A Deep Learning Model for Source Code Generation

Raymond Tiwang
Department of Computer Science
Bowie state University
Bowie, Maryland, 20715, USA
tiwangr0611@students.bowiestate.edu

Timothy Oladunni
Department of Computer Science
University of the District of Columbia
Washington, DC, 20008, USA
Timothy.oladunni@udc.edu

Frank Xu
College of Public Affairs
University of Baltimore
Baltimore, Maryland, 21201, USA
wxu@ubalt.edu

ABSTRACT
Natural Language Processing (NLP) models have been used extensively to study relationship among words in a corpus. Inspired by models such as n-gram we developed a model for analyzing source code via its Abstract Syntax Tree (AST). This study has applications in source code generation, code completion, and software forensics and investigation. The study also benefits software developers and programmers striving to improve code efficiency and speed up development process. With source code analysis, the Natural Language Tool Kit (NLTK) which is very useful in NLP, becomes severely limited by its inability to handle the semantic and syntactic properties of source codes. Therefore, we processed the source code datasets as Abstract Syntax Trees (ASTs) rather than the source code text itself to take advantage of the information-rich structure of the AST. The proposed model is built on the deep learning-based Long Short-Term Memory (LSTM) and Multiple Layer Perceptron (MLP) architectures. Results from our intrinsic evaluation on a corpus of python projects have demonstrated its ability of effectively predicting a sequence of source code tokens and show an improvement over previous work in this field.

Keywords

Abstract Syntax Tree (AST), Natural Language Toolkit (NLTK), Recurrent Neural Network (RNN), Multiple Layer Perceptron (MLP), Long-Short Term Memory (LSTM)

1. *INTRODUCTION*

Several machine learning tasks deal with sequential data of various types. Recently it has become a regular practice for software programmers/developers to use integrated development environments (IDEs) to complete various software tasks. Source code completion has become a very useful feature of IDEs as it suggests the next probable token based on code already in the typing context. Prior work on code completion was based on compile time type information [*1*] to predict the next token. This strategy is efficient with statically-typed language like C++, java and other type defined languages, but encounters difficulty when used with dynamically-typed languages like python and JavaScript because there is no type declaration in these latter languages.

One way to apply code completion to dynamically-typed programming language is to build a Natural Language Processing (NLP) model [*2*,*3*] adapted to source code datasets. This approach has limitations because the tools used for NLP do not handle symbols as being important for the text. The most popular NLP model is the n-gram model which has been used to model software codes [*4*,*5*]. The problem with this approach is that it is limited in practice to short context windows and so may not capture long term dependencies.

In this model we improve the repetitiveness of the dataset by generating the abstract syntax tree of the source code and working with the dump file which is the pre-order traversal of the AST tree of the code. We train our model with a network of LSTM and MLP. Our approach outperforms previous models by a considerable margin.

The rest of the paper is organized as follows: in section 2, we discuss related work; section 3 presents the theoretical background. Sections 4 and 5 describe our methodology and experimental results respectively. Model performance was discussed in section 6. We conclude our work in section 7.

2. *RELATED WORK*

2.1 Literature Review

Code completion or code generation has been the subject of study by several researchers in recent times. Asaduzzaman, et al., [*6*], proposed a context-sensitive code completion approach. The proposed model was based on lightweight source code analysis and contextual information methodology for an efficient API method calls. Roos [*7*], proposed an N-gram language model approach for a fast and precise API code completion. According to him, the proposed model works in real time and achieves completion within seconds.

In [*8*] , the authors proposed a novel approach to code completion. A non-predefined abbreviated multiple input keyboard methodology was found to be more efficient than the conventional one keyword input. A 30% and 40% reduction in time and keystrokes respectively were achieved when compared to the conventional approach. The use of code effectiveness of completion tools was discussed in [*9*]. The authors explored the history of several developers on an integrated development environment. The outcome of the experiment showed that developers regularly performed repetitive task of inserting the same text.

Omar et.al., [*10*] proposed an active code completion methodology. Their experiment showed that a palette definition approach of expression is more efficient than the conventional one. In another study, Cross and Huang [*11*] focused on making

978-1-7281-0137-8/19/$31.00 ©2019 IEEE

the task of code completion a sequence to sequence prediction. This was achieved by either flattening the output or having a representation as a sequence of construction decision [*12*]. Li et. al. [*13*] conducted a study on code completion with neural attention and pointer networks. They developed a tailored attention mechanism capable of exploiting the structured information on a program's abstract syntax tree. In a study on Automatic Code Completion, Ginzberg et. al. [*14*] implemented a vanilla LSTM model to complete code generation tasks.

2.2 Sentences as probability

In a language model, an n-gram is a continuous sequence of items (words, characters …) from a given corpus. To illustrate the probabilistic model, let's consider the sentence S= "the car runs fast". We can evaluate the probability of sentence S as the joint probability of each individual word s_i in S.

$$P(S) = P(s_1, s_2, s_3, \ldots, s_n) \quad (1)$$

Using the chain rule (a generalization of the product rule) of conditional probability, this equation can be reduced to;

$$P(s_1, s_2, s_3, \ldots, s_n) = P(s_1)P(s_2|s_1)P(s_3|s_1, s_2,) \ldots P(s_n|s_1, s_2, s_3 \ldots s_{n-1}) \quad (2)$$

Using the actual words in our sample sentence, eq.(1) translates to:

$$P(\text{"the car runs fast"}) = P(\text{"the"}, \text{"car"}, \text{"runs"}, \text{"fast"}) \quad (3)$$

$$P(\text{the car runs fast}) = P(\text{"}the\text{"})P(\text{"}car\text{"}|\text{"}the\text{"}) P(\text{"}runs\text{"}|\text{"}the\ car\text{"})P(\text{"}fast\text{"}|\text{"}the\ car\ runs\text{"}) \quad (4)$$

Equation (3) calculates the joint probability that all the individual words in sentence S appear together. Equation (4) is an expression of equation (3) as conditional probability. As shown in equation (4), for example, the probability $P(\text{"}runs\text{"}|\text{"}the\ car\text{"})$ is the probability that *"runs"* appears knowing the bigram *"the car"* has already appeared. On the right-hand side of equation (4), each term of the product is an n-gram probability which can be estimated using the counts of n-grams in a corpus. The probability of the entire sentence is calculated by looking up the probabilities of each component part in the conditional probability. Because we cannot possibly compute the n-grams of every length, we apply the Markov Assumption to simplify the equation by assuming that future states of the model only depend on its current state. This consideration leads to the following simplification of our conditional probabilities.

$$P(\text{the car runs fast}) = P(\text{"}the\text{"})P(\text{"}car\text{"}|\text{"}the\text{"}) P(\text{"}runs\text{"}|\text{"}car\text{"})P(\text{"}fast\text{"}|\text{"}runs\text{"}) \quad (5)$$

In Equation (5), with the Markov assumption, the probability $P(\text{"}fast\text{"}|\text{"}the\ car\ runs\text{"})$ from equation (4) simplifies to $P(\text{"}fast\text{"}|\text{"}runs\text{"})$. All other probabilities in the expression reduce to that of a bigram. Given a dataset composed of source code text, the individual words are replaced by the code tokens which could be variables, keyword, symbols etc. all representing different classes for the classification problems.

3. THEORETICAL BACKGROUND

The study under consideration is a case of multi-class classification where the number of classes is the number of distinct tokens in the vocabulary of our dataset. One of the most useful ways of representing this type of classifier is by using a set of discriminant functions $g_i(x)$, *for* $i=1,\ldots,c$. The classifier is said to assign a feature vector x to class w_i if

$$g_i(\mathbf{x}) > g_j(\mathbf{x}) \quad for\ all\ i \neq j \quad (6)$$

The classifier function as defined in equation (6) is a network or machine that computes *c* discriminant functions and selects the category with the largest discriminant.

In classification problems, especially in sequence problems, the occurrence of a token is usually associated with one other token. In the set of *c* classes, an action α_i is usually interpreted as the decision that the observed token belongs to the class ω_i. Suppose action α_i is taken when the observed token is ω_j, then the decision is correct if $i=j$ and incorrect if $i \neq j$. The goal is to minimize the probability of error i.e., reduce the error rate.
The risk corresponding to the loss function is the average probability of error. The loss function we used in our case is categorical cross-entropy.

The **Cross-Entropy Loss** (CE) Loss is defined as:

$$CE = -\sum_{i=1}^{C} t_i \log(s_i) \quad (7)$$

Where *C* is the number of classes, t_i and s_i are the real value and the RNN prediction score respectively for each class *i* in *C*. Usually an activation function (Sigmoid / SoftMax) is applied to the scores before the CE Loss computation, so we write $f(s_i)$ to refer to the activations, therefore the equation (7) is rewritten as:

$$CE = -\sum_{i=1}^{C} t_i \log(f(s)_i) \quad (8)$$

where $f(s)_i = \frac{e^{s_i}}{\sum_{i=1}^{C} e^{s_i}}$ is the output of the SoftMax classifier.

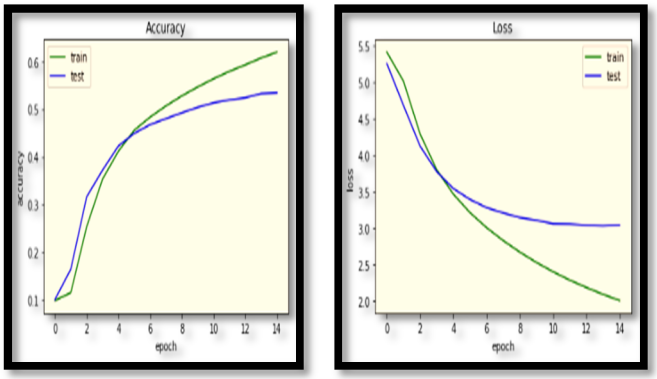
In general, the final layer of the neural language model for a multi class classifier, is a SoftMax classifier which is used to output the most probable token learned by the system. The token generated is part of learned vocabulary, having a dimension defined by the dimension of the embedding layer's dense output vector. In our model for probabilistic classification, we aim to map the model's inputs to probabilistic *predictions*. As is the norm, we *trained* our model by incrementally adjusting the model's parameters so that our predictions get closer to the *ground-truth probabilities.* Thus, the use of *categorical cross entropy* since we are working with multiple classes.

4. METHODOLOGY

4.1 Natural Language Processing Approach

We began our investigation by applying the ubiquitous Natural Language Processing (NLP), approach. Raw python source code datasets were trained and tested using LSTM as the learning algorithm. The dataset was obtained from a large GitHub repository with 1274 python source codes. In the preprocessing stage, the files containing the source codes were concatenated

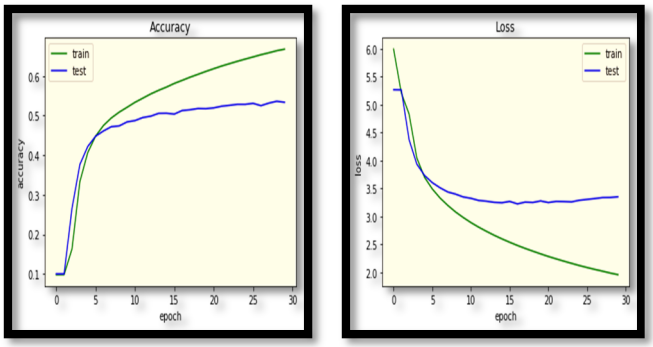
to produce one big file. The dataset was further cleaned by removing redundant spaces and other meaningless characters. After the cleaning, we tokenized and organized the dataset into sequences of fixed length. We then fed the sequences through the embedding layer of our model. Figure 1 and 2 show the result with 15 epochs and 30 epochs respectively.



-a- -b-

Figure 1: Accuracy and loss for 15 epochs, without AST data

As shown, the model with 15 epochs produced an accuracy rate of 53.43 % with a loss of 3.0382. The model with 30 epochs yields an accuracy rate of 53.36% with a loss of 3.3532.



-a- -b-

Figure 2: Accuracy and loss for 30 epochs, without AST data

Based on the above approach, the result showed a very low accuracy of correctly predicting the next token for completing a source code statement. This demonstrates that the source code syntax, structure and semantics have a significant effect on its predictability. A source code may consist of any number of variables which may be different in nature, type or length. The implication is that the conventional natural language processing (NLP) methodology has a limited scope in source code pattern recognition or knowledge discovery.

4.2 Proposed Approach

In view of the short comings of the NLP, we explored the application of abstract syntax tree for source code generation. We argue that this is a better approach. Using the same datasets as in the section 4.1, we pre-processed it using the abstract syntax tree (AST). We loaded the pre-processed data, defined a reference dictionary for all unique tokens of the vocabulary, stored into a list and split each line into individual tokens. Since the model only takes integer values for input, we encoded the tokens with a custom encoder. After encoding, the dataset which is now ready for training, is split into input-output proportions using sklearn. The resulting processed data was then ready to be fed into model for training, testing and evaluation.

4.3 Experimental design

We chunked the training dataset into windows of *n* tokens where the first *n-1* tokens are used as input and the n^{th} token is the output. The diagram in Figure 3 shows our experimental design and explains the process which we used to collect, clean, process the data and train the model.

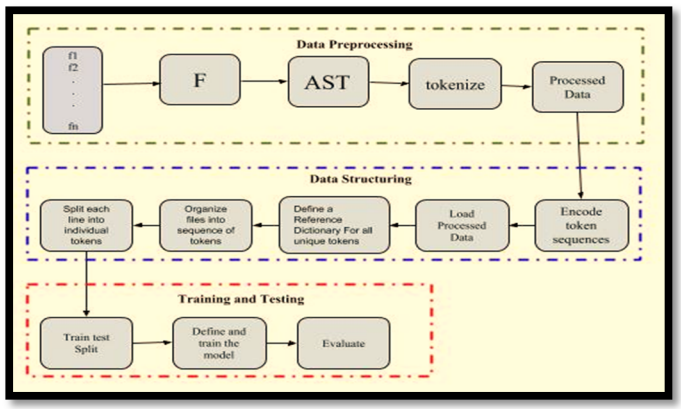


Figure 3: Experimental Design

The experimental design in Figure 3 is composed of three main phases: the data processing phase, the data structuring phase and the training and testing phase. In the data processing phase, source codes (data) was loaded into the model. For effectiveness, loaded files were transferred into one large python object F. The AST was generated from the unique file F. In the data structuring phase, we prepared the data for training by organizing the text into tokens and created a dictionary for each unique token. After tokenizing and structuring the data, we split the text into training and testing data and feed it into the model.

4.4 AST Processing

In this code generation learning model, word or character in traditional character-based model was defined to capture the syntax, structure and semantics of source code tokens. In this model tokens represent any unique unit of the code, it could be a symbol (e.g. '[', ')', '{'), a keyword (e.g. "for", "and", "or"), a variable name, a string literal, a number literal, an operator (+,-,/,*) or any other singular expression.

In an AST processed source code, each node of the tree has a predetermined structure depending on the type of keyword or operation involved. For example, leaf nodes are usually a function name, target name or argument name. Each of these leaf nodes is identified by an identification (*id)* and a context (*ctx*). The *id* can be a string variable, number literal or native function while *ctx* represents the task that the name performs. Possible values of *ctx* are Load, Store, Del , etc. Take for example the node *Assign* in the tree in Figure 4. The figure shows that it has one target of type *name* whose *id* is *"y"* and the *ctx* is *Store*. The number of unique types is relatively small compared to the size of the generated corpus (datasets).

While there is infinite choice of values for the *id* which essentially is the value of the field in consideration, the structures of the nodes are fixed and easily predictable. So, whenever *id*

appears, regardless of its values, we should expect a *ctx* to follow and which also has a limited number of values it can take, thus making it highly predictable also. Representing the program by its AST tree rather than the plain text source code allows for easy prediction of the program structure and thus ensures better accuracy of next token generation.

The use of the AST tree is essential in our model because it creates a consistent structure that fits every source code and assures regularity of the text format that's easily trainable. Figure 5 below shows the AST tree for the following simple code:

```
for x in range(10):
    y=2*x+1
    print(y)
```

Figure 4 Sample code

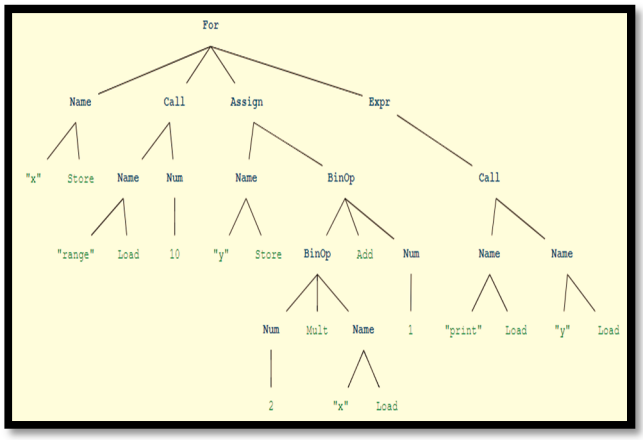


Figure 5: Python source code and its corresponding abstract syntax tree (AST)

The tree on Figure 5 when flattened, produces the text in Figure 6 below.

"Module(body=[For(target=Name(id='x', ctx=Store()), iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)], keywords=[]), body=[Assign(targets=[Name(id='y', ctx=Store())], value=BinOp(left=BinOp(left=Num(n=2), op=Mult(), right=Name(id='x', ctx=Load())), op=Add(), right=Num(n=1))), Expr(value=Call(func=Name(id='print', ctx=Load()), args=[Name(id='y', ctx=Load())], keywords=[]))], orelse=[])])"

Figure 6: Corresponding flattened abstract syntax tree (AST)

In the box in Figure 4, we show a sample python program and its corresponding abstract syntax tree (AST) in Figure 5. The flattened tree structure is shown in Figure 6. It represents the pre-order traversal of the AST tree in which each node is identified by its type. The tree comes with keywords and predetermined fields, which are properly interpreted by the compiler for execution. As a result, the presence of these keywords adds to the size of our corpus considerably, but the vocabulary size does not increase significantly. This is because each keyword is counted only once when building the vocabulary. The generated dump from the source code, suggests that the multiple occurrences of the same keywords in AST provides a better ratio of corpus size to vocabulary size. This makes the AST file more trainable than the raw source code. In the corpus each program is represented by its corresponding AST. The processed dump file can be converted back to source code in a one to one correspondence using astunparse module or astor.

4.4.1 RNN-LSTM Learning

Although typical RNN could be used to capture long term dependency in a sequential model, it suffers from exploding or vanishing gradient. This is a challenge for training for very long sequential datasets. Therefore, the LSTM variant of RNN was used for the experiment. The LSTM was first introduced by Hochreiter et. al. [*15*] to resolve the problem of vanishing gradients. The attractiveness of the LSTM resides in its design. It is built with sigmoid gates designed to scale or filter out information that the network should retain. After every time cycle, it decides what information and how much of it to keep or discard. The standard structure of an LSTM unit is shown in Figure 7.

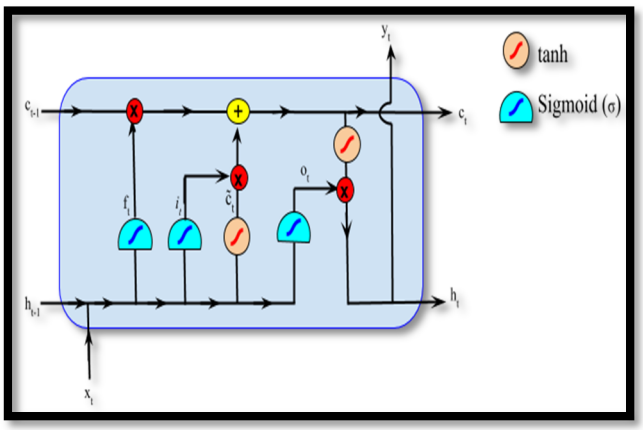


Figure 7: LSTM structure with forget gate

The previously integer-encoded tokens were fed into the embedding layer of the keras model. The output sequence from the embedding later was input into an LSTM layer. The LSTM unit consists of an input gate i_t, a forget gate f_t, and an output gate o_t. These gates each take two input vectors; the input data x_t and the output h_{t-1} from the previous time step. The vectors are processed as describe in equations (9-11).

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \quad (9)$$

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \quad (10)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \quad (11)$$

$$\tilde{c}_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (12)$$

$$c_t = f_t \bullet c_{t-1} + i_t \bullet \tilde{c}_t \quad (13)$$

$$h_t = o_t \bullet tanh(c_t) \quad (14)$$

As shown in equations (9 – 14) σ is the sigmoid function, $\bullet$ the element-wise product, the subscript t stands for the time

step. $W_f, W_i, W_o, and\ W_c$ represent the weight matrices at each gate. The bias terms are represented as b_f, b_i, b_o, b_c for each gate. The input vector to the LSTM unit is represented by x_t. The activation vector for the forget gate and the input gate are represented by f_t and i_t respectively. The output vector o_t is the output gate. *input node* $\tilde{c}_t$ eq.(12) takes the activation from the input layer x_t at the current time step and output from the previous time step h_{t-1}. The summed weighted input is run through a *tanh* activation function. The *input gate* eq.(10) takes the weighted sum of the current data point x_t and the output from the previous time step h_{t-1} as argument of a sigmoid function σ which ensures that unwanted values are filtered. At the *forget gate* f_t eq (9), features that are not relevant for the next step are flushed using the sigmoid function and an update is generated for the internal state c_t eq.(13) of the LSTM cell. Equation (14) is the content of the internal memory that will be used to evaluate and update the internal state as new information is received in the next time step.

4.4.2 MLP Learning

We continued the experiment by training, testing and evaluating the processed dataset using the Multiple Layer Perceptron (MLP) learning algorithm. This is a more complex artificial neural network which consists of several nodes, multiple input features and activation functions. In between the output and input layers is the hidden layer. The hidden layer may be multiple layers of nodes. When a machine learning task becomes too complex to be handled by the normal machine learning techniques like logistic regression, linear regression, support vector machine, scientists turn to MLP. An MLP is a multi-layer perceptron that has the same structure as the single layer perceptron but with more than one hidden layer. The backpropagation algorithm consists of two phases: the forward phase where the activations are propagated from the input to the output layer, and the backward phase, where the error between the actual and the predicted value in the output layer is propagated backwards. The backward propagation evaluates and modifies the weights and bias values to reduce the error. Figure 8 below shows the sketch of a multi-layer perceptron with n-dimensional input and k hidden layers

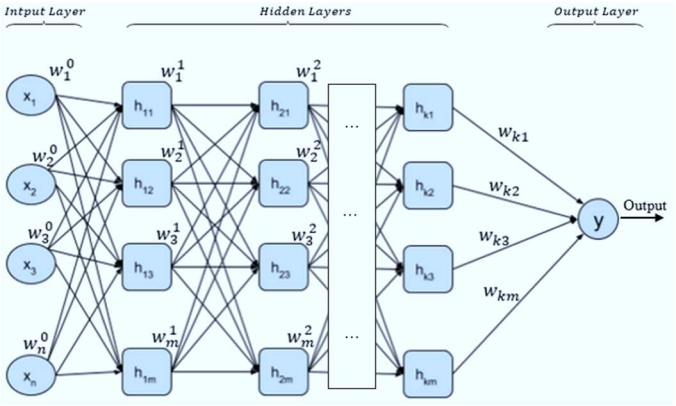
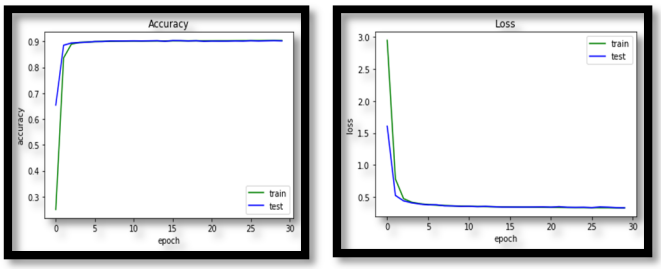


Figure 8: A Multiple Layer Perceptron

The structure of the MLP in figure 8 is represented by a directed graph whose nodes are the neurons, and each edge in the graph connects the output of some neurons to the input of another one [*16*]. This neural network can be described by a directed acyclic graph G = (V, E) and the weight function over the edge $w: E \rightarrow \mathbb{R}$. Each neuron is modeled as a scalar function $f: \mathbb{R} \rightarrow \mathbb{R}$. The function f is called the activation function.

5. RESULT

Using the AST dump of the source code, the LSTM and MLP learning algorithms produce the graphs in figures 9 and 10 respectively.



-a- -b-

Figure 9: Accuracy and loss for AST-LSTM

As shown in figure 9, the LSTM model's accuracy was 90.32% and a loss of 0.3505. in addition to using the AST dump for training the data, we also factor a regularization as one the hyper-parameters. Running the model with L2 regularization of 0.1 not only makes the system test appropriately, that is gradually grows with the training curve, but it also maintains a good accuracy of 90.32%. Regularization is an important technique in machine learning that helps to prevent overfitting. Mathematically speaking, it adds a *regularization term* to the loss function in order to prevent the coefficients to fit so perfectly that they cause overfitting. The L2 regularization which is also called ridge regression is the penalized sum of the square of the weights added to the loss function. The generic equation of this function for a neural network with L2 is a combination of an unregularized objective and a regularization term;

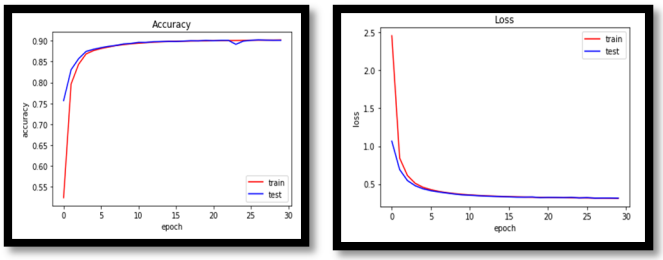
$$L_\alpha(\omega) = L(\omega) + \alpha||\omega||^2 \qquad (15)$$

The L1 regularization called LASSO regression adds the sum of the penalized weights to the loss function. And is expressed as follows:

$$L_\alpha(\omega) = L(\omega) + \alpha||\omega|| \qquad (16)$$

In equations (15) and (16), ω represents the weight and α is the regularization parameter. L1 helps to minimize and even eliminate the features that have little effect on the training model and is generally not affected by outliers.

The key difference between L1 and L2 techniques, apart from the penalty term, is that L1 shrinks the less important feature's coefficient to zero thus, removing some feature altogether. So, this works well for feature selection in case we have a huge number of features. L2 loss function will try to adjust the model according to the outlier values, it reduces overfitting by allowing some samples to be misclassified [*17*], but in general L2 is more popular in machine learning than L1.



-a- -b-

Figure 10: Accuracy and loss for AST-MLP

The graph in Figure 10 is the results of running MLP learning algorithm. As shown in the graph, the accuracy and loss were 90.11% and 0.314 respectively.

The LSTM result in Figure 9 shows that, the learning algorithm achieved the accuracy of 90% around the 5th epoch. On the other hand, for the MLP result in Figure 10, the 90% accuracy was attained around the 19th epoch. This observation suggests that, although MLP and LSTM both achieved comparable accuracy of above 90%, LSTM is a superior learning algorithm to MLP on code completion task, because it learns much faster than the MLP algorithm.

6. MODEL PERFORMANCE

We validated our work by comparing our results to similar studies conducted on code completion. Programming languages such as java, C++, etc. have been used to train code completion models, however, our work focused on python. Therefore, our result was compared with the state-of-the-art study on python code generation or completion. We summarized the findings from various researchers on this topic in Table 1. In this table, we see that our model has a sharp improvement over the pointer mixture network by *Li et al.* [*13*].

Table 1: Comparison against state of the arts studies:

Study	Result (Accuracy)
NLP	**53.4%**
Vanilla LSTM [*13*]	67.3%
Attention LSTM (no parent attention)	69.8%
Attention LSTM [*13*]	69.8%
Pointer Mixture Network [*13*]	70.1%
Probabilistic Model [*18*]	69.2%
Deep Learning model with AST-LSTM (ours)	**90.3%**
Deep Learning model with AST-MLP (ours)	**90.1%**

As shown in Table above, using Vanilla LSTM, Li et.al., achieved an accuracy of 67.3% [*13*] . The result was obtained from a standard LSTM network. In the same study, they ran another experiment using Attentional LSTM. The model was based on an LSTM which could remember the last 50 hidden states per time step. However, pointer mixture network was found to have the best performance with 70.1% accuracy. Compared with our work, the table shows a considerable improvement of 20.2 percentage points over the pointer mixture network.

7. CONCLUSION

In this study we:

1. Designed, developed and evaluated an AST-LSTM/MLP code completion model.
2. Tokenized 1274 python source codes
3. Achieved a 69.5% and 29% increase over NLP and Pointer Mixture Network methodologies respectively
4. Obtained a 90.3 % and 90.1% for LSTM and MLP accuracy respectively.

The implications of the study are the following:

I. The conventional natural language processing (NLP) methodology has a limited scope in source code pattern recognition or knowledge discovery.

II. LSTM and MLP learning algorithms have the capability of code completing or generation with high accuracy.

III. AST-LSTM is an efficient mechanism in python code completion or generation. Although MLP and LSTM both achieved comparable accuracy of above 90%, LSTM is a superior learning algorithm to MLP on code completion task, because it learns much faster than the MLP algorithm.

The application of our work includes; source code generation, plagiarism detection, copyright investigation, software forensics and malware detection.

In further studies we plan to investigate de-anonymizing source code authorship, develop a heuristic to quantify a developer's contributions to an open source project, and study the evolution of a programmer's coding style over a time period.

8. References

[1] Zhaopeng Tu, Zhendong Su, and Devanbu Premkumar, "On the Locallness of software.," in *Proceddings of the International Symposium on Foundations of Software Engineering*, 2014, pp. 261-280.

[2] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu, "On The naturalness Of software," in *Procedings of the International Conference of Software Engineering*, 2012, pp. 837-847.

[3] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshuvanyk, "Toward deep learning software repositories," in *Procedings of the Working Conference on Mining Software Repositories*, 2015, pp. 334-345.

[4] A Hindle, E T Barr, Z Su, M Gabel, and Devanbu P, "On the Naturalness of software," in *In Proceedings of the 34th International Conference On Software Engineering, ICSE*, 2012, pp. 837-847.

[5] T T Nguyen, A T Nguyen, H A Nguyen, and T N Nguyen, "A statistical Semantic Language model foir source code.," in *In proceedings of the 2013 9th joint meeting on foundation of software Engineering*, New York, 2012, pp. 532-542.

[6] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou, "Context-Sensitive Code Completion Tool for Better API Usability," in *IEEE*

International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 621-624.

[7] Pascal Roos, "Fast and Precise Statistical Code Completion," in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, 2015, pp. 757-759.

[8] Sangmok Han, David R. Wallace, and Robert C. Miller, "Code Completion from Abbreviated Input," in *IEEE/ACM International Conference on Automated Software Engineering.*, Auckland, 2009, pp. 332-343.

[9] Takayuki Omori, Hiroaki Kuwabara, and Katsuhisa Maruyama, "A study on repetitiveness of code completion operations," in *28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, 2012, pp. 584-587.

[10] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers, "Active code completion," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Pittsburg, 2011, pp. 261-262.

[11] James Cross and Liang Huang, "Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles," in *Conference on Empirical Methods in natural language Processing, EMNLP 2016*, Austin, TX, 2016, pp. 1-11.

[12] Chris Dyer, Adhiguna Kuncoro, Miguel Ballerestos, and Noah A. Smith, "Recurrent Neural Network grammars," in *North American Chapter Of The Association for Computational Linguistics: Human Language Technologies*, San Diego, 2016, pp. 199-209.

[13] Jian Li, Wang Wang, Michael R. Lyu, and Irwin King, "Code Completion with Neural Attention and pointer Networks," in *The Twenty-Seventh International Joint Conference On Artificial Intelligence (IJCAI-18)*, 2018.

[14] Adam Ginzberg, Lindsey Kostas, and Tara Balakrishnan, "Automatic Code Completion," in *Stanford, class project 2017*, 2017.

[15] Sepp Hochreiter and Jurgen Schmidhuber, "Bridging lont time lags by weight guessing and "long short-term memory"," in *Spatiotemporal Models in Biological and Artificial Systems*, vol. 37, 1996, pp. 65-72.

[16] Shai Shalev-Shwartz and Shai Ben-David, *Understanding Machine Learning: From Theory to Algorithm*. New York, USA: Cambridge University Press, 2014.

[17] Tanay. Thomas and Lewis Griffin D. (2018, June) A new Angle on L2 Regulatrization. arXiv preprint. [Online]. https://arxiv.org/pdf/1806.11186v1.pdf

[18] Veselin Raychev, Pavol Bielik, and Martin Vechev, "Probabilistic model for code with decision tree," in *Proceedings of the International Conference Of Object Oriented Programming*, 2016, pp. 731-747.