AUTOMATED DIAGNOSIS AND CORRECTION OF LOGICAL ERRORS IN C PROGRAMS

A PROJECT REPORT

Submitted by

NAVINASHOK SWAMINATHAN 2019115126

PAARUSH SENTHILKUMAR 2019115063

AALIA KHIASUDEEN 2019115002

to the Faculty of

INFORMATION AND COMMUNICATION ENGINEERING

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY

COLLEGE OF ENGINEERING, GUINDY

ANNA UNIVERSITY

CHENNAI 600 025

MAY 2023

ANNA UNIVERSITY
CHENNAI - 600 025

BONAFIDE CERTIFICATE

Certified that this project report titled AUTOMATED DIAGNOSIS OF LOGICAL ERRORS IN C PROGRAMMING ASSIGNMENTS is the bona fide work of NAVINASHOK SWAMINATHAN (2019115126), AALIA KHIASUDEEN (2019115002), PAARUSH SENTHILKUMAR (2019115063) who carried out project work under my supervision. Certified further that to the best of my knowledge and belief, the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or an award was conferred on an earlier occasion on this or any other candidate.

PLACE:CHENNAI
DATE:

Dr. SELVI RAVINDRAN
ASSISTANT PROFESSOR
PROJECT GUIDE
DEPARTMENT OF IST, CEG
ANNA UNIVERSITY
CHENNAI 600025

COUNTERSIGNED

Dr. S.SRIDHAR
HEAD OF THE DEPARTMENT
DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY
COLLEGE OF ENGINEERING, GUINDY
ANNA UNIVERSITY
CHENNAI 600025

ABSTRACT

Programs are likely to contain logical errors.It is challenging for professors and students to find and fix logical errors in source code. Compilers and integrated development environments have the ability to detect and correct syntax errors but it is also difficult for them to detect and correct logic errors. The conventional way of detecting a logical error is an instructor assisting those errors to a large number of students which is extremely challenging. It is often easier to provide a correct implementation rather than search for logical errors in an error-ridden program. Generic feedback on logical errors when given to students is mostly ineffective. Providing the answer code is also troublesome as there are many ways to implement the program.

The aim of this project focuses on developing a system for fixing logical errors in a given source code. The model enables debugging of many logic errors in the source code through recurrent trials of error detection, correction, and source code testing. The model involves an iterative approach for detecting and correcting errors in the source code and testing it. It includes two submodels they are the Correct Code Model and the Editing Operation Predictor. The Correct Code Model generates a sequence of tokens for correct code by learning the structure of a set of correct codes, and compares the given source code to predict multiple correction candidates for logic errors. The Editing Operation Predictor indicates the appropriate editing operation for the correction candidate obtained by the Correct Code Model. Experimental results show that the correction accuracy is, on average, 51.63% higher than that of the conventional model without iterative trials.

ABSTRACT IN TAMIL

நிரல்களில் லாஜிக்கல் பிழைகள் இருக்க வாய்ப்புள்ளது. இது சவாலானது பேராசிரியர்கள் மற்றும் மாணவர்கள் மூலக் குறி யீட்டில் உள்ள தருக்கப் பிழைகளைக் கண்டறிந்து சரி செய்ய வேண்டும். தொகுத்தல் மற்றும் ஒருங்கிணைந்த வளர்ச்சி சூழ ல்கள் கண்டறியும் திறன் மற்றும் தொடரியல் பிழைகளை சரி செய்யவும் ஆனால் அவற்றைக் கண்டறிந்து சரிசெய்வது கடின ம் தர்க்கப் பிழைகள். தர்க்கரீதியான பிழைகள் பற்றிய கருத்துக் களை வழங்குவதற்கான வழி தோல்வியுற்ற சோதனை நிகழ் வுகள். வழங்குதல் செயல்படுத்த பல வழிகள் இருப்பதால்பதில் குறியீடும் பிரச்சனையாக உள்ளது திட்டம். இந்த திட்டத்தின் நோ க்கம் சரிசெய்வதற்கான அமைப்பை உருவாக்குவதில் கவனம் செலுத்துகிறது கொடுக்கப்பட்ட மூலக் குறியீட்டில் தருக்கப் பி ழைகள். மாடல் பலவற்றின் பிழைத்திருத்தத்தை செயல்படுத் துகிறது பிழை கண்டறிதலின் தொடர்ச்சியான சோதனைகள் மூலம் மூலக் குறியீட்டில் உள்ள தர்க்கப் பிழைகள், திருத்தம், மற் றும் மூல குறியீடு சோதனை. இது இரண்டு துணை மாதிரிக ளை உள்ளடக்கியது - சரியான குறியீடு மாதிரி மற்றும் எடிட்டிங் ஆபரேஷன் ப்ரெடிக்டர். சரியான குறியீடு மாதிரி ஒரு வரிசை யை உருவாக்குகிறது- சரியான குறியீடுகளின் தொகுப்பின் கட் டமைப்பைக் கற்றுக்கொள்வதன் மூலம் சரியான குறியீட்டிற்கா ன கென்ஸ், மற்றும் பல திருத்தம் வேட்பாளர்களைக் கணிக்க கொடுக்கப்பட்ட மூலக் குறியீட்டை ஒப்பிடுகிறது தர்க்க பிழைக ளுக்கு. எடிட்டிங் ஆபரேஷன் ப்ரெடிக்டர் பொருத்தமானதைக் கு றிக்கிறது - கரெக்ட் மூலம் பெறப்பட்ட திருத்த வேட்பாளருக்கு எடிட்டிங் ஆபரேஷன் சாப்பிட்டது குறியீடு மாதிரி

ACKNOWLEDGEMENT

First and foremost, we would like to express our sincere gratitude to Dr. SELVI RAVINDRAN, Assistant Professor, Department of Information Science and Technology, College of Engineering Guindy, Anna University, who served as our mentor and was a great source of guidance. She gave us the confidence and flexibility to successfully complete our final year project.

We would like to thank Dr. S. SRIDHAR, Professor and Head, Department of Information Science and Technology, College of Engineering Guindy, Anna University for his gracious assistance and provision of the tools needed to complete the project.

We are grateful to the project coordinator Dr. S. SWAMYNATHAN and project committee members Dr. J. INDUMATHI, Professor,
Dr. S. SENDHILKUMAR, Associate Professor, Dr. K. VIDYA, Assistant Professor, and Dr. D. NARASHIMAN, Teaching Fellow at College of Engineering Guindy, Anna University, Chennai for their reviews and valuable guidance throughout the course of our project.

We also thank the faculty members and non-teaching staff members of the Department of Information Science and Technology, College of Engineering Guindy Campus, Anna University, Chennai for their valuable support throughout the course of our project work.

NAVINASHOK SWAMINATHAN
PAARUSH SENTHILKUMAR
AALIA KHIASUDEEN

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AC	Accepted
BiLSTM	Bidirectional Long Short-Term Memory
C2AE	C Code AutoEncoder
CCM	Correct Code Model
CE	Compilation Error
Code2vec	Code to Vector
EOP	Edit Operation Prediction
IDE	Integrated Development Environment
LLVM	Low-Level Virtual Machine
LSTM-LM	Long Short-Term Memory Language Model
ML	Machine Learning
NPD	Nextword probability distribution
PIPE	Predicting Logical Programming Errors in Programming Exercises
SVM	Support Vector Machine

LIST OF VARIABLES

cur	Current word in sequence of tokens
nxt	Next word in sequence of tokens
nxt’	Predicted next word in sequence of tokens

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 PROJECT OVERVIEW

Programming is an exercise or practice that boost our logical thinking and improves a problem-solving skill. It teaches us how to accomplish a task with the help of a computer program or software. Therefore programming is a task to implement a solution to a problem in the form of computer language.

Errors are the problems or the faults that occur in the program, which makes the behaviour of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as debugging [1]. These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program. In the following section the different types of errors are described.

1.2 TYPES OF ERRORS

There are different kinds of errors that are encountered while compiling programs. Some of those errors are Syntax errors, Semantic errors and Logical errors.

1.2.1 Syntax Errors

Syntax errors are also known as the compilation errors as they occur at the compilation time [1]. These errors mainly occur due to the mistakes while typing or not following the syntax of the specified programming language. Syntax errors are mostly spelling and punctuation errors and incorrect labels. These errors exist in the program, and will cause the program to crash or not run at all. Syntax errors are caught by a software program called a compiler, and the programmer must fix them before the program is compiled and then run.

1.2.2 Semantic Errors

Semantic errors are the errors that occurred when the statements are not understandable by the compiler. The semantic error can arises using the wrong variable or using wrong operator or doing operation in wrong order. Some of the commonly seen semantic errors are incompatible types of operands, undeclared variable, not matching of actual argument with formal argument. They are encountered at run time [1] but would lead to exceptions that can be debugged.

1.2.3 Logical Errors

The logical error is an error that leads to an undesired output. These errors produce the incorrect output [1], but they are error-free.The occurrence of logical errors mainly depends upon the logical thinking of the developer. Logic errors are not always easy to recognize immediately because syntax errors, are valid when considered in the language, but do not produce the intended behavior. Logical errors in programs refer to mistakes in the design or implementation of the program logic, which may cause the program to behave incorrectly or produce incorrect results. They are not detected by compilers or IDE s and require manual identification and correction. Debugging logical errors can be challenging as they may require careful analysis of the program's code, execution, and data flow to identify and fix the problem.

1.3 TYPES OF LOGICAL ERRORS

A program can have different types of logical errors like control flow errors occur when the program's control flow does not follow the expected path due to incorrect or missing conditional statements or loops, calculation errors occur when the program produces incorrect results due to incorrect arithmetic or other mathematical calculations, input/output errors occur when the program fails to process input data correctly or produces incorrect output data, initialization errors occur when variables or objects are not initialized correctly, leading to unexpected behavior or results, timing errors occur when the program's timing or sequencing is incorrect, leading to race conditions, deadlocks, or other synchronization problems, memory errors occur when the program accesses or manipulates memory incorrectly, leading to crashes or other undefined behavior.

1.4 LOGICAL ERROR DETECTION USING ML

Diagnosis and correction of logical errors in C programs can be implemented using machine learning algorithms by analyzing the source code of a C program, identifying logical errors, and suggesting corrections to those errors. One approach in achieving this is to use supervised learning techniques. This involves training an ML model on a large dataset of C programs that have been annotated with correct and incorrect solutions. The model is then able to learn patterns in the data that allow it to distinguish between correct and incorrect solutions. Once the model has been trained, it can be used to analyze new C programs and identify logical errors. The model can also suggest corrections to those errors based on the patterns it has learned from the training data. Another approach is to use unsupervised learning techniques to analyze the code of a C program and identify patterns that are associated with logical errors. Clustering techniques are used to group similar pieces of code together, or anomaly detection techniques to identify pieces of code that do not match the expected patterns.

1.5 BACKGROUND

Providing non-personalised feedback on logical errors to the students was not helpful in rectifying these logical errors. The standard way of giving feedback on logical errors is failed test cases which may not help the students to fix these errors. Providing the answer code is also troublesome as there are many ways to implement the program.

1.6 OBJECTIVES

The goal of our project is to develop an automated system to find logical errors in C programs. The objectives of this project are :

- To develop a debugging model that iteratively detects and corrects errors and tests the source code.
- To develop CCM by LSTM-LM, which has learned the structure of a set of correct codes to predict the position of logical errors.
- To develop EOP that can predict the editing operation for the correction candidate by learning the editing operation performed between the incorrect code and the corresponding correct code.
- To train the proposed model with real solution codes oriented to programming tasks as correction candidates obtained from an online judge system.

1.7 PROBLEM STATEMENT

In programs logical errors highly occur identifying and rectifying them is a tedious process. The existing solutions are usually through test cases or manually identifying logical errors through a mentor. The aim of this project is to develop a model for debugging of multiple logical errors in the source code by iterative trials of identifying the errors, correcting the errors, and testing the source code.

1.8 SOLUTION OVERVIEW

A system is developed that identifies the position of the logical error in a particular C code and also corrects the code using two models first is the CCM which analyzes correct codes before using it to construct a sequence of tokens of correct code. The second is the EOP, which details the editing process. It aims to fix the logical errors in the source code based on the operations that were obtained. If the source code is incorrect, it serves as a new input for CCM. This allows the model to fix source code that has several logic errors.

1.9 ORGANIZATION OF THE REPORT

The project report is organized as follows:

Chapter 2 discusses the existing systems and various methods required for the proposed system.

Chapter 3 discusses the working of various modules of the proposed system with the overall architecture.

Chapter 4 discusses the implementation detail of the proposed system and illustrates the experimental results of the proposed system.

Chapter 5 discusses the results from the work carried out for this project.

Chapter 6 discusses the possible enhancements that can be done in future.

CHAPTER 2

LITERATURE SURVEY

This chapter gives a detailed view on the existing methods related to this project, technologies involved in each method and also gives a brief description on the advantages and disadvantages involved in each of the existing works.

2.1 STATIC APPROACHES FOR BUG-LOCALIZATION

This section gives an overview of various papers mainly focusing on static approaches for logical error localization that refer to techniques that analyze the program code without actually executing it. These techniques typically involve the use of formal methods or program analysis techniques to identify potential logical errors in the code.

2.1.1 Semi-Supervised Verified Feedback Generation

Shalini Kaleeswaran et al. [2] proposes a methodology called semi supervised verified feedback generation. This methodology clusters student submissions based on their solution strategy and asks the instructor to identify or add a correct submission in each cluster. The submissions in each cluster are then verified against the instructor-validated submission in the same cluster, and if faults are detected, feedback suggesting fixes to them is generated. Clustering helps to reduce the burden on the instructor and also handles the variations that have to be managed during feedback generation. The verified feedback generation ensures that only correct feedback is generated. The paper has implemented a tool called CoderAssist based on this approach and evaluated it on dynamic programming assignments

2.1.2 Pattern-Based Error Detection

This paper presented by Yuto Yoshizawa et al. [3] proposed a technique which is based on the analysis of the structure patterns of a program and the degree of errors. The approach involves the use of a pattern-based representation of programs, which is then compared to the ideal structure pattern for the given programming language. The system then analyzes the differences between the two patterns to identify potential errors in the program's structure. Additionally, the approach takes into consideration the degree of errors present in the program, using a scoring system to assign weights to different types of errors. This allows for a more accurate and comprehensive assessment of the program's quality and the identification of logic errors. The proposed approach was evaluated on a set of benchmark programs and showed promising results in detecting errors and providing feedback to developers.

2.2 LSTM BASED LOGICAL ERROR CORRECTION

This section gives an overview of various papers mainly focusing on detection and correction of logical errors using LSTM model. LSTM is a variety of recurrent neural networks that is capable of learning long-term dependencies, especially in sequence prediction problems.

2.2.1 Iterative Logical Error Correction

Taku Matsumoto et al. [4] proposes a system that enables debugging of many logic errors in the source code through recurrent trials of error detection, correction, and source code testing. An editing operation predictor in this model uses a list of correction candidates that was initially provided by a deep learning model to forecast the editing operation that will be performed on each correction candidate. The system employs a set of solution codes developed to complete the related programming challenges in a real e-learning system in order to learn the internal parameters of the proposed model. The system uses LSTM-LM to construct CCM which predicts the position of the logical error in the source code. EOP predicts the editing operation performed between the incorrect code and the corresponding correct code. By repeatedly locating and fixing logical errors as well as testing the updated code, the model debugsthe errors present in the source code. The editing operation is predicted by the EOP based on the CCM's correction candidates. EOP attempts to fix the source code's logical error based on the operations that were obtained. The updated source code is then tested to see if it is still correct. Incorrect source code serves as new input information for CCM. Thus the model is able to debug source code that has numerous logic errors.

2.2.2 Bug Detection based on LSTM Networks

Yunosuke Teshima et al. [5] proposes a bug detection techniques based on LSTM networks and language models, a type of probability distribution. In addition, the paper looks into the ideal model for real-world bug detection given that LSTM networks have some hyperparameters. The main objective of the paper is to create a feedback system that flags potential errors in program lines so that students can detect bugs in a specific area of the program code. The gathered solutions are used to train the language model. The model can determine the appearance probabilities of words that will arise based on the program structure through deep learning.

2.2.3 Attentive LSTM Language Model

Md. Mostafizer Rahman et al. [6] proposes a system that utilizes artificial intelligence for assessing and detecting errors and classifying source code as correct or incorrect code. The system provides a sequential language model that evaluates and categorises source code according to the predicted error probability using an attention-mechanism-based LSTM neural network. The attentive mechanism improves the suggested language model's precision for categorising and assessing errors. Using the right source code,the model is trained, and then its performance is assessed. To improve the performance of error detection and source code categorization, the system coupled the attention mechanism with LSTM to strengthen the suggested model. The LSTM attention mechanism model can detect many common errors in source code, including logic errors. The model can also use long source code sequences as the input to generate the optimal output.

2.2.4 Code Evaluation and Repair using BiLSTM

Keita Nakamura et al. [7] proposes a language model for evaluating source codes using a BiLSTM neural network. The system trained the BiLSTM model with a large number of source codes with tuning various hyperparameters. Then the model was used to evaluate incorrect code and assessed the model's performance in three principal areas: source code error detection, suggestions for incorrect code repair, and erroneous code classification. Although LSTM neural networks are useful for predicting or outputting based on previous input sequences, they may not be optimal for source code analysis since functions, classes, methods, and variables in a code may depend on both previous and subsequent code sections. To address this limitation, a BiLSTM language model is proposed to evaluate and repair source codes. The BiLSTM neural network can combine both past and future code sequences to produce output. To apply this model, the source code is first pre-processed and encoded with a sequence of IDs. Then, the encoded source codes are used to train the BiLSTM neural network. Finally, the trained BiLSTM model can be used for source code evaluation and repair.

2.3 ERROR PREDICTION USING DEEP LEARNING

This section gives an overview of various papers mainly focusing on logical error prediction using deep learning which involves using machine learning techniques to predict the likelihood of a program containing logical errors. Deep learning models, such as neural networks, can be trained on large datasets of program code to learn patterns and features that are indicative of logic errors.

2.3.1 Predicting Logical Errors in Programming Exercises

Dezhuang Miao et al. [8] the paper proposed PIPE, a deep learning model designed to predict logical programming errors in student programs. The model consists of a representation learning component for obtaining latent features of a program and a multi-label classification component for predicting error types. This allows for end-to-end learning and prediction. The model was trained on C programs submitted to an online judge system and showed superior performance compared to baseline models. The error-feedback feature was implemented in the online judge system using PIPE, providing automated feedback to students on logical programming errors. PIPE is based on two models Code to vector and C Code AutoEncoder, which are originally developed to predict semantic properties of code snippets and boost the performance of multi-label classification tasks, respectively. Code2vec was used to obtain the latent representations of C programs. C2AE was used to obtain latent representations of error types.

2.3.2 DeepFix: Fixing Language Errors

Rahul Gupta et al. [9] presents an end-to-end solution, called DeepFix, that can fix multiple such errors in a program without relying on any external tool to locate or fix them. At the heart of DeepFix is a multi-layered sequence-to-sequence neural network with attention which is trained to predict erroneous program locations along with the required correct statements. The network is trained to predict an erroneous program location along with the correct statement. DeepFix invokes it iteratively to fix multiple errors in the program one-by-one.The paper highlights the challenges of automatically fixing common programming errors and the high accuracy bar required for program repair.

2.4 ERROR LOCALIZATION USING NEURAL NETWORKS

This section gives an overview of various papers mainly focusing on logical error localization using neural networks that involves using a machine learning approach to identify the location of logical errors in a code. The neural network is trained on a dataset of code snippets with known logical errors and their corresponding error locations.

2.4.1 Neural Attribution for Semantic Bug-Localization

Rahul Gupta et al. [10] proposes a system called NeuralBugLocator which is a technique based on deep learning that can locate bugs in a faulty program without actually running it, with respect to a failing testcase. The proposed technique is capable of identifying the location of bugs in a faulty program, in relation to a failing test, without the need to run the program. The technique operates in two phases. In the first phase, a tree convolutional neural network is trained to predict whether a program passes a specific test, using a combination of the program and test ID as input. In the second phase, a neural prediction attribution technique is employed to identify the lines of code within the faulty program that contribute to the network's prediction of failure.

2.5 LIMITATIONS OF EXISTING SYSTEMS

- The correction candidates indicated by CCM are likely to have logical errors [4].
- The existing BiLSTM model may not be scalable enough to handle large-scale codebases or complex programs, which could limit its real-world usefulness [7].
- The paper focuses only on predicting the error and not localizing the error i.e., telling the students which lines of the program may contain logical errors and what are the potential types of the errors [8].
- The existing system is not implemented for arbitrary programs in the context of regression testing to localize bugs in a program [10].
- The existing approach is tailored to specific programming languages, and may not be applicable to other languages. This limits the scope and generalizability of the approach [3].
- The model does not use the later parts of the code for prediction of logical errors in the incorrect source code [5].

CHAPTER 3

SYSTEM DESIGN

This chapter deals with the overall architecture of the automated diagnosis and correction of logical errors in C programs and description about each module.

3.1 OVERALL DESIGN AND DESCRIPTION

Figure 3.1 gives the overall system design of this project.This project implements tokenizer, next word probability distribution model, error localiser, edit operation predictor and an editor modules.

The input to the system involves a code wherein logical error has to be detected and corrected. A copy of the code is created on which corrections will be performed and the original source code is retained. The code is iteratively corrected and tested until all the testcases are passed.

Testcases extracted from the database are used by the tester to check if code is correct. Correct codes are used to train the LSTM model which predicts the next word given current word. The error localiser selects current word whose next word and predicted next word don't match. EOP predicts edit,replace and delete operations to be performed to correct the code. Tokenizer is used to convert the code to a vector form suitable for ML.

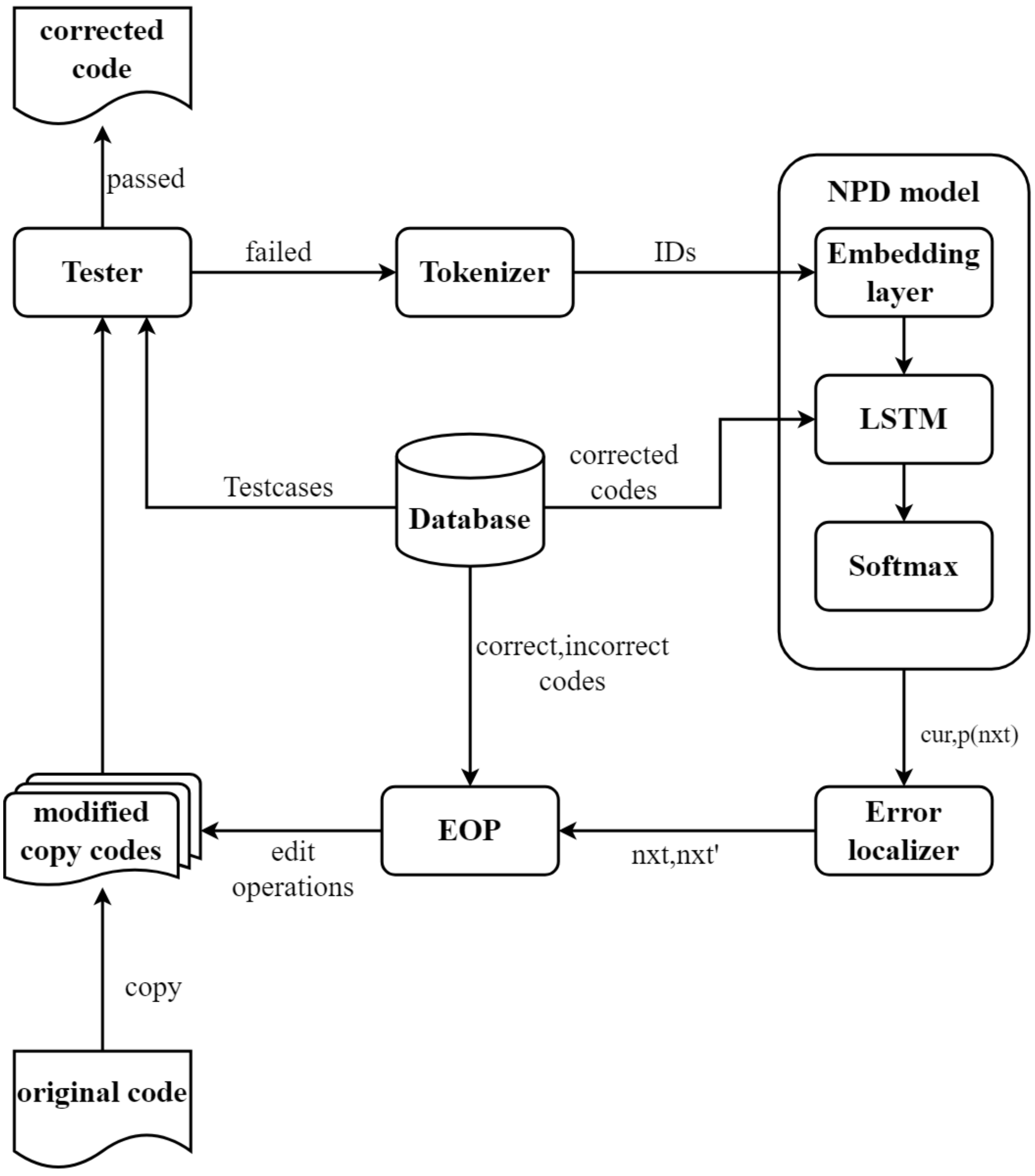


Figure 3.1: Architecture Diagram of Logical Error Detector - Corrector

3.2 SYSTEM MODULES OF PROPOSED WORK

The proposed system follows a tightly coupled architecture as shown in the Figure 3.1 comprising of the following modules.

1. Tokenizer
2. Next Word Probability Distribution Model
3. Error Localiser
4. Edit Operation Predictor
5. Data Extraction and Model Training

3.2.1 Tokenizer

Figure 3.2 shows the tokenization module wherein the source code is split into a sequence of words by recognizing words from the vocabulary table, the words are then mapped to IDs as predefined by a table that maps words to IDs. Thus the input source code is split into tokens with each mapped to an ID. This module is used to tokenize a C file and return a list of tokens. The tokens are classified into different types such as keywords, identifiers, literals, and punctuation. The tokenizer also has the ability to identify functions and variables in the source code.

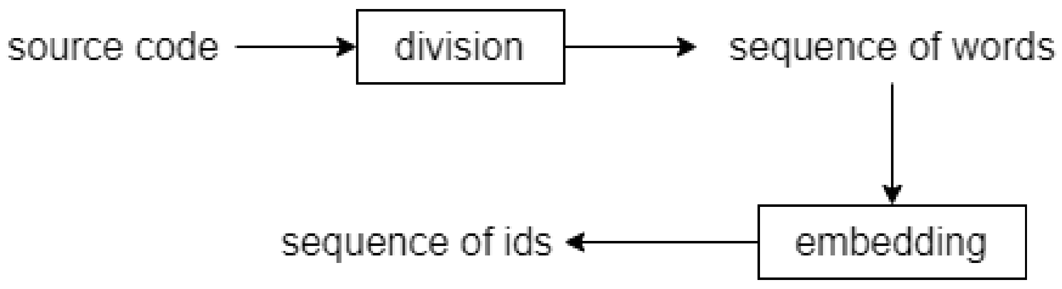


Figure 3.2: Tokenization

This modules processes a literal token, a punctuation token and an identifier token and returns a list of its characters. The method tokenizes the entire source code and returns a list of tokens. The method splits the C code into a list of functions and returns a list of tuples, where each tuple contains the name of the function, a list of its tokens, and the number of lines in the function.

3.2.2 Next Word Probability Distribution Model

The next word probability distribution model is a statistical model that predicts the probability of the next word in a sequence. The language model [13] enables sentence generation and machine translation based on the structure of the learned data. The model is used to predict the next word in a sequence of words. Figure 3.3 shows the nextword probability distibution model that gives the probability distribution of all possible nextwords from which we can extract predicted next word which is argmax(probability) and the probability of given next word.

The model works by first converting the sequence of words into a sequence of IDs. The IDs represent the position of each word in the C source code. The embedding layer then converts the sequence of IDs into a dense vector of size equal to the size of the vocabulary table. The LSTM model then takes the dense vector as input and predicts the probability of each word in the vocabulary occurring next. The model then outputs the probability distribution of all possible next words.

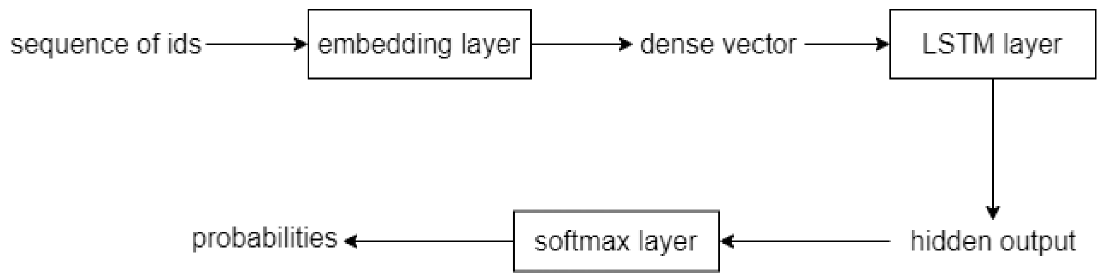


Figure 3.3: Next Word Probability Distribution Model

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}} \tag{3.1}$$

The softmax functions equation as shown in equation 3.1 converts the output to probabilities where z is the vector of raw outputs from the neural network, e is a constant approximately 2.718 and K is the number of classes.

3.2.3 Error Localiser

Figure 3.4 shows the process involved in error localisation module.The input consists of table with the following columns current word, nextword, probability of next word and predicted nextword. The rows are sorted by probability of next word in ascending order. The rows where nextword and predicted next word are different are extracted by which candidate with the highest possibility of a logic error is selected from the correction candidates list which is the output.

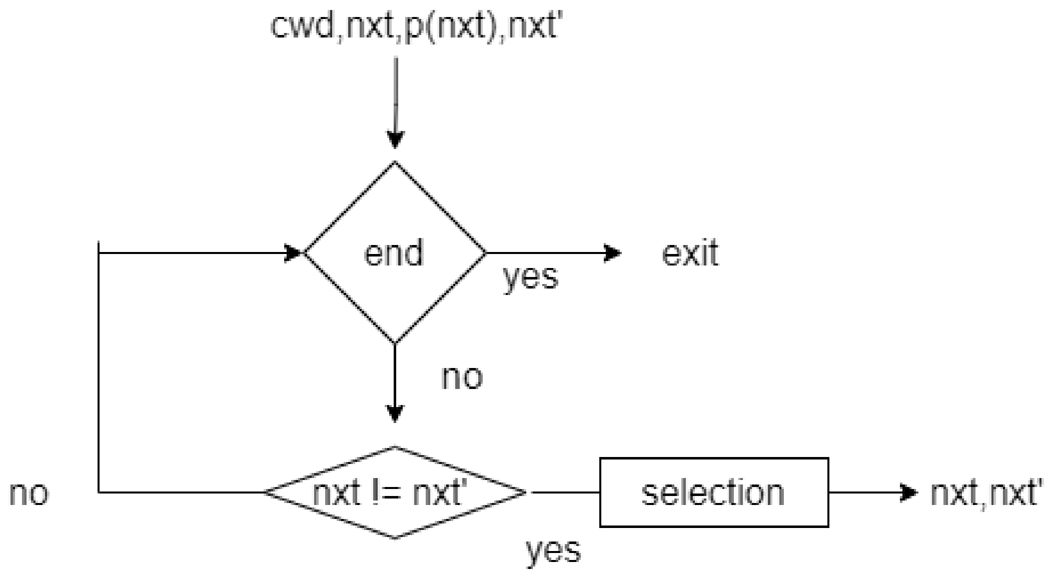


Figure 3.4: Error Localisation

3.2.4 Edit Operation Predictor

Figure 3.5 shows the edit operation predictor model wherein if the token predicted nxt' and the original token nxt from the input do not match for current token cur, the source code needs to be edited. The SVM model is trained using correct and incorrect code pairs from the database. The editing operations that are considered using the three tokens include inserting the predicted token nxt' between token cur and the next token nxt or deleting the next token nxt between token cur and nxt' or replacing the next token nxt with the predicted token nxt'. The output operation predicted by the model is used to accordingly modify the copy code.

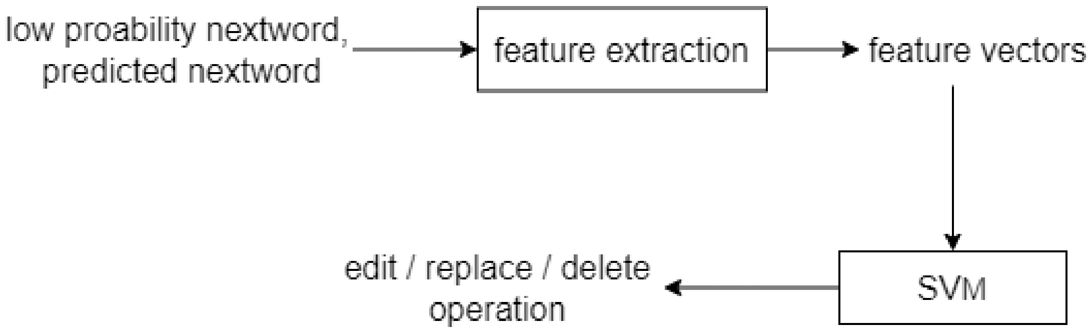


Figure 3.5: Edit Operation Predictor

3.2.5 Data Extraction and Model Training

Figure 3.6 shows the process involved in Data extraction and model training. To extract these source codes, the problemID, userID, judge status, and judgeID included in the metadata are used. Firstly, to classify the metadata by each programming task, metadata whose problemID matches the target problemID is extracted. Next, the metadata classified by each programming is categorized by users. If a user has attempted each programming task multiple times, there may be multiple judge data and source codes. Among them, the last source code whose status is AC is extracted as the correct code. On the other hand, the second-to-last source code whose status is neither AC nor CE is extracted as incorrect code. If there are both incorrect and correct codes created by the user, these source codes are extracted as a pair. By using the judgeIDs corresponding to these source codes, these data sets can be extracted.

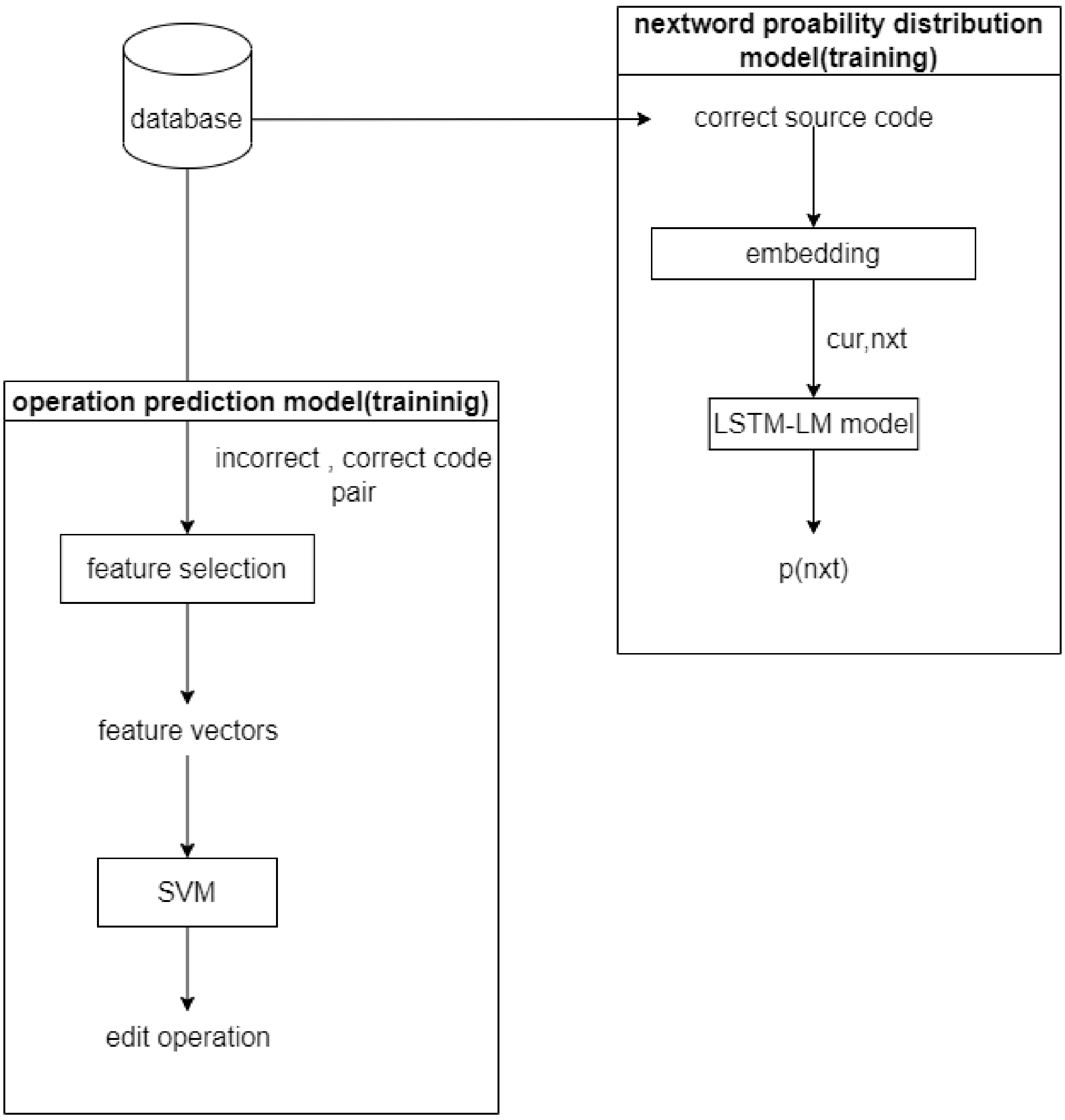

database
nextword proability distribution model(training)
correct source code
embedding
cur,nxt
LSTM-LM model
p(nxt)
operation prediction model(traininig)
incorrect , correct code pair
feature selection
feature vectors
SVM
edit operation

Figure 3.6: Model Training

CHAPTER 4

IMPLEMENTATION

This chapter deals with the modules and submodules involved in implementing the Automated Diagnosis and Correction of Logical Errors in C programs.

4.1 DATASET DESCRIPTION

Aizu Online Judge is one of the online judge systems that can automatically judge the source code submitted by learners [11]. AOJ rigorously evaluates the submitted source code in the AOJ's Judge Server, which is equipped with an execution environment [12]. The AOJ has test cases for each input and output to verify whether the source code meets the specifications of a programming task. If the source code passes all these test cases, it is evaluated as correct code. On the other hand, if the source code cannot pass even one of these test cases, it is evaluated as incorrect code.

4.2 TOKENIZATION

Tokenizer class can be used to tokenize C or C++ code and convert it into a sequence of tokens that can be used as input for machine learning models. The Tokenizer class uses the Clang Python bindings to parse C or C++ code and extract tokens from it. The method uses Clang to extract tokens from the code and processes them based on their type. For example, punctuation tokens are filtered out, while identifiers are processed based on their type (function or variable).

A dictionary called token_to_ix is defined that maps each token to its corresponding index. The index is obtained by enumerating through the tokenlist list and assigning an integer value to each token. If a token is not found in tokenlist, its index is set to 0. The prepare_sequence function returns a PyTorch tensor that contains the index of each token in the input sequence, as specified by the to_ix dictionary. If a token is not found in to_ix, its index is set to 0. The resulting tensor is of type torch.long, which is a PyTorch data type for integer tensors. The tokenization algorithm is explained in algorithm 4.1.

Algorithm 4.1 Tokenization

```
Input: Source Code
Output: Tensor of integers that represent each token of the source code
1: tokenlist = ["", "continue", "unsigned", "default", "typedef", "define",
   "double", "extern", "signed", "sizeof" ... ]
2: Create an empty dictionary named toke_to_ix
3: for i in tokenlist do
4:     token_to_ix[i[1]]=i[0]
5: end for
6: function prepare_sequence(seq , to_ix)
7:     idxs = [to_ix.get(w,0) for w in seq]
8:     return torch.tensor(idxs)
9: end function
```

4.3 NEXT WORD PROBABILITY DISTRIBUTION MODEL

A CCM class is defined in algorithm 4.2, which is a PyTorch module. The purpose of this class is to define a neural network model that can be used for character-level language modeling. The model architecture consists of an embedding layer, a dropout layer, an LSTM layer, and a linear layer. The input tensor is first passed through the embedding layer, then through the dropout layer, and then through the LSTM layer. The output of the LSTM layer is then passed through the linear layer to get the final output probabilities for each token in the vocabulary. The output probabilities are obtained by applying a softmax activation function to the output of the linear layer.

When an input sequence is given, the model generates a probability distribution over the next character in the sequence. This distribution is then sampled from to obtain a predicted next character, which is appended to the input sequence. This process is repeated to generate a sequence of characters, which constitutes the output of the model.

Algorithm 4.2 CCM Model

```
function initialization($embedding\_dim = 32, hidden\_dim = 64, num\_layers = 4, vocab\_size = 177$)
    token_embeddings = nn.Embedding(vocab_size, embedding_dim)
    dropout = nn.Dropout(p=0.5)
    lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers, dropout=0.5)
    hidden2tag = nn.Linear(hidden_dim, vocab_size)
end function
function forward($x$)
    embeds = token_embeddings(x)
    dropout = dropout(embeds.float())
    lstm_out,_ = lstm(dropout.view(x.size(dim=0), x.size(dim=1)))
    tag_space = hidden2tag(lstm_out.view(x.size(dim=0), x.size(dim=1)))
    tag_prob = F.softmax(tag_space, dim=2)
    return tag_prob
end function
```

4.3.1 Next Word Probability Distribution Model Training

CrossEntropyLoss() is a commonly used loss function in classification problems with multiple classes. It is often used in combination with a softmax activation function in the final layer of a neural network. The nn.CrossEntropyLoss() function computes the softmax of the input tensor and then computes the negative log-likelihood loss between the predicted probabilities and the target labels. It combines the computation of softmax and negative log-likelihood loss into a single efficient function as shown in algorithm 4.3.

Adam optimizer is an optimization algorithm used to update the parameters of the neural network model in order to minimize the loss function during the training process. It is a variation of the stochastic gradient descent optimization algorithm and is widely used in deep learning applications.

Algorithm 4.3 Cross Entropy Loss Function and Adam Optimizer

```
1: loss_fn = nn.CrossEntropyLoss()
2: optimizer = torch.optim.Adam(model.parameters(), lr=0.001, betas=(0.9,
   0.999), eps=1e-8)
```

The train_one_epoch which is used to train a machine learning model for one epoch The function takes two arguments as explained in algorithm 4.4, epoch_index, which indicates the current epoch number, and tb_writer which is a tensorboard writer object used to log training progress.Within the function, there is a loop that iterates over the batches in the training data. For each batch, the input and label data are extracted , the gradients are zeroed and the model is used to make predictions for the inputs.

Then, the loss is computed using the predictions and true labels and the gradients of the loss with respect to the model parameters are computed using backpropagation. The model parameters are then updated using an optimizer after gradient clipping. During the loop, the running loss is accumulated and after every 1000 batches the average loss per batch is computed. At the end of the loop, the function returns the last computed loss value.

Algorithm 4.4 NPD Model Training

```
function train_one_epoch(epoch_index,tb_writer)
    running_loss = 0.
    last_loss = 0.
    for i, data in training_loader do
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs.to(device))
        loss = loss_fn(outputs,labels.to(device))
        loss.backward()
        optimizer.step()
        if not torch.isnan(loss) then
            running_loss += loss.item()
        else
            running_loss += 10
        end if
        if i % 1000 == 999 then
            last_loss = running_loss / 1000
        end if
        return last_loss
    end for
end function
```

4.3.2 Error Localisation

The localise function takes a model object and a list of tokens x as input as shown in algorithm 4.5. The purpose of this function is to identify potential errors in the token list x using the given model. For each token in the input list, the function checks whether the token with the highest probability from the model's output is different from the original token. If they are different, it assumes an error and adds information about the potential correction to a list.

The information includes the index of the token with the lowest probability, the original token, the next token in the list, the suggested correction, and the indices of the tokens before, during, and after the potential error.The list of potential corrections is sorted based on the index of the token with the lowest probability, and then returned as the output of the function.The localise function works by first predicting the next token for each token in the input list. The predicted token is the token with the highest probability of occurring next, according to the model. If the predicted token is different from the original token, then the function assumes that there is an error in the input source code.

Algorithm 4.5 Localisation of logic errors

```
1:  function localise($model$, $x$)
2:      p <- CCM(x)
3:      listCorrections = []
4:      for t in range(len(x) do
5:          xBtn = tokenlist[p[t].argmax()]
6:          if x[t+1]!=xBtn then
7:              listCorrections.append([p[t].argmin(), x[t], x[t+1], xBtn, t, t+1, t+2])
8:          end if
9:      end for
10:     listCorrections.sort() by probabilities
11:     return listCorrections
12: end function
```

4.3.3 Generation Of Probabilistic Corrections

The probabalise function takes in a model and an input sequence x, and returns a list of tuples containing information about each token in the input sequence as shown in algorithm 4.6. The function first creates a list listCorrections to store the information about each token. Then it creates a list y that contains the next token for each token in the input sequence x. The first element of y is set to 0.The function then iterates through the tokens in x and y.

The function runs the input sequence through the model which returns a tensor. This tensor represents the probability of each token in the vocabulary for each position in the input sequence. The function then iterates through each token in the input sequence and its corresponding probabilities in p. For each token, it appends a tuple to listCorrections containing the following information the current token, the next token, the probability of the next token given the current token, the maximum probability in the corresponding position of p, the predicted next token based on the maximum probability.

Algorithm 4.6 Probabilization of Logic Errors

```
1: function probablise(model, x)
2:     listCorrections = []
3:     y=[*x,0][1:]
4:     for t, pt in (p) do
5:         listCorrections.append((x[t],y[t], pt[token_to_ix[y[t]]].item(),
6:         pt[pt.argmax()].item(),tokenlist[pt.argmax()]))
7:     end for
8:     return listCorrections
9: end function
```

4.4 EDIT OPERATION PREDICTION MODEL

The EOP predicts an editing operation from a correction candidate obtained by LSTM-LM. Based on the editing operation, EOP seeks to correct the logic error in the source code. Editing operations of source code can be classified into three categories: insertion, deletion, or replacement of tokens. Therefore, we consider the prediction of edit operations for source code as a multi-classification problem. EOP was constructed using Scikit-learn [15]. EOP, uses SVM model, which can solve multi-classification problems.

The source code must be modified to use the predicted token nxt' for the position of the token cur and the token nxt if the predicted token nxt' and the original token cur differ. Insert the predicted token nxt' between the current token cur and the next token nxt; alternatively, delete the current token cur and the next token nxt; or replace the current token nxt with the predicted token nxt'.

The data that is got from Aizu Online Judge System is filtered according to various aspects. The CSV file is expected to have 11 columns with headers: ”judge_id”, ”user_id”, ”problem_id”, ”language”, ”accuracy”, ”status”, ”cpu_time”, ”memory”, ”code_size”, ”submission_date”, and ”judge_date”. The filter selects only the rows where the ”language” column contains C and the ”status” column contains ”Accepted”. Then it checks if a file exists in the ’code’ directory with the name specified by the ’judge_id’ column of the row. If such a file exists, it copies it to a new directory called ’filtered_code’ with the same name. Next it selects all rows in the dataFrame where the combination of ”problem_id” and ”user_id” appears more than once in the original dataset.The data records are then passed through the tokenizer class.

4.4.1 Wagner-Fischer Algorithm for Edit Distances

The Wagner-Fischer Algorithm is a dynamic programming algorithm that measures the Levenshtein distance or the edit distance between two strings of characters. Levenshtein Distance calculates how similar are two strings. The distance is calculated by three parameters to transform the one string to another. They are the number of deletions,insertions and substitutions. First, the dynamic programming table cells are defined as tuples of (partial cost, set of all operations reaching this cell with minimal cost). As a result, the completed table can be thought of as an unweighted, directed graph. The bottom right cell the one containing the Levenshtein distance is the start state and the origin as end state. The set of arcs are the set of operations in each cell as arcs. Every path between the bottom right cell and the origin cell is an optimal alignment. These paths can be efficiently enumerated using breadth-first traversal.

The Wagner-Fischer Algorithm is explained in algorithm 4.7. The function first declares a 2D array d to store the distances between the strings. The array d is indexed by the positions of the characters in the strings.Then it initializes the first row and column of the array d to the lengths of the strings and iterates through the rows and columns of the array d. For each row and column, the function calculates the distance between the characters in the strings at those positions. The function then updates the value of the array d at that position with the minimum distance between the characters in the strings.Finally, it returns the value of the array d at the last row and column which is the levenshtein distance between the two strings.

Algorithm 4.7 Wagner–Fischer Algorithm

```
function Distance($s[1..m]$, $t[1..n]$)
    declare int d[0..m, 0..n]
    for i from 1 to m do
        d[i, 0] := i
    end for
    for j from 1 to n do
        d[0, j] := j
    end for
    for j from 1 to n do
        for i from 1 to m do
            if s[i] = t[j] then
                substitutionCost := 0
            else
                substitutionCost := 1
            end if
            d[i, j] := minimum(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + substitutionCost)
        end for
    end for
    return d[m, n]
end function
```

4.4.2 Support Vector Machine

The SVM algorithm is used to learn the relationship between the feature vectors and the labels for editing operations. Here the feature vectors are the extracted representations of the source code, and the labels are the types of editing operations that are required to correct the incorrect code. During the training phase, the SVM algorithm learns to classify the feature vectors based on the corresponding editing operation labels. Once trained, the SVM can then be used to predict the correct editing operation for new, unseen incorrect code submissions. Overall, SVM is used as a classification tool to learn the relationship between the extracted feature vectors and labels for editing operations, enabling the EOP system to automatically correct incorrect code submissions based on the learned patterns.

Algorithm 4.8 Training an SVM

```
Require: X and y loaded with training labeled data, $\alpha \Leftarrow 0$ or $\alpha \Leftarrow$ partially
    trained SVM
 1: C $\Leftarrow$ source code
 2: repeat
 3:     for all $(x_i, y_i), (x_j, y_j)$ do
 4:         Optimize $\alpha_i$ and $\alpha_j$
 5:     end for
 6: until no changes in $\alpha$ or other resource constraint criteria met
Ensure: : Retain only the support vectors ($\alpha_i > 0$)
```

In the above algorithm 4.8, X represents a dataframe that consists of sum of the vector 1, vector 2, vector 3, and count columns and y DataFrame contains the sum of all the columns in each row.Both these dataframes are fit into the SVM model.When a C source code is given as input it is tokenized and converted to feature vectors which is used by the model to make predictions.

CHAPTER 5

RESULTS AND PERFORMANCE ANALYSIS

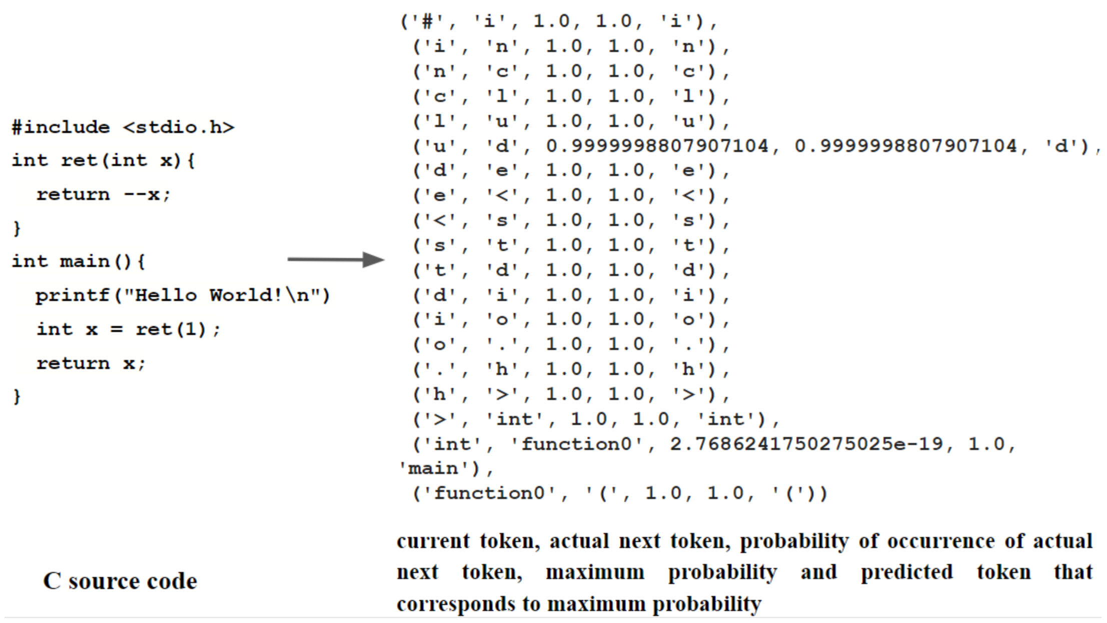
In this chapter, the aim is to establish that the system developed to detect and correct logical errors is significantly an improved method compared to the traditional debugging systems with the help of certain standard metric evaluations. The model used to develop this system is the Next Word Probability Distribution Model and the Edit Operation Predictor. The dataset used is the Aizu Online Judge system which has several source codes. The dataset is split into train set and validation set.

5.1 NEXT WORD PROBABILITY DISTRIBUTION MODEL

The model provides a combination of error localisation and probablistic corrections as output. To train the model a batch size of 4 is set, the number of hidden neurons to 64 ,cross entropy is chosen as the loss function. To prevent overfitting of the model, the Adam optimizer is used with four parameters [14]. The dropout rate was set to be 0.5. Five-fold cross validation was used to evaluate the performance of the model. Number of training data was divided into 5 parts, four of which were used for training, and one was used as test data. The NPD model works by first creating a probability distribution over all possible words. This probability distribution is created by counting the number of times each word appears in the training corpus.

Then, the NPD model predicts the next word by finding the word with the highest probability. The probability of a word is calculated by multiplying the probability of the word occurring in the current position by the probability of the word occurring after the current word. Figure 5.1 shows the results of the Next

Word Probability Distribution Model, given and input C source code the output is as follows: the current token, the actual next token, probability of the occurrence of the actual next token, maximum probability and the predicted token that corresponds to maximum probability.



```c
#include <stdio.h>
int ret(int x){
  return --x;
}
int main(){
  printf("Hello World!\n")
  int x = ret(1);
  return x;
}
```

C source code

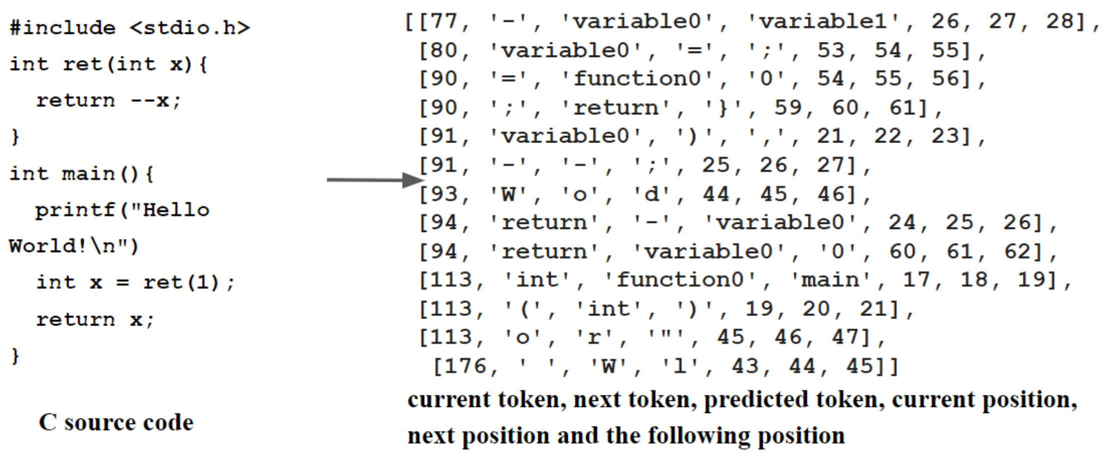
```
('#', 'i', 1.0, 1.0, 'i'),
 ('i', 'n', 1.0, 1.0, 'n'),
 ('n', 'c', 1.0, 1.0, 'c'),
 ('c', 'l', 1.0, 1.0, 'l'),
 ('l', 'u', 1.0, 1.0, 'u'),
 ('u', 'd', 0.9999998807907104, 0.9999998807907104, 'd'),
 ('d', 'e', 1.0, 1.0, 'e'),
 ('e', '<', 1.0, 1.0, '<'),
 ('<', 's', 1.0, 1.0, 's'),
 ('s', 't', 1.0, 1.0, 't'),
 ('t', 'd', 1.0, 1.0, 'd'),
 ('d', 'i', 1.0, 1.0, 'i'),
 ('i', 'o', 1.0, 1.0, 'o'),
 ('o', '.', 1.0, 1.0, '.'),
 ('.', 'h', 1.0, 1.0, 'h'),
 ('h', '>', 1.0, 1.0, '>'),
 ('>', 'int', 1.0, 1.0, 'int'),
 ('int', 'function0', 2.7686241750275025e-19, 1.0,
'main'),
 ('function0', '(', 1.0, 1.0, '('))
```

current token, actual next token, probability of occurrence of actual next token, maximum probability and predicted token that corresponds to maximum probability

Figure 5.1: Probablistic Corrections

To localize errors, the NPD model is used to predict the next token in a sequence. If the predicted next token does not match the actual next token, then there is an error in the input C source code. The model predicts the probability of each token in the sequence. If the actual next token has a lower probability of occurring than the predicted token, the error is more likely to be in the actual next token.

Figure 5.2 shows the results of error localisation wherein given a C source code the results are as follows if the token with maximum probability is not equal to the next token then the current token, next token, predicted token, current position, next position and the following position is appended into a list. The results of NPD model are filtered out wherein only those current tokens for which the actual next token has lesser probability of occurring than predicted token are passed through the filter. This filtering process helps to improve the accuracy of error localization by removing results that are likely to be false positives.



```
#include <stdio.h>
int ret(int x){
  return --x;
}
int main(){
  printf("Hello
World!\n")
  int x = ret(1);
  return x;
}
```

C source code

```
[[77, '-', 'variable0', 'variable1', 26, 27, 28],
 [80, 'variable0', '=', ';', 53, 54, 55],
 [90, '=', 'function0', '0', 54, 55, 56],
 [90, ';', 'return', '}', 59, 60, 61],
 [91, 'variable0', ')', ',', 21, 22, 23],
 [91, '-', '-', ';', 25, 26, 27],
 [93, 'W', 'o', 'd', 44, 45, 46],
 [94, 'return', '-', 'variable0', 24, 25, 26],
 [94, 'return', 'variable0', '0', 60, 61, 62],
 [113, 'int', 'function0', 'main', 17, 18, 19],
 [113, '(', 'int', ')', 19, 20, 21],
 [113, 'o', 'r', '"', 45, 46, 47],
  [176, ' ', 'W', 'l', 43, 44, 45]]
```

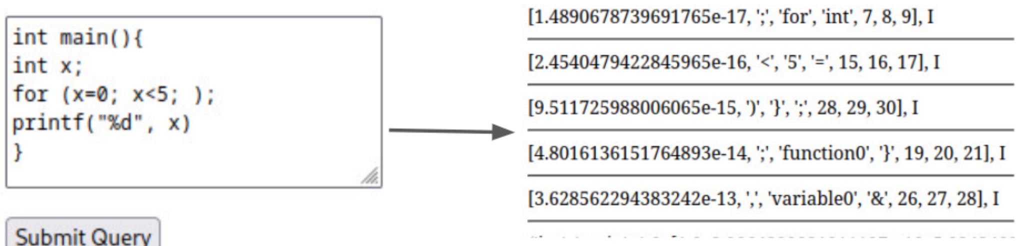
current token, next token, predicted token, current position, next position and the following position

Figure 5.2: Localisation of Errors

5.2 EDIT OPERATION PREDICTION

The EOP model is used to correct errors in C source code. It is trained on a large corpus of correct C source code, and can be used to correct errors in new C source code. The EOP model works by first identifying the location of errors in the C source code. This is done using a technique called error localization. Once the errors have been localized, the EOP model then predicts the correct code for each error. The model gives a list of operations to be made in the C source code in order to correct the incorrect source code.

The results of EOP model are shown in Figure 5.3 given an incorrect C source code the output when passed through the SVM model we get the output with values of the current token, next token, the following token, current position, next position, the following position of the corresponding tokens and also an operation like Insert to insert the predicted token between current token and the next token, Delete to delete the next token between current token and predicted token, Substitute to substitute the next token with the predicted token. The above one of the operations is appended to the error localised output.



```
int main(){
int x;
for (x=0; x<5; );
printf("%d", x)
}
```

```
[1.4890678739691765e-17, ';', 'for', 'int', 7, 8, 9], I
[2.4540479422845965e-16, '<', '5', '=', 15, 16, 17], I
[9.511725988006065e-15, ')', '}', ';', 28, 29, 30], I
[4.8016136151764893e-14, ';', 'function0', '}', 19, 20, 21], I
[3.628562294383242e-13, ',', 'variable0', '&', 26, 27, 28], I
```

C source code

current token, next token, predicted token, current position, next position, the following position and edit operation

Figure 5.3: Edit Operation Prediction

5.3 METRICS USED FOR EVALUATION

The following metrics are used for evaluating the performance of the model and obtain the results.

5.3.1 Learning Curves for Loss

A loss function, is a measure of how well a machine learning model is performing on a given task. The goal of the loss function is to provide feedback to the model during training so that it can improve its predictions. It compares the predicted output of the model to the true output and calculates the error between them. This error is then used to adjust the model's weights and biases during the training process, with the goal of minimizing the error.

Cross-entropy loss measures the difference between the predicted probability distribution and the true probability distribution of the target class. The predicted probability distribution is generated by the model, while the true probability distribution is the one-hot encoded vector representing the target class. It is a metric used to measure how well a classification model in machine learning performs. It's value ranges from zero to worst case of total number of classes. Given that the classification is among 178 classes.

The cross-entropy loss is minimized during training by adjusting the model parameters so that the predicted distribution more closely matches the target distribution. The model predicts the probability of each class for the input data. The target distribution is calculated by setting the probability of the correct class to 1 and the probability of all other classes to 0. It is calculated by taking the negative of the sum of the logarithms of the predicted probabilities for each class. Cross Entropy loss is formulated as shown in equation (5.1) where 'ti' refers to weight of each class 'i',and 'pi' represents probability of belonging to class 'i'.

$$L_{CE} = -\sum_{i=1}^{n} t_i log(p_i) \tag{5.1}$$

The data is split into 5 folds using the KFold class. This ensures that each fold contains a representative sample of the data. For each fold, the model is trained on 4 folds and validated on the remaining fold. This is done to avoid overfitting the model to the training data. The training and validation losses are logged to TensorBoard to track the progress of the model and identify any potential problems. The model with the best loss is saved this ensures that the model performs best on unseen data. This project is evaluated using a graphical representation on the training set as seen in Figure 5.4. As we can see in the below graph a cross entropy loss of 4.51 has been achieved. This is a relatively low loss, which indicates that the model is confident in its predictions. Hence there is high probability that the correct token is in the top five predicted tokens.

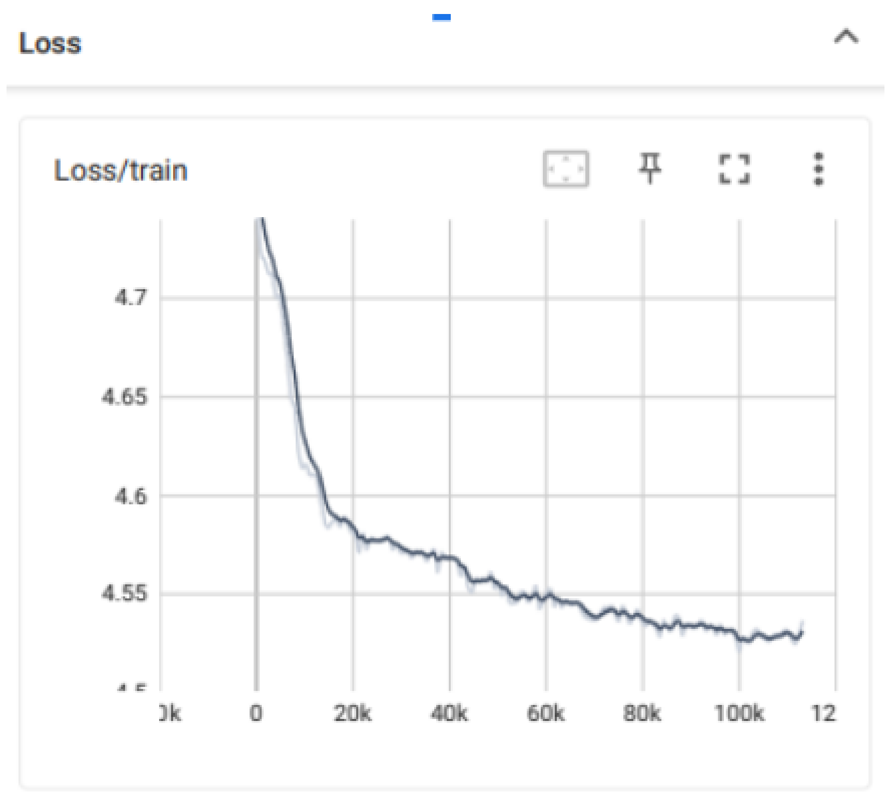
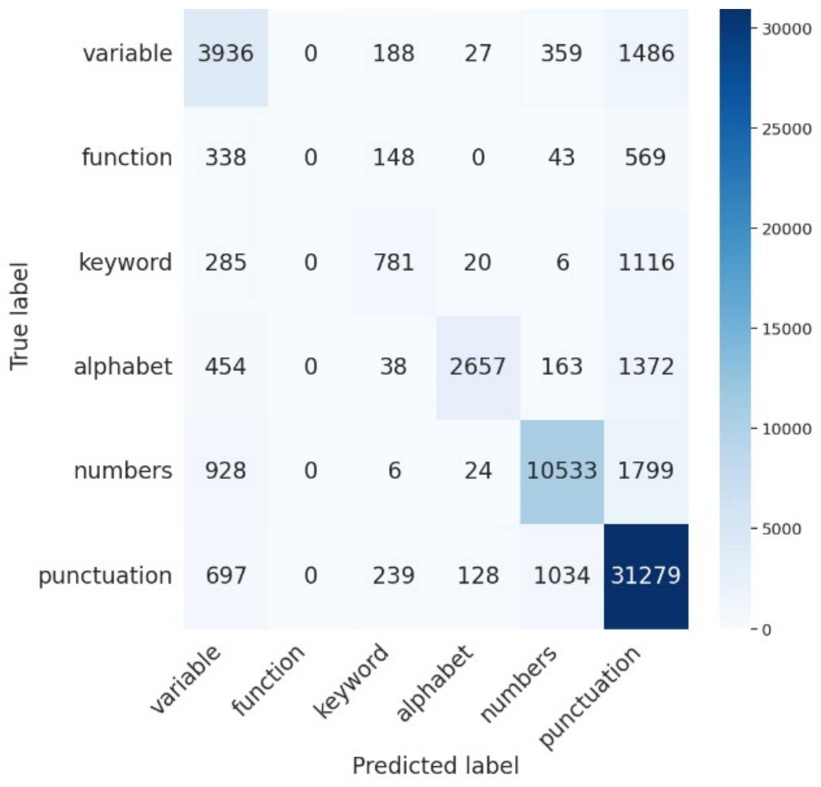


Figure 5.4: Loss curve for Next Word Probability Distribution Model

5.3.2 Confusion Matrix

Confusion matrix is a measure used while solving classification problems. It can be applied to binary as well as for multiclass classification problems. Confusion matrices represent counts from predicted and actual values. The confusion matrix has four main components: true positives are instances that were correctly classified as positive, false positives are instances that were incorrectly classified as positive, true negatives are instances that were correctly classified as negative, false negatives are instances that were incorrectly classified as negative,



	variable	function	keyword	alphabet	numbers	punctuation
variable	3936	0	188	27	359	1486
function	338	0	148	0	43	569
keyword	285	0	781	20	6	1116
alphabet	454	0	38	2657	163	1372
numbers	928	0	6	24	10533	1799
punctuation	697	0	239	128	1034	31279

Figure 5.5: Confusion Matrix

The confusion matrix shown in Figure 5.5 shows the results of a classification task. The rows of the table represent the true labels of the data, and the columns represent the predicted labels. The diagonal values in the table represent the number of data points that were correctly classified. The off-diagonal values represent the number of data points that were incorrectly classified. The sum of each row represents the total instances actually belonging to the class. The diagonal cell corresponding to the row represents the number of such instances that were correctly classified. The accuracy of the model is represented by the sum of the the diagonal values divided by the sum of all the cell values. The class wise accuracy of the model is determined by dividing the corresponding diagonal for each label by the row.

5.3.3 Accuracy

Accuracy is a metric used in machine learning to measure how well a classification model is able to correctly predict the class of a given sample. It is the ratio of correctly classified samples to the total number of samples in the dataset. It is calculated as per formulation in equation (5.2) wherein 'tp' is the true positive rate, 'fp' is false positive, 'fn' is false negative and 'tn' is true negative for l classes. Here accuracy is represented as the ratio of correctly classified tokens (tp+tn) to the total number of tokens (tp+tn+fp+fn).

$$\frac{\sum_{i-1}^{l} \frac{tp_i+tn_i}{tp_i+fn_i+fp_i+tn_i}}{l} \tag{5.2}$$

Model accuracy is a measure of how well a model predicts the correct output for a given input. It is calculated by dividing the number of correct predictions by the total number of predictions made. When the SVM model is trained and tested on the dataset, an accuracy score of 82.25% is achieved means that the SVM model was able to correctly predict the next tokens and corresponding edit operations like insert, delete, substitute of 82.25% of the samples in the test set.That is 82.25% of next tokens and corresponding edit operations are correctly predicted.

The existing approach of predicting next tokens for a generic program without either a reference correct program nor the question associated with the program has an accuracy of 13.90% as shown in Table 5.1 whereas the proposed NPD model has attained a significant increase in the accuracy of 45.70%. This means that the existing approach was only able to correctly predict the next token for 13.90% of the samples in the test set.

Model	**Existing system**	**Proposed System**
NPD	13.90%	45.70%
EOP	80.00%	82.25%

Table 5.1: Accuracy Comparison.

The proposed EOP model is a supervised learning model. This means that it is trained on a dataset of labeled data. The labeled data consists of pairs of strings, where the first string is the previous tokens and the second string is the next token. The proposed model uses a support vector machine SVM to learn the relationship between the next token and the previous tokens. It also uses a technique called edit distance to correct errors in the predicted next tokens. Edit distance is a measure of the similarity between two strings. The proposed model uses edit distance to identify the most likely edit operations that need to be performed to correct the predicted next tokens. Thus the proposed approach as an increase in the accuarcy of 82.25% from the existing approach as shown in Table 5.1.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

This documentation explains the working of automatic diagnosis and correction of logical errors. The model uses an iterative process of identifying and correcting errors, and testing the source code.We proposed a method that optimizes a threshold that regulates the number of correction candidates detected by LSTM-LM to improve the reliability of the correction candidates. A debugging support model is developed that introduced an EOP that predicts an editing operation, such as insertion, deletion, and replacement, using the correction candidates detected by the LSTM-LM. The model iteratively corrects the incorrect code until it meets the specification of the programming task. The proposed models for logical error diagnosis which used LSTM-LM achieved an accuracy of 45.70% which is relatively higher than the existing methodologies used and the EOP gave an accuracy of 82.25%.

6.2 FUTURE WORK

Future works will aim to to realize Hybrid Intelligence for logic error detection. To improve the correction performance of our model, we would like to analyze what logic errors are likely to occur in each programming task. The system can also be expanded by developing a deep learning-based logic error model for multiple programming languages such as C, C++, Java, Python, and so on. The increase in likelihood of diagnosis of these errors can be achieved by choosing the correcting candidates that contain only true logic errors.

REFERENCES

[1] BBC bitesize. Logic errors - writing error-free code - ks3 computer science revision, 2022.

[2] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th Association for Computing Machinery Special Interest Group on Software Engineering International Symposium on Foundations of Software Engineering*, page 739–750, 2016.

[3] Yuto Yoshizawa and Yutaka Watanobe. Logic error detection system based on structure pattern and error degree. *Advances in Science, Technology and Engineering Systems Journal*, 2019.

[4] Matsumoto Taku, Watanobe Yutaka, and Nakamura Keita. ”A model with iterative trials for correcting logic errors in source code”,. *Applied Sciences*, 2021.

[5] Yunosuke Teshima and Yutaka Watanobe. Bug detection based on lstm networks and solution codes. In *2018 Institute of Electrical and Electronics Engineers International Conference on Systems, Man, and Cybernetics*, pages 3541–3546, 2018.

[6] Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura. Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education. *Applied Sciences*, 2020.

[7] Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura. A bidirectional lstm language model for code evaluation and repair. *Symmetry*, 2021.

[8] Dezhuang Miao, Yu Dong, and Xuesong Lu. Pipe: Predicting logical programming errors in programming exercises. In *Educational Data Mining*, 2020.

[9] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*, page 1345–1351, 2017.

[10] Rahul Gupta, Aditya Kanade, and Shirish Shevade. *Neural Attribution for Semantic Bug-Localization in Student Programs*. 2019.

[11] Yutaka Watanobe, Md Rahman, Taku Matsumoto, Uday Rage, and Ravikumar Penugonda. Online judge system: Requirements, architecture, and experiences. *International Journal of Software Engineering and Knowledge Engineering*, 32:1–30, 05 2022.

[12] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1), jan 2018.

[13] Vladimir Naumovich Vapnik. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780, 1963.