Using LSTMs to Model the Java Programming Language

Brendon Boldt BRENDON.BOLDT1@MARIST.EDU
Pablo Rivas PABLO.RIVAS@MARIST.EDU
Marist College, 3399 North Rd. Poughkeepsie, New York 12601

Abstract

Recurrent neural networks (RNNs), specifically long-short term memory networks (LSTMs), 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. Java source code from four different data sets undergoes a transformation that preserves the logical structure of the source code and removes the code's various specificities such as variable names and literal values. Such datasets and an additional English language corpora are used to train and test standard LSTMs' ability to predict the next element in the sequence. Results suggest that LSTMs, under the proposed model, can effectively model Java code achieving perplexities under 22 and accuracies above 0.47, which is an improvement over LSTMs over the English language which achieves a perplexity of 85 and an accuracy of 0.27. This research can have further meaning in other areas such as syntactic template suggestion and automated bug patching.

1. Introduction

Machine learning techniques of language modelling are often applied to natural languages, but techniques used to model natural languages such as n-gram, graphed-based, and context sensitive models can be applicable to programming languages as well (Allamanis & Sutton, 2013) (Nguyen & Nguyen, 2015) (Asaduzzaman et al., 2016). One such an application of a language model is next-word prediction which can prove very useful for tasks such as syntactic template suggestion and bug patching (Nguyen & Nguyen, 2015) (Kim et al., 2013). 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 (Zaremba et al., 2014).

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. Equation 1 illustrates the basic structure of a sequence-to-word language model where L is the language model, w_n is the nth 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 se-

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

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

mantics 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 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) studied 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 produce an RNN-based 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 (Foundation).

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

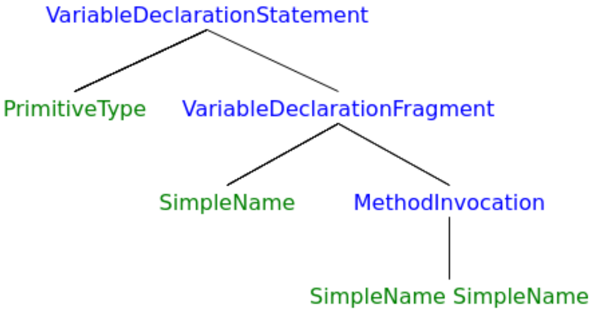

Figure 1. The abstract syntax tree (AST) representation of of the Java statement `int x = obj.getInt();`

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

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

In the representation used for the LSTM, the AST node names are replaced with integer IDs corresponding to each type of AST node (e.g., 60 corresponds to "PrimitiveType" nodes and 42 corresponds to "SimpleName" nodes). 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. For example, variable and function names are discarded during the translation to make the model independent of variable and function names.

2.2. Method-Level Tokenization

Now let us consider the following;

```
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

[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

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). The method above becomes the following sequence of tokens:

```
(_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 that the LSTM network uses to train and make predictions.

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 PTB 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.

While the PTB data set is larger than the Java corpora, all datasets are within one order of magnitude in size. The exact sizes of the data sets are given by Table 1.

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

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 approximately 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 PTB to the JDK 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.

The statistical similarities between the English and the translated Java corpora suggest that the Java statement tokens have an adequate amount of detail in terms of mimicking English words. If the Java statement tokens were too detailed, their frequencies would be far lower than those of English words; if the Java statement tokens were not detailed enough, their frequencies would be much higher than those of English words.

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).

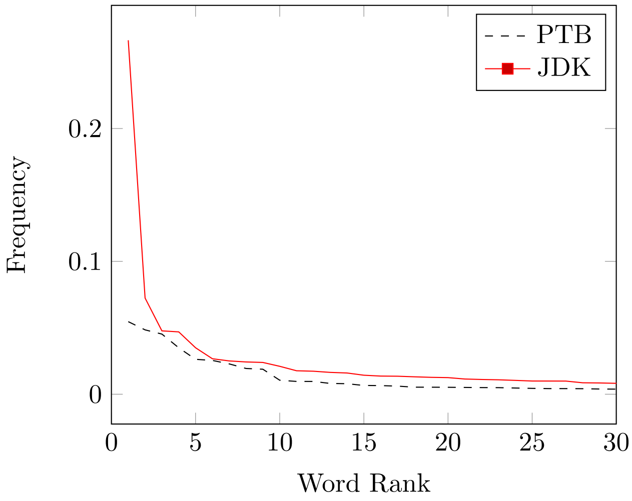

Figure 2. Comparison of English and Java word frequency distributions. The y-axis represents the total proportion of the word with a given rank (specified by the x-axis).

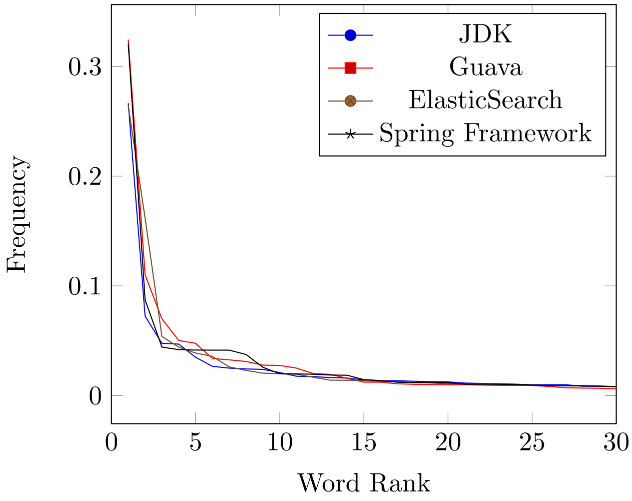

Figure 3. Comparison of Java corpora frequency distributions

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 represent enough words in the 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 PTB since that would suggest that $10,000$ is too small a vocabulary size to describe the tokenized Java source code.

3. Language Modelling

3.1. Neural Network Structure and Configuration

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 LSTM neural network, a type of 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 instead of 39 epochs. Beyond 15 epochs (on both the English and Java datasets), the training cost metric (perplexity) continued to decrease while the validation cost metric remained steady. This suggests that the model was beginning to overfit the training data and that further training would not improve performance on the test 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. Perplexity is the same metric that is used in Zaremba et al. (2014) to compare language models.

3.2. Language Model Metrics

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 (Sundermeyer et al., 2015). Perplexity for a given model is calculated by exponentiatiting (base e) the mean cross-entropy across all words in the test set. This is formally expressed as follows:

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

where N is the test data set size, L is the language model, w_i is the ith word in the test set, and $H(L, w_i)$ is the natural log cross-entropy from w_i to the prediction given by $L(w_i)$. A lower perplexity represents a language model with better predictiive performance (Wang et al., 2016).

The cross-entropy is the opposite of summing the product of the probability of that word appearing, i.e., 1 for the correct word and 0 for all other incorrect words, and the natural logarithm of the output value of LSTM's softmax layer. The cross-entropy is defined as follows:

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

where V is the vocabulary size and $p(w_i)$ is the probability of w_i being the correct word. Since the probability for incorrect words is 0 and the correct word is 1, the sum can be reduced to 1 times the the natural log of the probability of the correct word as given by the LSTM.

$$H(L, w) = -\ln L_w(w) \qquad (4)$$

where $L_w(w)$ is the LSTM's softmax output specifically for the word w. Additionally, the word-level accuracy was calculated for each language model considering the top 1, 5, and 10 predictions made by the model.

Table 3. Perplexities given by Equation 2.

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

Table 4. Proportion of predictions which had the correct word in their top k predictions. "ElasticSearch" is written as "ES" and "Spring Framework" is written as "SF".

Corpus	Top 1	Top 5	Top 10	Language
PTB	0.269	0.470	0.552	English
JDK	0.474	0.652	0.716	Java
Guava	0.519	0.696	0.751	Java
ES	0.576	0.739	0.784	Java
SF	0.560	0.722	0.783	Java

4. Results

The the perplexities achieved on the copora by the LSTM are displayed in Table 3. The smallest perplexity for non-english data sets was measured for the Spring Framework, while the largest was for the JDK data. The table also indicates that all four Java data sets showed a drastic reduction in perplexity compared to the English data set. Nonetheless, the perplexity achieved on the English dataset is similar to that reported by Zaremba et al. (2014). These results indicate the superiority of LSTMs on both programming languages and a language as complex as the English language.

Table 4 shows the top-k accuracies for each copora. Clearly, results suggest that the proposed LSTM model is able to more accurately model pre-processed Java source code than it can English. The table also indicates that, for the English data set, the use of a large number of predictors can dramatically increase the overall rate of predictors with the correct next word; e.g., increasing from one to ten predictors at least doubled the proportion of predictors. There is a similar effect over Java-based data sets; however, in these data sets the predictors start at a higher proportion than with English.

5. Conclusion

In this paper, we have presented a way of modelling a predictive strategy over the Java programming language using an LSTM. Using datasets such as PTB, JDK, Guava, Elastic Search, and Spring we have shown that LSTMs are suitable in predicting the next syntactic statements of source code based on preceding statements. Results indicate that the indicate that LSTMs can achieve lower perplexities and, hence, produce accurate models on the Java datasets than the English dataset.

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.

Asaduzzaman, Muhammad, Roy, Chanchal K., Schneider, Kevin A., and Hou, Daqing. A simple, efficient, context-sensitive approach for code completion. *Journal of Software: Evolution and Process*, 28(7):512–541, 2016. ISSN 2047-7481. doi: 10.1002/smr.1791. URL http://dx.doi.org/10.1002/smr.1791. JSME-15-0030.R3.

Foundation, Eclipse. Eclipse documentation on the AST class. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html. Accessed: 2016-8-18.

Kim, Dongsun, Nam, Jaechang, Song, Jaewoo, and Kim, Sunghun. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 802–811, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486893.

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.

Sundermeyer, Martin, Ney, Hermann, and Schlüter, Ralf. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(3):517–529, 2015.

Wang, Minsi, Song, Li, Yang, Xiaokang, and Luo, Chuanfei. A parallel-fusion rnn-lstm architecture for image caption generation. In *Image Processing (ICIP), 2016 IEEE International Conference on*, pp. 4448–4452. IEEE, 2016.

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