–216, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487127.

Nguyen, Anh Tuan and Nguyen, Tien N. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 858–868, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818858.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014. URL http://arxiv.org/abs/1409.2329.