Journal of Data Acquisition and Processing
ISSN: 1004-9037
https://sjcjycl.cn/
DOI: 10.5281/zenodo.777638

CROSS LANGUAGE, ADVANCE LSTM FOR SOFTWARE DEFECT PREDICTION

Yashwant Kumar [a], Dr. Vinay Singh [b]
[a]Department of computing and Information Technology, Usha Martin University, Ranchi, Jharkhand
[b]Associate Professor, Department of computing and Information T echnology, Usha Martin University, Ranchi, Jharkhand

Abstract
The current scenario of Software Development evolves from single programming language development framework (vis Java, C#, Php, Python, Java Scripts etc) to multi language micro services based architecture. For Software Products there are plethora of low level and high level programming languages. The challenge is to create a single software defect predictor model which is common to language as well as project. The Idea of conceptualizing this work emphasized that, Similar to natural language, every programming language also have linguistics characteristics like syntax, semantics, pragmatics and grammars. The technique of Natural Language Processing (NLP) is one of the oldest areas of machine learning research and is employed in significant fields such as machine translation, speech recognition, sentiment analysis, and various other text processing (Kumar & Singh, 2020). In this paper we have leveraged the concept of NLP in (Deep Learning Network) DLN to convert the Source code into sequence of lexer and parser classes based on the defined grammar, fixed length feature vectors classes are passed into embedded layer of Advance Long short-term memory (A-LSTM) Network. This Network can learn the linguistic pattern in source code, then it is used to predict the defective modules in the projects regardless of programming language. The results outperformed as compare to hand crafted software matrices based DLN in identification of buggy modules in software projects.

Key Words: Software Defect, Natural Language, LSTM, Software Metrics, Software Quality

1. Introduction

In practice Software Quality Assurance and Software Testing is considered as the last leg of Software Development Life cycle. This involved Software Quality Team to use lot of resources and time consuming processes, this may lead to missing targeted product launch deadline this results in lots of re-work by software developers. Sometime it becomes very cumbersome task and due to tight deadline of product launch, it may lead to launch of inferior version of software and to face product failure. Early detection of fault in software during combined practices of continuous integration and continuous delivery (CI/CD) pipeline helps the team to minimizes the cost of testing and improves the effectiveness of software development process.

This approach is articulated on novel Deep learning LSTM-Bidirectional-Attention (A-LSTM) based model. This extracted the language neutral tokens from source code using ANother Tool for Language Recognition (ANTLR) (*ANTLR*, n.d.). It uses Language specific Lexer and Parser Grammar to generate the tokens. These sequence of tokens are preprocessed and normalized by unsupervised Learning Techniques to generate fixed length Vectors (Pornprasit & Tantithamthavorn, 2021). These Vector data is input to this Deep Learning Based Advance

LSTM (A-LSTM) Network to train. Its efficiency of classification is evaluated using metrics accuracy, precision, recall, F1-score and Area Under the Curve (AUC) (Singh, 2019).

2. Related Work

In Most of Research work, Machine Learning (ML) Techniques, (H. Wang et al., 2021) are widely used for Software Defect Prediction. The traditional ML Techniques is based on Manually Extracted Feature from source codes like MOOD, CK, Halstead, McCabe Metrics (Kumar & Singh, 2021). Which requires lots of historical data of previous version of the same product, to predict the defect in future product. However, in practice newly created software product lacks historical data and has either little or no training data of itself to construct a model; directly using static code metrics based bug data sets from other projects to construct models could not achieve satisfactory prediction performance in most cases, since the metrics (features) distribution between different projects is not the same (H. Li et al., 2019).

The traditional software metrics, which are hand crafted and designed manually based on the analysis of code complexity or process, are not able to extract such complicated information from source code. Some researchers resort to extract semantic features from source code like abstract syntax tree (AST) which have smaller feature distribution difference among different programming language and projects (Fan et al., 2018). This type of approach proves to be more efficient then transfer learning methods for cross project defect prediction approach.

For example, (S. Wang et al., 2016) deployed DBN (deep belief network) (Hinton et al., 2006) to learn semantic features for SDP, based on token sequences extracted from source code. (J. Li et al., 2017) further applied CNN (convolution neural network) (Goodfellow et al., 2016) to learn semantic features from token sequences and build end-to-end WPDP models.

3. Methodology

In analyzing the defects of software prediction, the application of NLP technique is used to play an important role. One of the applications of natural language processing that has been used here is the automatic summarizing of text using software.

Similar to natural or formal language, the programming language also have linguistics characteristics like syntax, semantics, pragmatics and grammars. Like in the formal language like English, the Language Recognition Tools like ANTLR can be applied in source code to convert it into sequence of language neutral tokens. Like below:

packageDeclaration, annotation, importDeclaration, typeDeclaration, classOrInterfaceModifier, classDeclaration, enumDeclaration, interfaceDeclaration, annotationTypeDeclaration, modifier, classOrInterfaceModifier, variableModifier, typeParameters, typeType, typeList, classBody, typeBound, enumConstants, enumBodyDeclarations, interfaceBody, memberDeclaration, methodDeclaration, methodBody, typeTypeOrVoid, genericMethodDeclaration, genericConstructorDeclaration, constructorDeclaration, fieldDeclaration, constDeclaration etc

The characteristics of the process are identified to examine the functions of the software prediction. It influences the operations on the practices of social strategy in the field of human resources context.

3.1 Lexer and Parser Classes

Lexer and parser classes are fundamental components of a compiler or parser generator like ANTLR. They are responsible for analyzing the input source code and breaking it down into a

more structured representation that can be easily processed by the compiler or other language-based applications.

3.1.1 Lexer (also known as a tokenizer or scanner)

The lexer is the first phase of the compilation process. Its primary function is to read the input source code character by character and convert it into a stream of tokens. Tokens are small units of the source code that represent the language's basic elements, such as keywords, identifiers, literals, operators, and punctuation.

For example, in a programming language, the lexer would recognize and output tokens like "if," "else," "while" for keywords, variable names as identifiers, numeric literals, arithmetic operators, etc.

The lexer's output is passed to the parser for further processing. It serves as a crucial first step in the compilation process by simplifying the source code into a sequence of tokens that can be more easily handled by the subsequent phases.

3.1.2 Parser

The parser takes the token stream generated by the lexer and constructs a hierarchical representation of the source code's syntactic structure. This hierarchical representation is typically represented as an Abstract Syntax Tree (AST) or a Parse Tree.

The parser analyzes the grammar rules of the language to recognize the valid combinations of tokens that make up expressions, statements, and other language constructs. It ensures that the input source code conforms to the language's syntax rules.

For instance, if the source code contains the expression "a = b + c", the parser will construct a tree representation showing that this is an assignment statement where the variable "a" is assigned the sum of "b" and "c".

The AST or parse tree generated by the parser becomes the foundation for subsequent compilation phases like semantic analysis, optimization, and code generation.

Lexer and parser classes are automatically generated by parser generators like ANTLR based on the grammar specifications provided by the developer. These classes implement the logic for recognizing tokens (in the case of the lexer) and building the parse tree or AST (in the case of the parser). By using these generated classes, developers can save significant effort and avoid writing the low-level parsing code from scratch.

3.2 Tool for Language Recognition

ANTLR can be utilized to build a source code-based software defect predictor by creating a custom parser that understands the structure of the source code. This parser can extract relevant features from the source code, which are then used as input to a machine learning model for predicting software defects.

Here's a general outline of how ANTLR has been used for this purpose:

3.2.1. Define the Grammar:

This is a onetime step of defining a grammar for the programming language that has to be analyzed. ANTLR allows to create a grammar specification that describes the syntax and structure of the source code in that language.

3.2. 2. Generate the Parser:

Used ANTLR to generate lexer and parser classes based on the defined grammar. These classes will be able to read and process the source code files and create an Abstract Syntax Tree (AST) representation.

3.2. 3. Traverse the AST:

Written custom code to traverse the AST generated by ANTLR. During the traversal, relevant features were extracted from the source code that can be used for defect prediction. These features might include metrics related to code complexity, size, nesting levels, and usage of certain programming constructs etc.

3.2.4 Data Collection:

Used the custom traversal code to extract features from a set of source code files that are labeled with defect information (defective or defect-free). Stored these features along with the corresponding labels in dataset files for building the training and testing datasets.

ANTLR simplifies the process of creating the custom parser for the programming language, allowing to focus on the specific aspects of defect prediction. By combining ANTLR's parsing capabilities with machine learning, effective software defect predictor has been created that can aid in identifying potential issues in the source code early in CI/CD pipeline.

3.3 Dataset Collection and Pre-processing for a Cross-Language Software Defect Predictor

The dataset used for training and evaluation in the Cross-Language Software Defect Predictor project was collected from various open-source repositories on platforms like GitHub, PROMISE-backup-master, Bitbucket, or GitLab. These repositories contain code written in different programming languages, including but not limited to Java, C++, Python, JavaScript, and C#. The dataset was curated to include projects with labeled information about defective and defect-free code examples.

3.3.1 Dataset Pre-processing:

The raw dataset obtained from the repositories underwent several pre-processing steps to prepare it for training and evaluation. The pre-processing steps include:

3.3.1.1 Language Identification:

Since the repositories contain code from multiple programming languages, an initial language identification step was performed to categorize each file into its respective programming language using language-specific heuristics or machine learning classifiers.

3.3.1.2 Lexical Analysis

The code files were tokenized using the language-specific lexer generated by ANTLR for each supported programming language. This process involved converting the source code into a stream of language-specific tokens, such as keywords, identifiers, literals, and operators.

3.3.1.3 AST Generation:

The token streams were then passed through the corresponding language-specific parser generated by ANTLR to construct Abstract Syntax Trees (ASTs) or parse trees representing the hierarchical structure of the code.

3.3.1.4 Feature Extraction:

From the generated ASTs, various language-independent features patterns were extracted automatically. These features may have included code complexity metrics (e.g., cyclomatic complexity), code size, control flow information, variable usage patterns, function call patterns, and many more. The goal was to create a uniform set of features that could be used for defect prediction across different programming languages.

3.3.1.5 Labeling:

The code files were labeled with information about whether they contain defects or are defect-free. The labels were obtained by analyzing issue tracking systems, bug databases, and code review comments associated with each code file in the repositories.

3.3.1.6 Balance the Dataset:

Ensured the dataset has a balanced representation of defective and defect-free instances for each programming language. Based on size of buggy or clean data, SMOTE and Resampling techniques were applied. This step was crucial to avoid biases in the model's predictions.

3.4 Model Architecture for software defect prediction

The Long Short-Term Memory (LSTM) architecture is a type of recurrent neural network (RNN) that has proven to be effective in sequence modeling tasks, including natural language processing and time series analysis. The traditional RNNs can have difficulty learning and maintaining dependencies that span long time steps, which is a common issue in tasks involving sequences like natural language processing or time series analysis. LSTM was introduced to overcome this limitation and has become widely used in various applications. In the context of software defect prediction, LSTM can be applied to analyze the sequence of code tokens or features extracted from the source code and predict the presence of defects.

The key idea behind LSTM is the incorporation of special memory cells, known as LSTM cells, which allow the network to selectively store and access information over long periods of time. These cells have three main components: an input gate, a forget gate, and an output gate. These gates regulate the flow of information inside the cell, enabling it to retain relevant information and discard unnecessary information over multiple time steps.

3.4.1 LSTM Cell:

Here's a detailed explanation of the components of an LSTM cell:

3.4.1.1 Cell State (Ct):

The cell state, denoted as Ct, is the memory component of the LSTM. It runs through the entire sequence and can pass information across time steps, allowing the network to maintain dependencies over long distances. It acts as a conveyor belt that carries information, updated through the forget gate, input gate, and output gate.

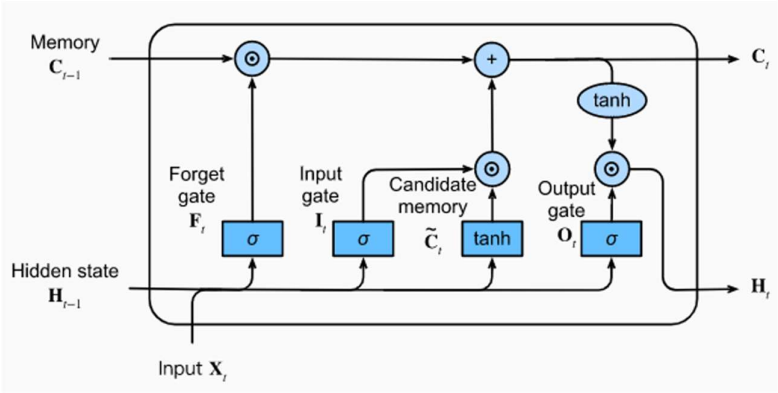

Figure: Representation of Single Forward LSTM Single Cell Architecture

3.4.1.2. Hidden State (ht):

The hidden state, denoted as ht, represents the output of the LSTM cell at a given time step t. It is a function of the cell state and the input at that time step. The hidden state captures relevant information that the LSTM has learned from the input sequence up to that point.

3.4.1. 3. Input Gate (it):

The input gate, denoted as it, determines how much of the new information should be added to the cell state. It takes the current input and the previous hidden state as inputs and outputs a value between 0 and 1 for each element of the cell state. This value controls which elements of the cell state should be updated.

3.4.1.4 Forget Gate (ft):

The forget gate, denoted as ft, decides which information should be discarded from the cell state. It takes the current input and the previous hidden state as inputs and outputs a value between 0 and 1 for each element of the cell state. This value determines how much of the previous cell state should be retained.

3.4.1.5 Output Gate (ot):

The output gate, denoted as ot, controls how much of the cell state should be exposed as the output of the LSTM cell. It takes the current input and the previous hidden state as inputs and outputs a value between 0 and 1 for each element of the cell state. This value determines which elements of the cell state should contribute to the hidden state output.

3.4.2 Number of Layers:

The number of layers in the LSTM architecture is a hyper parameter that defines how many LSTM cell layers are stacked on top of each other. Common choices for the number of layers range from 1 to 3. Deeper architectures with more layers can capture more complex patterns, but they also require more computational resources and may be prone to overfitting if not properly regularized.

3.4.3 Hidden Units:

The number of hidden units in an LSTM cell determines the dimensionality of the LSTM's internal representation. Higher numbers of hidden units can capture more intricate patterns in the data but can also increase the model's computational complexity.

3.4.4 Activation Functions:

Typically, the LSTM uses a combination of activation functions to control the information flow and transformations within the cell. The activation functions commonly used in LSTM are:

3.4.4.1. Sigmoid Activation: Used for the input gate, forget gate, and output gate to control the flow of information.

3.4.4.2. Hyperbolic Tangent (tanh) Activation: Used to compute the candidate value that could be added to the cell state.

3.4.5 Other Relevant Hyper parameters:

Apart from the architecture-related hyper parameters, there are several other hyper parameters that need to be tuned during the training process of an LSTM model for software defect prediction. These include:

3.4.5.1. Learning Rate: The step size that determines how much the model weights are updated during backpropagation.

3.4.5. 2. Batch Size: The number of samples used in each update step during training.

3.4.5.3 Dropout Rate: A regularization technique to prevent overfitting by randomly setting a fraction of the LSTM units' outputs to zero during training.

3.4.5.4. Optimization Algorithm: The choice of optimization algorithm used to update the model weights, such as Adam, RMSprop, or stochastic gradient descent (SGD).

3.4.5.5. Sequence Length: The length of input sequences provided to the LSTM model during training and prediction.

3.4.6. Advance LSTM Layers

3.4.6.1 Bidirectional LSTM Layer:

Bidirectional LSTMs process the input sequence both forward and backward, allowing the model to capture dependencies in both directions

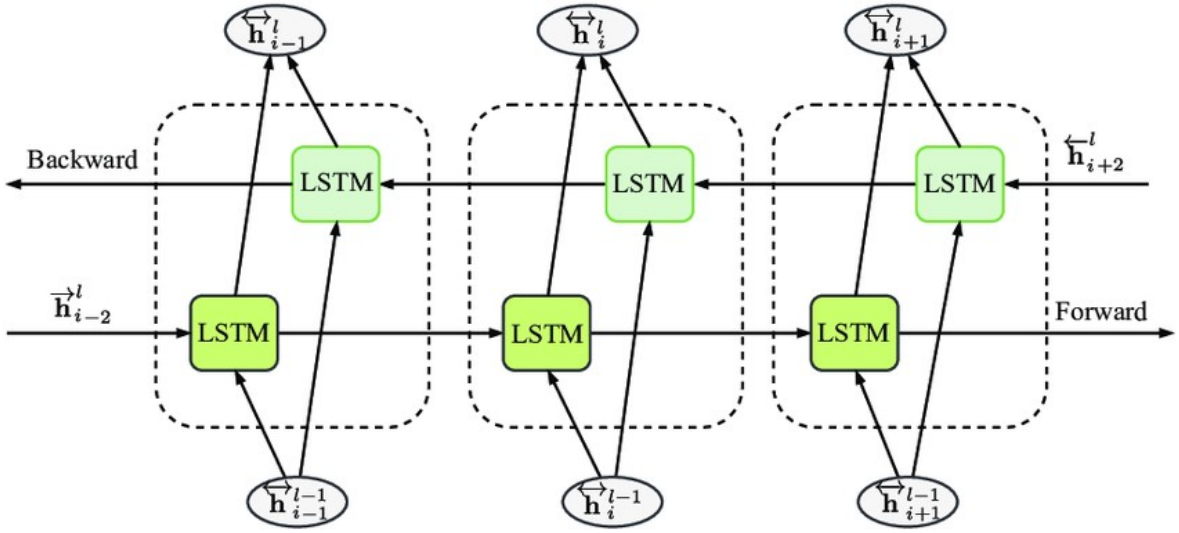

Figure: Representation of Bi-LSTM

3.4.6.2 Attention Layer:

The attention layer applies the Attention mechanism to the output sequences of the LSTM layer. Attention mechanisms allow the model to focus on different parts of the input sequence during the decision-making process. The Attention layer helps the model assign different weights to different time steps (words) in the input sequence based on their importance for the task.

3.4.6.3 Dense Output Layer:

The output layer is a Dense layer with one unit and a sigmoid activation function. This layer is responsible for making the final binary classification prediction (e.g., defect presence or absence). The sigmoid activation function squashes the output between 0 and 1, making it suitable for binary classification problems.

This model architecture is suitable for text classification tasks where the input is a sequence of words, and the goal is to predict a binary outcome (e.g., defect prediction). The use of Bidirectional LSTM with Attention allows the model to effectively learn from the sequence data and capture important patterns for the classification task.

3.5 Rationale of Choosing LSTM

Elaboration and Rationale for Choosing Attention-Based Bidirectional LSTM over other Prediction Models in Software Defect Prediction

3.5.1. Handling Long-Term Dependencies:

The primary reason for choosing an LSTM-based model, particularly the Bidirectional LSTM, is its ability to handle long-term dependencies in sequential data. In software defect prediction, the presence of defects often depends on patterns that span multiple code lines or functions. Bidirectional LSTM enables the model to capture context from both past and future time steps, making it suitable for understanding complex relationships in source code.

3.5.2. Attention Mechanism:

The addition of an attention mechanism to the Bidirectional LSTM further enhances the model's performance. Attention mechanisms allow the model to focus on more relevant parts of the input sequence, giving higher importance to specific time steps and effectively ignoring noise or less important elements. This adaptability makes the model more robust to variations in code structure and improves its ability to detect defect-related patterns.

3.5.3 Language Independence:

Software defect prediction often deals with code written in various programming languages. The attention-based Bidirectional LSTM, with its token-level processing, can be language-independent. By feeding the model with language-agnostic features extracted from code tokens, it can be applied to projects written in multiple programming languages without significant modification.

3.5.4 Bidirectional Processing:

In many defect prediction scenarios, the context of a code token can depend on both preceding and succeeding tokens. Bidirectional LSTM can capture dependencies in both directions, allowing the model to leverage information from the past and future in its decision-making process. This bidirectional processing is advantageous for detecting defect patterns that rely on context from both directions.

3.5. 5. Sequence-to-Sequence Prediction:

Attention-based Bidirectional LSTM can perform sequence-to-sequence prediction. In the context of software defect prediction, this means the model can take a variable-length sequence of code tokens as input and generate a corresponding sequence of predictions, indicating the presence or absence of defects at each code token. This fine-grained prediction capability allows the model to identify specific locations of potential defects in the source code.

3.5. 6. Generalization:

The attention-based Bidirectional LSTM model has the potential to generalize well to unseen projects and codebases. The attention mechanism enables the model to focus on relevant features and patterns, making it adaptable to new projects with varying code structures and styles.

3.5. 7. Previous Success in NLP Tasks:

Attention-based LSTMs have demonstrated remarkable success in natural language processing (NLP) tasks, such as machine translation and sentiment analysis. Given the similarity between code tokens and natural language words, the success of attention-based LSTMs in NLP tasks provides a strong rationale for their application in software defect prediction.

3.5. 8. Better Feature Extraction:

The attention mechanism helps the model to selectively attend to important features during the prediction process. This capability enhances the model's ability to extract salient information from the input sequences and focus on the most relevant parts of the code to make accurate predictions.

In summary, the rationale for choosing an attention-based Bidirectional LSTM for software defect prediction lies in its capability to handle long-term dependencies, language independence, bidirectional processing, sequence-to-sequence prediction, generalization ability, and the success of attention-based models in NLP tasks. This model offers a powerful approach for accurately detecting defects in source code, allowing for improved software quality and maintenance in diverse programming projects.

4. Experimental Setup

This approach is divided into various steps like Data Pre-processing, address class imbalance issues, LSTM model building, training, testing and validating the model. and finally analysis of results.

4.1 Data Pre Processing

4.1.1 Merge the Bug Data Set with Source Code Tokens

Merged the File wise source code tokens and Corresponding Bug Flag from Bug Data Set to Create Final Data Set for Training and Test Purpose. The labeled data is gathered from well-known GitHub Bug Data Set. which consist of Multiple language and versions, Package wise and file Wise Bug Data with Static Code Metrics. Then gathered the Source Code of corresponding Versions from same GitHub like for example Promise Backups (*PROMISE-Backup/Bug-Data at Master · Feiwww/PROMISE-Backup*, n.d.). The Bug Data and Source code of following Products and versions were analyzed.

- ant: 1.3, 1.4, 1.5, 1.6.0, 1.7.0
- camel: 1.0, 1.2, 1.4, 1.6
- ivy: 1.0, 1.1, 1.2
- jedit: 3.2, 4.0, 4.1, 4.2, 4.3
- log4j: 1.0, 1.1, 1.2
- lucene: 2.0, 2.2, 2.4
- poi: 1.5, 2.0, 2.5, 3.0
- synapse: 1.0, 1.1, 1.2
- velocity: 1.4, 1.5, 1.6
- xalan: 2.4, 2.5, 2.6, 2.7
- xerces: 1.1, 1.2, 1.3, 1.4.4

4.1.2 Final Structure of the pre-processed data:

The two data sets (source code token datasets created as mention in section 3 and corresponding labeled datasets) are joined using custom code and created the final data set, which columns are **Project Name** and version like Jakarta15, Module Name of the Source Code file with complete package identifier, Source code converted into language independent tokens and the bug status, here 0 means no bug and 1 means source code have bug as shown in table 4.

Project	Name	Language Independent Source code Tokens	bug
jakarta15	org.apache.tools.ant.taskdefs.ExecuteOn	... .qualifiedName,block,typeDeclaration,	0
….	….	….	…

Table 1: Tokens and Source Code Mapping merged with Bug Status for Training the model.

4.1.3. Removal of Class Imbalance:

In the next phase the system removes unwanted tokens and separator symbols from Tokens to neutralize the tokens. The ratio of bug to non-bug showed the class imbalance problems. The class imbalance problems may lead to biased output. So to remove class imbalance issue Random Sampling Techniques or SMOTE is used.

4.1.4. Convert Source Code into Language Independent Tokens vector

System represented the tokens as sequence of words of fixed vocabulary size of 2000 and split it. Then it converted the text to sequence using Tokenizer. Finally, these tokens are converted into two dimensional Vectors. For Example, it looks like below.

```
array([[  0,   0,   0, ...,   1,   1, 279],
       [  0,   0,   0, ...,   1,   1,  42],
       [  0,   0,   0, ...,   0,   2,  16],
       ...,
       [  0,   0,   0, ..., 278,  12,  91],
       [  0,   0,   0, ...,   1,   1, 1272],
       [  0,   0,   0, ..., 278,  12,  92]])
```

4.1.5 Convert feature vector into Dense Feature Vector

The most basic pre-processing in LSTM model is to convert words into an embedding Vectors using Word2Vec or GloVe. Since here Word2Vec or GloVe model is not used, this model has added its own embedding's layers by defining input dimension that means the tokenizer will consider only the top most frequent unique words in the dataset, output dimension means the dimension of the dense word embeddings and sequence length to which all input sequences will be padded or truncated. It ensures that all sequences have the same length for feeding into the LSTM model.

Figure-1 Represents the bird eye view of data flow in A-LSTM Framework.

As discussed in Section 3, the detail approach is depicted in **Figure 2**.

This approach used ANTLR Lexer and Parser Grammar Files to extract tokens from Source code.

The Source code is converted into token using ANTLR Tool.

In the real experimental setup, the above process is applied to all the source code files taken from *Bug Repository.*

The above feature vector is put in A-LSTM layer of model for training and testing the model.

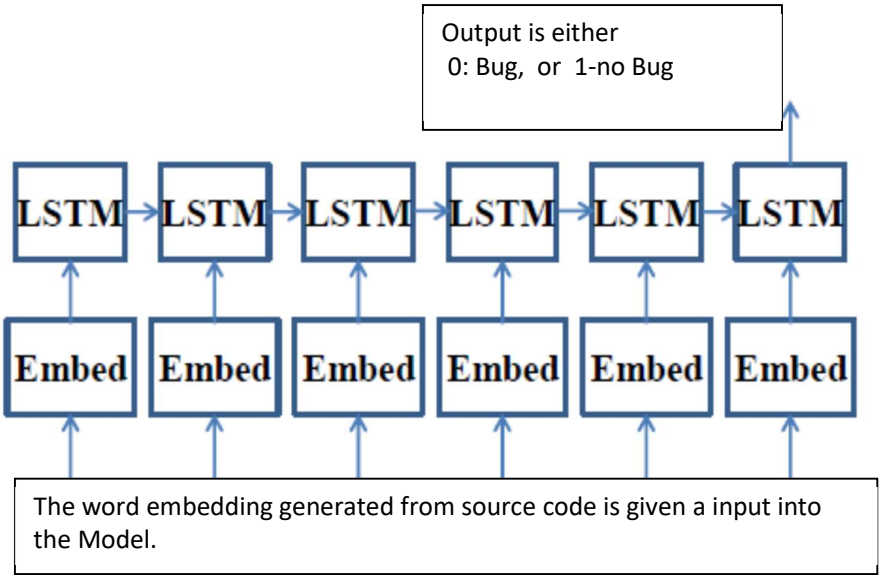

Figure 2: A-LSTM based approach.

4.2. hardware and software environment used for training and testing the LSTM model.

The model depicted in Figure -2 is created using the open source library Tensor Flow, Keras and using Python programming language. The brief final structure of DL model with various layer details are described in Table-4 below:

Model Name	Attention-Bi LSTM	**Sequential**
Sl No.	**Layer (type)**	
1.	Embedding	256000
2.	spatial dropout 1D	0
3.	Attention	328
4.	Bidirectional LSTM	509600
5.	Dense	394
Total params	766,322	
Trainable params	766,322	
Non-trainable params	0	

Table 4: A-LSTM Model Summary.

The Final Dataset is Split into Training and Test Set in the ratio of 85:15, feed into A-LSTM Network. The Network is trained initially with training data and epoch of 3 and batch size of 32. With transfer learning principles model is re-trained with new data set from other languages. Transfer learning involves taking a pre-trained model on one task and fine-tuning it

on another related task. By doing this, model can leverage the knowledge and representations learned by the pre-trained model, which helps in improving the performance on the new task. Initially trained the base model with a small amount of data to reduce training time. This step is often referred to as pre-training. Then saved the pre-trained model and continued training it with new data to further refine its representations for a specific task. This is known as fine-tuning or transfer learning.

By repeating this process iteratively improve the performance of the model as it was trained with more and more data. Keep in mind that the performance gains may not be as significant with each iteration, but it can still be an effective way to make use of limited resources (such as not having a CUDA-based machine) and achieve better results.

The Complete data flow is described in below figure-2.

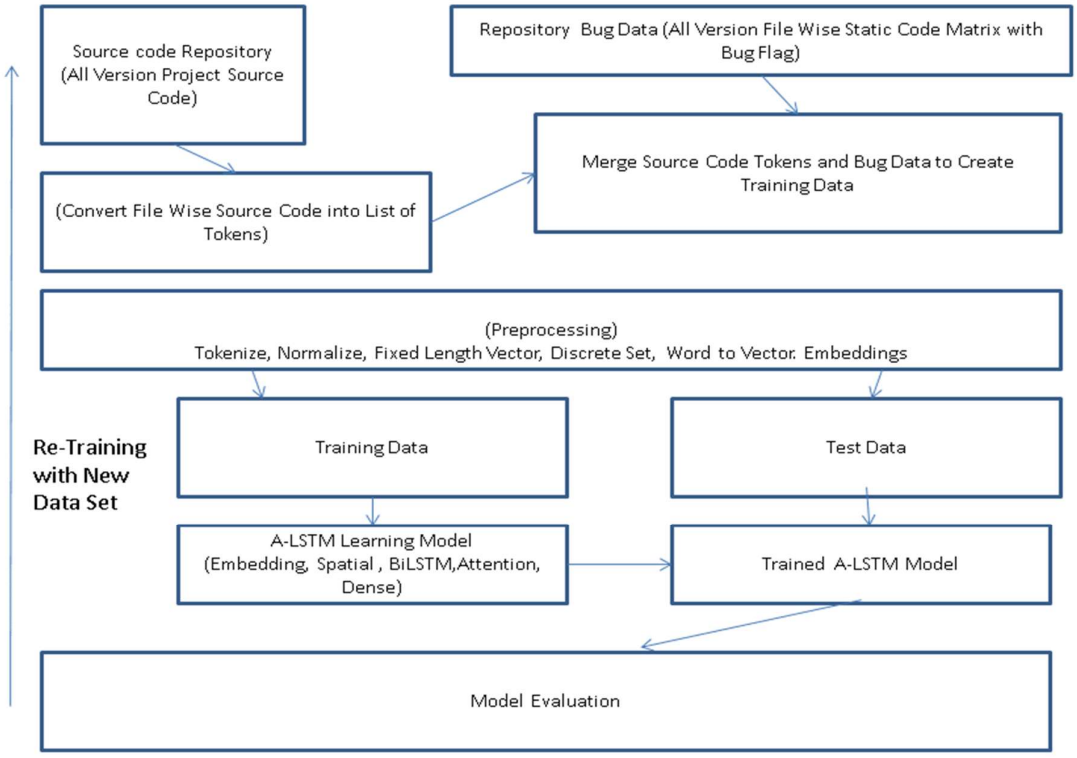

Figure 2: Complete Framework of A-LSTM based approach.

5. The Result of Experiment.

The Model is compiled with specifying the loss function, optimizer, and evaluation metrics to be used during training. The parameters are as follows:

1. The loss function used for binary classification problems, which is binary cross-entropy.
2. Optimizer: The optimization algorithm used during training, which is the Adam optimizer.
3. Evaluation Metrics: A list of metrics to evaluate the model's performance during evaluation. In this case, it included accuracy, F1 Score and the Area Under the ROC Curve (AUC) as evaluation metrics.

The results of three type of LSTM Model is given in Table-5 below.

Model Type	Loss Score	F1 Score	Area Under the Curve (AUC)
Single Forward LSTM	4.02%	.60	.72
Bi-LSTM	3.90	.81	.76
A-LSTM	.028	.90	.99

Here we can see that our accuracy and losses of the model in the data has changed drastically where we are receiving the accuracy around 76% for 12 epochs using a Bi-LSTM model. After using the attention in the model we increased the accuracy to 99% and also the loss has decreased to 0.028.

Table 5: Evaluation Results of A-LSTM Model.

6. Discussion

Building a cross-language defect predictor can be more challenging than language-specific ones due to differences in language syntax, semantics, and programming paradigms. Some challenges are:

6.1 Feature extraction:

Designing language-independent features that capture defects across various languages effectively.

6.2 Imbalanced datasets:

Ensuring a balanced representation of defects and defect-free examples for each language in the cross-language dataset.

6.3 Language-specific quirks:

Some languages may have specific constructs or patterns that are unique to them, making it harder to capture defects universally.

6.3 Language coverage:

Supporting a wide variety of programming languages might require significant effort to implement their lexers and parsers.

Despite these challenges, building a cross-language defect predictor was highly valuable, as it allowed to apply defect prediction techniques consistently across projects written in different programming languages, promoting code quality and maintenance across diverse codebases.

5. Conclusion and Future Work

In this paper, we propose a novel language independent approach via Embedding and A-LSTM-based neural network. Final empirical results on real open source projects demonstrate the effectiveness of our proposed approach. In the future, we want to extend our research in several ways. First we want to investigate the generalization of our empirical studies by considering datasets from more open source projects and commercial projects. Second we want to consider more reasonable input for our approach, combining program analysis techniques. Then we want to consider or propose more Advance DLN models and embedding methods to

improve the performance of our approach. Finally, we want to apply our approach to the real software quality assurance process of enterprises.

6. References

1. *ANTLR*. (n.d.). Retrieved December 25, 2021, from https://www.antlr.org/
2. Fan, Y., Cao, X., Xu, J., Xu, S., & Yang, H. (2018). High-Frequency Keywords to Predict Defects for Android Applications. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, *02*, 442–447. https://doi.org/10.1109/COMPSAC.2018.10273
3. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
4. Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*(7), 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527
5. Kumar, Y., & Singh, D. V. (2020). Cross Project and within Project Software Defection prediction using NLP Techniques. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, *11*(3), 842–849.
6. Kumar, Y., & Singh, D. V. (2021). A Practitioner Approach of Deep Learning Based Software Defect Predictor. *Annals of the Romanian Society for Cell Biology*, 18764–18785.
7. Li, H., Li, X., Chen, X., Xie, X., Mu, Y., & Feng, Z. (2019). Cross-project Defect Prediction via ASTToken2Vec and BLSTM-based Neural Network. *2019 International Joint Conference on Neural Networks (IJCNN)*, 1–8. https://doi.org/10.1109/IJCNN.2019.8852135
8. Li, J., He, P., Zhu, J., & Lyu, M. R. (2017). Software Defect Prediction via Convolutional Neural Network. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 318–328. https://doi.org/10.1109/QRS.2017.42
9. Pornprasit, C., & Tantithamthavorn, C. K. (2021). JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 369–379. https://doi.org/10.1109/MSR52588.2021.00049
10. *PROMISE-backup/bug-data at master · feiwww/PROMISE-backup*. (n.d.). GitHub. Retrieved December 25, 2021, from https://github.com/feiwww/PROMISE-backup
11. Singh, P. (2019). Learning from Software defect datasets. *2019 5th International Conference on Signal Processing, Computing and Control (ISPCC)*, 58–63. https://doi.org/10.1109/ISPCC48220.2019.8988366
12. Wang, H., Zhuang, W., & Zhang, X. (2021). Software Defect Prediction Based on Gated Hierarchical LSTMs. *IEEE Transactions on Reliability*, *70*(2), 711–727. https://doi.org/10.1109/TR.2020.3047396
13. Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. *Proceedings of the 38th International Conference on Software Engineering*, 297–308. https://doi.org/10.1145/2884781.2884804