TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

Minor Project Report

On

Automatic Name-based Software Bug Detection during Static Program Analysis

Submitted By:

Shashank Ghimire (THA076BEI032)

Shirshak Acharya (THA076BEI033)

Yunij Karki (THA076BEI044)

Dipu Dahal (THA076BEI045)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

March, 2023

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

Minor Project Report

On

Automatic Name-based Software Bug Detection during Static Program Analysis

Submitted By:

Shashank Ghimire (THA076BEI032)

Shirshak Acharya (THA076BEI033)

Yunij Karki (THA076BEI044)

Dipu Dahal (THA076BEI045)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor's Degree in Electronics, Communication and Information Engineering

Under the Supervision of

Er. Dinesh Baniya Kshatri

March, 2023

DECLARATION

We hereby declare that the report of the project entitled "**Automatic Name-based Software Bug Detection during Static Program Analysis**" which is being submitted to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, in the partial fulfillment of the requirements for the award of the Degree of Bachelor of Engineering in **Electronics, Communication and Information Engineering**, is a bonafide report of the work carried out by us. The materials contained in this report have not been submitted to any University or Institution for the award of any degree and we are the only author of this complete work and no sources other than the listed here have been used in this work.

Shashank Ghimire (THA076BEI032) _____

Shirshak Acharya (THA076BEI033) _____

Yunij Karki (THA076BEI044) _____

Dipu Dahal (THA076BEI045) _____

Date: March, 2023

CERTIFICATE OF APPROVAL

The undersigned certify that they have read and recommended to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, a minor project work entitled "**Automatic Name-based Software Bug Detection during Static Program Analysis**" submitted by **Shashank Ghimire**, **Shirshak Acharya**, **Yunij Karki** and **Dipu Dahal** in partial fulfillment for the award of Bachelor's Degree in Electronics, Communication and Information Engineering. The Project was carried out under special supervision and within the time frame prescribed by the syllabus.

We found the students to be hardworking, skilled and ready to undertake any related work to their field of study and hence we recommend the award of partial fulfillment of Bachelor's degree of Electronics, Communication and Information Engineering.

_____

Project Supervisor
Er. Dinesh Baniya Kshatri
Department of Electronics and Computer Engineering, Thapathali Campus

_____

External Examiner

_____

Project Co-ordinator
Mr. Umesh Kanta Ghimire
Department of Electronics and Computer Engineering, Thapathali Campus

_____

Mr. Kiran Chandra Dahal
Head of the Department,
Department of Electronics and Computer Engineering, Thapathali Campus

March, 2023

COPYRIGHT

ACKNOWLEDGEMENT

We are thankful to the Institute of Engineering(IOE) for including Minor Project on the syllabus of Bachelor in Electronics, Communication and Information Engineering.

Also, we would also like to express our deep gratitude to **Er. Dinesh Baniya Kshatri** for providing us with the proper guidance and the Department of Electronics and Computer Engineering, Thapathali Campus for creating a wonderful learning environment.

This project would not be possible without the support of our fellow classmates who provided us with constructive criticism, suggestion, and encouragement at all times. We believe that this project will play a great role in uplifting our technical, communication, and team skills to a great extent.

Shashank Ghimire (THA076BEI032)

Shirshak Acharya (THA076BEI033)

Yunij Karki (THA076BEI044)

Dipu Dahal (THA076BEI045)

ABSTRACT

The aim of this project is to develop a tool that can automatically find bugs in source code through the name analysis technique. Name-based bug detection involves analyzing source code to detect potential bugs based on the names or labels used for variables, functions, and other elements in the code. This project utilizes Abstract Syntax Tree (AST) to generate the negative (buggy) samples automatically due to the unavailability of a large set of negative samples. The project initially collects code snippets from C Code Corpus dataset and parses the code snippets into their corresponding AST using LibClang. It then extracts the positive samples from AST, adjusts and swaps contents of positive samples to generate negative samples. Similarly, it performs data cleaning and tokenization of data using fine-tuned CodeT5 Tokenizer and finally the extracted dataset is fed into the DistilBERT model for training to identify potential bugs. Currently, DistilBERT model is used for the detection of bugs related to swapped function arguments and wrong binary operators. The detection of other types of name-based bugs can be easily done following the similar steps taken in developing current models. The resulting system will be able to automatically detect specific type of bugs in source code, providing a valuable tool for software developers and improving the quality of their code.

Keywords: AST, CodeT5, DistilBERT, LibClang

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
AdaGrad	Adaptive Gradient Algorithm
Adam	Adaptive Moment Estimation
ANN	Artificial Neural Network
API	Application Programming Interface
AST	Abstract Syntax Tree
AWS	Amazon Web Services
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformers
BPE	Byte Pair Encoding
CNN	Convolutional Neural Network
CuBERT	Code Understanding BERT
CV	Cross Validation
DFG	Data Flow Graph
DL	Deep Learning
DOM	Document Object Model
ELU	Exponential Linear Unit
FF	Feed Forward
FN	False Positive
FP	False Negative
GD	Gradient Descent
GELU	Gaussian Error Linear Units
GNN	Graphical Neural Network
GPT	Generative Pretrained Transformer
GPU	Graphical Processing Unit
GRU	Gated Recurrent Unit
JDT	Java Development Tools
JSON	Javascript Object Notation
ML	Machine Learning

MLP	Multi Layer Perceptron
NAR-Miner	Negative Association Rules-Miner
NL	Natural Language
NLM	Neural Language Model
NLP	Natural Language Processing
NN	Neural Network
PCA	Principal Component Analysis
PDG	Program Dependence Graph
PL	Programming Language
PReLU	Parametric Rectified Linear Unit
PyBugLab	Python BugLab
PyPIBugs	Python Package Index Bugs
RAM	Random Access Memory
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SSC	Share Specialize Complete
SVD	Singular Value Decomposition
TN	True Negative
TP	True Positive
TPU	Tensor Processing Units
UI	User Interface

1. INTRODUCTION

1.1 Background

Automatic bug detection refers to the process of identifying software bugs (errors or defects) in a program or system using automated tools and techniques, rather than relying solely on manual testing or inspection. This can include techniques such as static analysis, dynamic analysis, and machine learning. The goal of automatic bug detection is to improve the efficiency and effectiveness of software development and maintenance by identifying and addressing bugs early in the development process before they can cause problems for users.

Likewise, name analysis, also known as symbol resolution, is a technique used in automatic bug detection to identify and track the use of variables, functions, and other named entities in a program or system. The process of name analysis involves analyzing the source code of a program to identify the names of variables, functions, classes, and other entities, and then tracking how those names are used throughout the program. This can help to identify potential bugs, such as the use of uninitialized variables or calls to undefined functions. Name analysis can be performed using static analysis tools, which analyze the source code without executing the program, or dynamic analysis tools, which analyze the program while it is running.

Due to the skyrocketing advancement of tools and technology, a large volume of programming languages and programs were required to fulfill simple and easy tasks. With these volume troubleshooting and manual code analysis were found to be tedious and monotonous. Code analysis is the process of systematically reviewing and evaluating source code to identify potential issues, such as bugs, security vulnerabilities, or performance bottlenecks. Although various tools and techniques were developed to overcome these complex tasks, they were unable to completely eradicate problems and vulnerabilities within the code. With this complexity in finding the bugs, different

techniques were adopted among which detection of bugs using the approach of machine learning was quite fascinating.

As the size of the codebase increases, manual code analysis becomes increasingly difficult as it requires more computational resources and more complex algorithms to analyze the code. However, with the advent of machine learning models, the process of code analysis can be made more efficient. Machine learning models can be trained on large codebases and can learn to identify patterns and anomalies in the code that may indicate bugs or other issues. These models can then be used to automatically analyze new code, making it possible to quickly and accurately identify potential bugs. Additionally, as the models are trained on a large dataset, they can adapt to the specific patterns of the codebase and thus improve the accuracy of bug detection.

The rising field of AI has been revolutionizing industries from healthcare to fashion and an enormous amount of research has been done in this field. Traditionally, the development of bug detection tools was done by relying on human experts to define bug features and they often missed many bugs and vulnerabilities (i.e. high false negative rate). These vulnerabilities can cause a serious impact on the industry and some exploiters may get into the system if they find one. So, Machine Learning (ML) techniques have been attractive for bug detection as they make the development of bug detection tools easier than previous techniques of manually defining the features and patterns of bugs. A compiler only detects syntax errors in the source code but not the swapped function arguments, wrong binary operator and wrong binary operand. Hence, the methods in this report can be utilized to correctly classify these types of bugs without relying on a compiler.

1.2 Motivation

In every field related to program or software development, there often arises a problem to hire IT personnel to carry out various tasks related to analyzing the source code and finding bugs and vulnerabilities within the program. It was found to be tedious because the IT personnel always had to go through the same sort of procedure to accomplish the same sort of task. So the author was motivated to provide a sophisticated solution to carry out this

task by replacing human resources with manual code analysis by using machine learning models to identify those bugs. With the increasing security risk in a program, it becomes very much difficult for security researchers to go through static code analysis to identify the loopholes, seeing this also motivated us to provide a method to carry out this task without much human effort using machine learning approach.

In short, the main motivation for this project is to develop an automated bug detection method that is more efficient and accurate than manual inspection, and which can help to improve the reliability and security of software.

1.3 Problem Definition

As software systems continue to grow in complexity and size, the task of identifying and fixing bugs in the code becomes increasingly difficult. Manual inspection and testing can be time-consuming and may not be able to detect all bugs. Even if the code is free of any wrong binary or function arguments bugs, the other static code analysis tools may still report an issue. This can lead to wasted time and effort in reviewing and addressing them. The other static code analysis tools miss these actual aforementioned bugs in the code, leading to undetected bugs.

1.4 Objectives

- To create suitable training data consisting of correct and buggy code from a corpus of source code written in the C-programming language.

- To train and evaluate a transformer model capable of name-related bug detection during static code analysis.

1.5 Scope and Applications

The main aim of this project is to detect and localize the name based bugs like swapped arguments in a function call, wrong binary operator. First, the correct source code is converted into AST. Only the swapped function arguments and wrong binary operator are fetched from AST. Then a negative source code is generated from the corresponding AST.

Bugs in the given source code are first detected and then localization is done through the information provided in the Abstract Syntax Tree (AST) of that source code. Users will be able to upload their source code through a web application which is sent to the web server for bug detection and finally the result is sent to the client side of the web app where it is displayed.

However, along with the features and benefits, this project may fail to detect bugs in the given source code if it contains name-based entities such as variables and functions that are not named properly or named in an abnormal way. As models are only trained for c corpus, the major limitation is that, it is not useful for other programming languages except C.

The project, which is focused on automatic bug detection using name analysis, has a wide range of applications, including software development, maintenance, and code review which are briefly described below:

- **Bug Identification**: The large code bases are time-consuming and highly difficult to inspect and analyze manually. Companies need to hire specific people for the sole purpose of static code analysis or code review before compilation. It may not be practically possible for a programmer to detect bugs in large quantities as bug identification is a tiresome process requiring a lot of manual effort and commitment. The use of machine learning in this project could help to automatically detect bugs in large code bases, reducing the need for manual code review and making the process more efficient.

- **Improving Code Quality**: Bugs and the resulting crashes can highly degrade the software's code quality and customer satisfaction rate. Frequent encounters with bugs may irritate the users and this may result in a decrease in potential profit for a company. By identifying and fixing bugs early, this project could help to improve the overall quality of the code, reducing the number of errors and crashes in the final product.

- **Coding Competitions**: This project can help during various coding competitions as it can detect and locate bugs in the source code and act as an automated judge. Simply by passing the code snippets of those competitions, faults and errors can be detected. The tool can help to ensure that all submissions are evaluated consistently and fairly, reducing the likelihood of human bias or error affecting the results.

- **Coding Interview**: The tool can help candidates identify potential bugs in their code before the interview, allowing them to fix any issues and better prepare for the interview. These can also be used during the interview to quickly and accurately identify potential bugs in the candidate's code, reducing the amount of time required to evaluate their skills. And finally it can enhance the hiring process, as it can help hiring managers to quickly and accurately evaluate a candidate's coding skills, allowing them to make more informed hiring decisions.

- **Boosting Coding Efficiency:** This model can save time and increase efficiency compared to manual bug detection and correction methods because it automates a significant part of the debugging process, freeing up time for developers to focus on other tasks. Additionally, automatic bug detection through name analysis can be more efficient because it eliminates human error and reduces the chances of introducing new bugs while fixing existing ones. By using this method, software development projects can be completed faster and with fewer errors, leading to time-saving and increased productivity of developers.

2. LITERATURE REVIEW

"Self-Supervised Bug Detection and Repair" [1] introduces a selector and detector model for bug detection and repair. Selector and detector models are trained such that the selector is responsible for selecting bugs to be introduced when generating data for the bug detector, and the detector detects the bugs and repairs them. Two models used to train the data in this research are GNN and Relational Transformer where they used an evaluation dataset of Random Bugs (700k random bugs), PyPIBugs (2k real bugs). The bugs mainly identified in this research are Argument Swapping, Wrong Assignment, Wrong Binary Operator, Wrong Boolean Operator, Wrong Literal, Variable Misuse and Wrong Comparison Operator. In the comparison with CuBERT models, the PyBUGLAB models did substantially better with greater recall value than that of the CuBERT-based models.

In the paper “Deep Learning-based Vulnerability Detection: Are We There Yet?” [2] the authors studied different aspects of Vulnerability detection to find vulnerabilities in the code. The paper uses an existing Deep Learning-based Vulnerability Detection prediction system. DL (Deep Learning) based vulnerability prediction approach suffers from problems like data duplication, and unrealistic distribution of vulnerable classes due to which this approach doesn’t learn what caused that vulnerability. They learn unnecessary items like variable names, function names, etc.

“EvilCoder: Automated Bug Insertion” [3] introduces a system for automatically finding potentially vulnerable source code locations and modifying the source code to be actually vulnerable. It detects bugs in C language through the method of invalidating security mechanisms and using security anti-patterns. It focuses on taint-style vulnerability which is the flow of insufficiently secure data between a user-controlled source and a sensitive sink. For inserting a bug, it is started by finding all sensitive sinks in the source code and tried to trace the data sources of a sensitive sink to a user-controlled source. This tool was evaluated on several open-source projects such as libpng and vs.ftpd, where between 22 and 158 unique connected source-sink pairs per project were found. Its main limitation is that it only supports a limited number of vulnerability types, which all belong to the class of taint-style vulnerabilities.

The paper named “Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code” [4] provides us with a dataset containing a total of 851,587 code snippets of C language. The scalability and performance issues in the Neural Language Model (NLM) of source code are due to both out-of-vocabulary and large vocabulary problems. In this paper, the authors present a study regarding the impact of modeling choices on resulting vocabulary and also introduce an NLM that can scale to a large corpus. The datasets used in this paper have been made public by the authors.

In the paper “Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks” [5] authors have used two key techniques: attention-based local context representation learning & network-based global context representation learning. In local context learning, source code is converted into AST using the Eclipse JDT package and AST node representations are learned using Word2Vec. After that, the output from Word2Vec is fed to the attention-based GRU approach which is used to capture sequential patterns from AST & convolutional-based approach to learn path vectors from a set of given node vectors. CNN is used to capture local coherence patterns from AST. After that, Multi Head attention is applied to learning unified representation which combines two path vectors into one code vector. In global context learning, a Program Dependence Graph (PDG) or Data Flow Graph (DFG) is generated for an entire project. Then, Node2Vec is used to convert large graphs into low-dimensional vectors without much information loss. These vectors are used to encode long paths in AST which were extracted in the local context computation step. After that, local context and global context vector representation are combined for the path, a representation for each model is obtained and CNN architecture is used to classify if the method is buggy or not. In detail, a convolutional layer is applied to the set of methods, then max-pooling & fully connected layer is applied followed by softmax for classification. The dataset of 4.973 million Java methods from 92 projects was extracted. Results show an improvement of 160% on F1 score when compared with state-of-the-art bug detection approaches, detected 48 true bugs in a list of top 100 reported bugs. Model outperforms four state-of-the-art bug detection baselines in detecting bugs: Deep Bugs, Bugram, NAR-miner, and Find Bugs by 107%, 37%, 155%, and 273% respectively, in terms of Precision, and by 69%, 25%, 92%, and

156% respectively, in terms of F1-score. Strengths includes that it considers the relationship among methods and paths from the perspective of the whole version of the project & learns to detect bugs rather than memorization. But it takes 4 minutes to finish detecting bugs which is 1 to 2 minutes more than other baselines. This approach doesn't work well on loops due to limitations of the static analysis approach and also doesn't work well on fixes needed to strings. In the future, authors plan to evaluate this approach to Python and C & also to investigate other techniques to efficiently model graph nodes and their surrounding structure.

In “DeepBugs: A Learning Approach to Name-based Bug Detection" [6], authors detect bugs through the name of variables, functions, etc. AST is generated through Acorn JavaScript parser which is then used for the automatic generation of the negative sample (bug) from the positive sample (correct code snippet). It detects three kinds of bugs: swapped function arguments, wrong binary operator, and wrong operand in binary operation with the help of name analysis, and a similar process can be used to train other machine learning models for the detection of new bugs. A simple feedforward NN with a dropout 0.2 in the input layer and the hidden layer is used for the classification of a code snippet to predict if it contains a specific type of bug or not. It breaks the dataset of 150,000 JavaScript files with 68.6 million lines of code into 100,000 files for training and 50,000 files for validation. It predicts swapped function arguments bug with an accuracy of 93.88% on random embedding and 94.70% on learned embedding, wrong binary operator bug with an accuracy of 89.15% on random embedding and 92.21% on learned embedding, and wrong binary operand bug with an accuracy of 84.79% on random embedding and 89.06% on learned embedding. Its drawback is that it relies on the reasonable name of identifiers that are in line with common practice so it may fail when developers choose to diverge from the variable naming norms. For example, it predicts the comparison of “value.length” with “is” identifier in the expression value.length !== is a bug because "is" is generally used for booleans.

In the paper “Deep Fix: Fixing Common C Language Errors by Deep Learning” [7] the authors present an end-to-end solution for inaccurate and misleading error messages from

compilers without the execution of the program. The authors used sequence to sequence neural networks with an attention model. Only about 27% of programs could be fixed completely and 19% of programs could be fixed partially. The evaluation was only available and only available to C programming language. Also, the model performance must be improved. And neural network architecture can’t effectively handle longer sequences.

In the paper "Semantic Code Repair using Neuro-Symbolic Transformation Networks" [8] authors introduced a novel neural network architecture that consists of SSC modules. The Share module performs a rich encoding of AST using a neural network and the Specialize module gives a score for every repair candidate. Inside it, Multilayer Perceptron (MLP) is used to perform the scoring of candidate fix and Pool Pointer module is used to predict token function names. The Complete module normalizes each repair candidate against one another via Local Norm or Global Norm. Around 19,000 repositories of python projects were used where bugs were synthetically injected. It classifies 4 classes of common bugs: VarReplace which is the use of incorrect local variable at a particular location, CompReplace which involves an incorrect comparison operator at a particular location, IsSwap where the operator is equal (=) is used instead of is not equal (!=) and vice versa, ClassMember which is missing a self accessor from the variable. The model was trained for 30 epochs. For 1 bug repair, it gives an F1 score of 85% and for 3 bug repairs, the F1 score was 81%. Model predicts the bug with an accuracy of 82%. The model outperforms humans with an accuracy of 60% when human accuracy is 37% after analyzing 2-6 minutes per snippet. Its weakness: no character level encoding and lexical similarity of code can't be modeled. Future plans include covering more bug types and exploring character level encoding and extending this model to do other tasks like natural language grammar correction, machine translation, program optimization, etc.

3. REQUIREMENT ANALYSIS

3.1 Hardware Requirements

3.1.1 GPU

GPU is used to perform mathematical calculations rapidly, especially for handling graphical and visual data. Kaggle's [14] backend GPU (i.e Nvidia's P100 or T4) of 15.9 GB RAM has been used to train the model. Similarly, 73.1 GB of disk space was accessed. It took around 45 minutes to train the swapped function arguments model for an epoch. It further took 1 hour and 40 minutes to train the wrong binary operator model for an epoch. Kaggle, a subsidiary of Google LLC, is an online community of data scientists and machine learning practitioners that allows users to find and publish data sets, and explore and build models in a web-based data-science environment. Kaggle's notebook has been used to write essential codes and train the model.

3.2 Software Requirements

3.2.1 Clang, LibClang and Graphviz

Clang is a C, C++, and Objective-C compiler developed by the LLVM Project. It is designed to be highly compatible with GCC, the most widely used compiler for Unix-like systems while offering improved diagnostics and warnings.

Libclang is a C interface to the Clang compiler. It provides an API for parsing C, C++, and Objective-C code into an abstract syntax tree (AST) and accessing information about symbols and source locations in the code. Clang and Libclang are used in these project's initial tasks to parse the code into AST thereby extracting the several required pieces of information.

Graphviz is a software package used to create visual representations of graphs and networks. For this project Graphviz is used to create a visual representation of the tree structure of an AST, making things easier in visualizing the required nodes of AST and for

the negative sample generation process in AST along with helping in the debugging process.

3.2.2 Pytorch

PyTorch is utilized as it is a popular open-source machine learning library and provides a high-level interface for creating and training deep learning models, and is known for its ease of use, dynamic computation graph construction, and fast training speeds. PyTorch also integrates well with other tools and libraries in the Python ecosystem, making it a popular choice for both researchers and practitioners in the field of AI. Tensorflow makes it easy for deployment but it is likely to bring memory issues in production and is difficult to update. Performance of pytorch is better than that of tensorflow so Pytorch is used.

3.2.3 Fast API

FastAPI provides an easy-to-use interface for defining the API endpoint, request and response data, and handling HTTP methods. FastAPI automatically generates documentation for the API using the OpenAPI specification, which makes it simple for other developers to understand and use the API. Additionally, FastAPI supports asynchronous programming and takes advantage of Python's async and await syntax to provide fast and efficient handling of multiple requests. All these features make FastAPI a great choice for building high-performance, scalable, and maintainable APIs.

3.2.4 React.js

React.js operates on a virtual DOM, which provides increased performance by allowing it to efficiently update only the necessary elements of a web page, rather than reloading the entire page. React.js also allows for a more efficient and scalable codebase through its use of components, which can be easily composed and reused throughout an application. Furthermore, React.js has a large and active community, which contributes to its continual development and improvement, and provides a wealth of resources and tools for developers. These features make React.js a popular choice for building complex and interactive web applications.

4. SYSTEM ARCHITECTURE AND METHODOLOGY

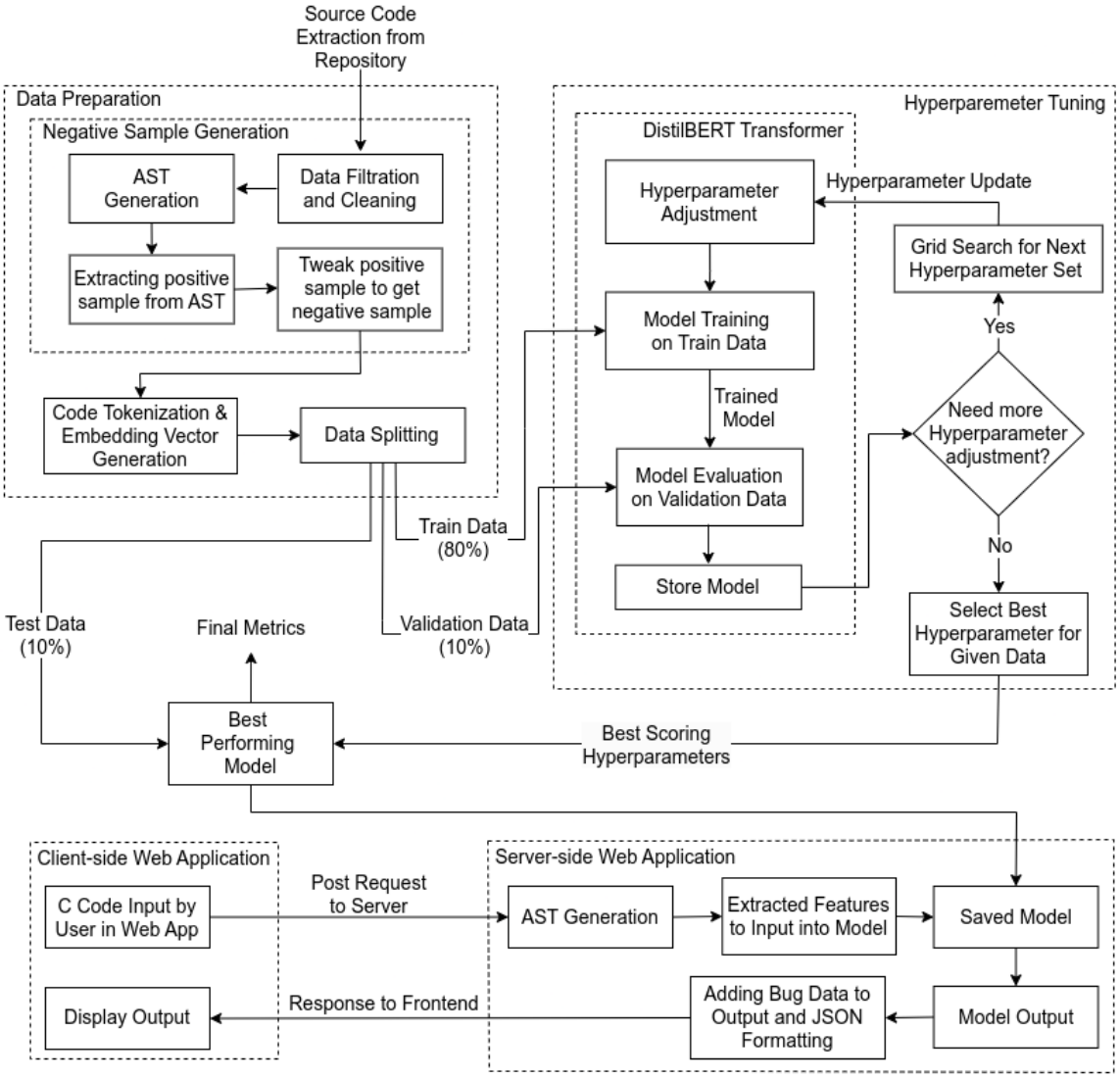

Figure 4-1: System Architecture

4.1 Data Preparation Block

4.1.1 Data Filtration and Cleaning

Around 720,000 code snippets of C language were taken from C Code Corpus [13] dataset by filtering only ".c" extensions from around 830,000 total files. The C Code Corpus dataset was created by the authors of "Big Code! = Big Vocabulary: Open-Vocabulary Models for Source Code" [4] and is mentioned in the same paper. The size of this dataset in compressed form is about 2.5 GB and about 12 GB when uncompressed.

4.1.2 AST Generation

An Abstract Syntax Tree (AST) is generated by parsing source code using libClang and constructing a tree-like structure that represents the source code's structure and meaning. This process involves breaking down the source code into smaller elements, such as tokens, and then grouping these elements into higher-level structures, such as expressions or statements. The resulting tree is a compact and easily processed representation of the source code that is used for various purposes, such as code analysis (positive samples) and code generation (negative samples).

4.1.3 Adjust Positive Sample to get Negative Sample

Each path in AST represents a sequence of nodes that correspond to a specific piece of code in the source code. The nodes along the path contain information about the code elements, such as their type, value, and location in the source code. As the AST provides features like function name, arguments, parameters and their datatype, the positive sample is then extracted. The positive sample is labeled '0'. After obtaining the positive sample, arguments are swapped and a new sample is generated referred to as the negative sample. Similarly, it is labeled '1' at the end of the expression. The same process applies for wrong binary operator where only the binary operators are swapped with possible binary operator swaps and thereby creating a negative sample.

4.1.4 Source Code Tokenization & Embedding Vector Generation

Tokenization in machine learning (ML) models refers to the process of converting a sequence of text into a sequence of tokens or words that can be easily processed and analyzed by the ML model. The code tokenization is done using the base version of CodeT5 tokenizer from Salesforce and was fine-tuned on C Code Corpus dataset of about 720,000 C language files to create a new tokenizer well-suited to C language dataset of our type. Subword level tokenization is used which includes BPE that merges most frequent characters into one. For example, if the dataset consists of: ‘get’, ‘getting’ and ‘gets’, then with byte pair encoding, it gives ‘get’ as one token and splits the other tokens as [‘get’, 'ting’] and [‘get’, ‘s’]. Hence, getting, gets and get will not be considered semantically different words in the vocabulary. Lemmatization could be used but long and infrequent words could not be recognized hence more words wouldn’t be considered into unknown [UNK] tokens. Each token from the code snippet is then converted into its corresponding embedding and then all the embeddings are grouped sequentially to generate the embedding vector.

4.1.5 Data Splitting

After the data is tokenized and its corresponding embedding vector is generated, the data is splitted into training and test sets. The model is trained on the training set and then evaluated on the validation set to see how well it generalizes to new, unseen data. This helps to avoid overfitting, which is when a model fits the training data too closely and performs poorly on new data. Hence, the data is split into 80% train data, 10% test data and 10% validation data.

4.2 Transformer

4.2.1 DistilBERT Transformer

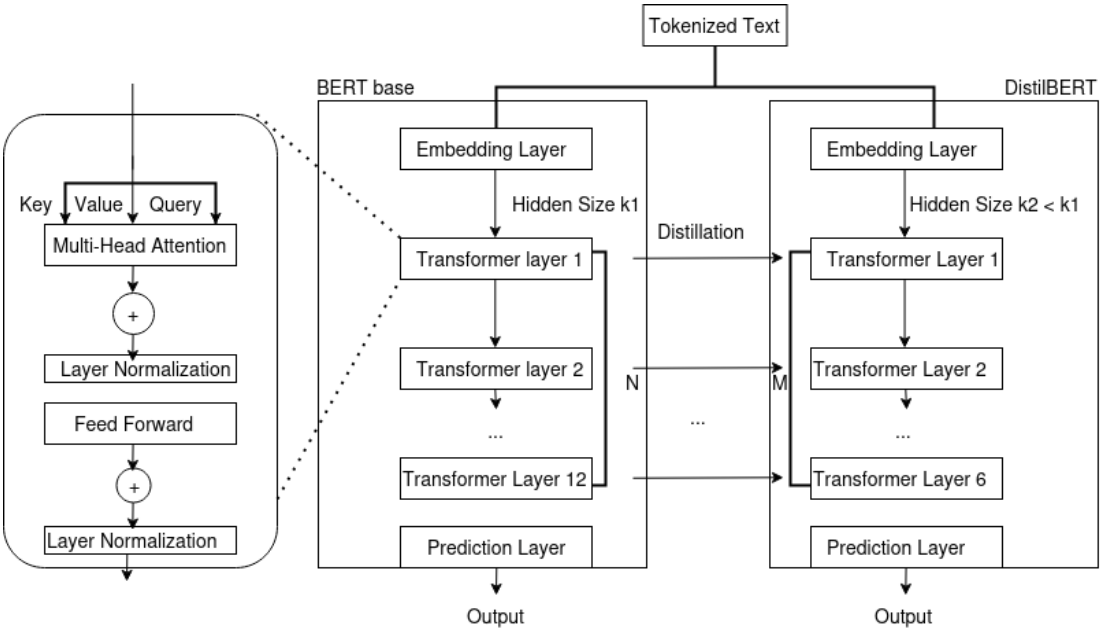

Figure 4-2: DistilBERT Architecture

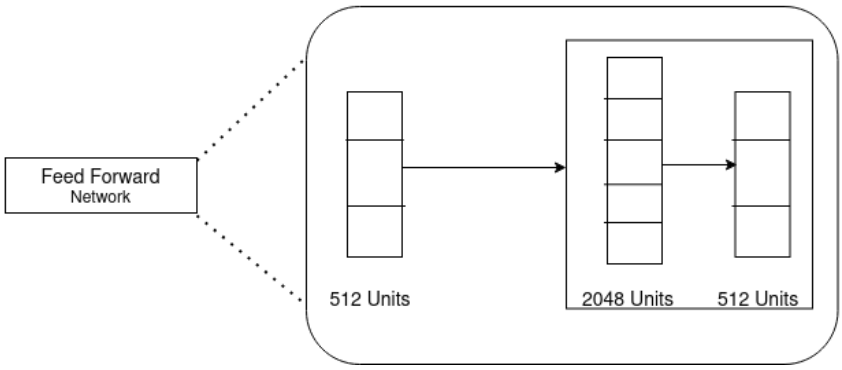

Figure 4-3: Feed Forward Network Architecture for BERT

DistilBERT is a smaller, lighter, and faster version of the popular BERT (Bidirectional Encoder Representations from Transformers) language model developed by Hugging Face. DistilBERT is trained to perform a wide range of natural language processing tasks, including sentiment analysis, named entity recognition, and question answering.

One key concept in DistilBERT is the use of the Transformer [10] architecture. The Transformer architecture was introduced in the original BERT paper and has since become

a cornerstone of NLP research. It allows for parallel processing of sequences and improved handling of long-range dependencies in language. In DistilBERT, the Transformer architecture is used to learn contextual relationships between words in a sentence, enabling it to perform advanced NLP tasks.

Another key concept in DistilBERT is the use of distillation, which is a process of transferring knowledge from a large and complex model (the teacher) to a smaller and simpler model (the student). In the case of DistilBERT, the teacher is the original BERT model, while the student is the smaller and faster DistilBERT model. By distilling the knowledge from the larger model, DistilBERT is able to maintain much of the performance of the original BERT model, while being faster and more memory efficient.

DistilBERT also uses attention mechanisms, which allow the model to selectively focus on different parts of the input to better understand the context. This allows DistilBERT to accurately handle tasks such as sentiment analysis and named entity recognition, where understanding the context is crucial.

Distillation in DistilBERT refers to the process of transfer learning, where a smaller, distilled version of a pre-trained language model is fine-tuned on a new task with a smaller dataset. The idea behind distillation is to transfer the knowledge learned by the large, pre-trained model to a smaller model, so that it can perform well on the target task using less computational resources.

During the distillation process, the outputs of the pre-trained model are treated as the "teacher" and used to train the smaller "student" model. The student model tries to mimic the behavior of the teacher model by minimizing the difference between its predictions and the teacher's predictions on the same input data. The final distilled model inherits the knowledge of the pre-trained model, but has a smaller architecture and requires less computational resources to fine-tune for new tasks.

In summary, DistilBERT is a powerful NLP model that leverages the Transformer architecture, distillation, and attention mechanisms to perform a wide range of NLP tasks.

Its smaller size and faster inference speed make it a popular choice for real-world applications where resources are limited.

4.2.2 CodeT5 Tokenizer

CodeT5 Tokenizer [9] is based on T5 transormer [15]. T5 transformer is closely based on originally proposed transformer [10]. In T5 transformer, an input sequence of tokens is mapped to sequence of input embeddings. The embeddings is then passed to the encoder. The encoder has a self-attention layer followed by a small feed-forward network. Layer normalization is applied to the input of each subcomponent. However, a simplified version of layer normalization where the activations are only rescaled and no additive bias is applied. After layer normalization, a residual skip connection adds each subcomponent's input to its output. Dropout is applied within the feed-forward network, on the skip connection, on the attention weights, and at the input and output of the entire stack (self-attention layer and a small feed-forward network).

4.3 Hyperparameter Tuning

Hyperparameters have a significant impact on the performance and convergence of the model. Hence, there must be careful selection of a particular hyperparameter to achieve the best results. For choosing the average split of cross-validation, different hyperparameters tuning methods like Grid Search CV, and Randomized Search CV can be used.

4.3.1 Grid Search

Grid search is a process of performing hyperparameter tuning in order to determine the optimal values of a given model. Adjusting hyperparameters can help improve accuracy. Hence, tuning must be done. For the tuning of hyperparameters using a basic approach, a certain time gap should be maintained until code is compiled after which hyperparameters are tuned again and run. Instead, Grid search helps to make the process easier by using GridSearchCV from sklearn library to automate the process and make it easier. Grid search tries all the combinations for the hyperparameters. For example, if 'a' number of hyperparameters and 'b' number of values are tested then grid search checks for 'a ×

b'combination. Hence it guarantees the optimal hyperparameter but it has an increased processing time than randomized search.

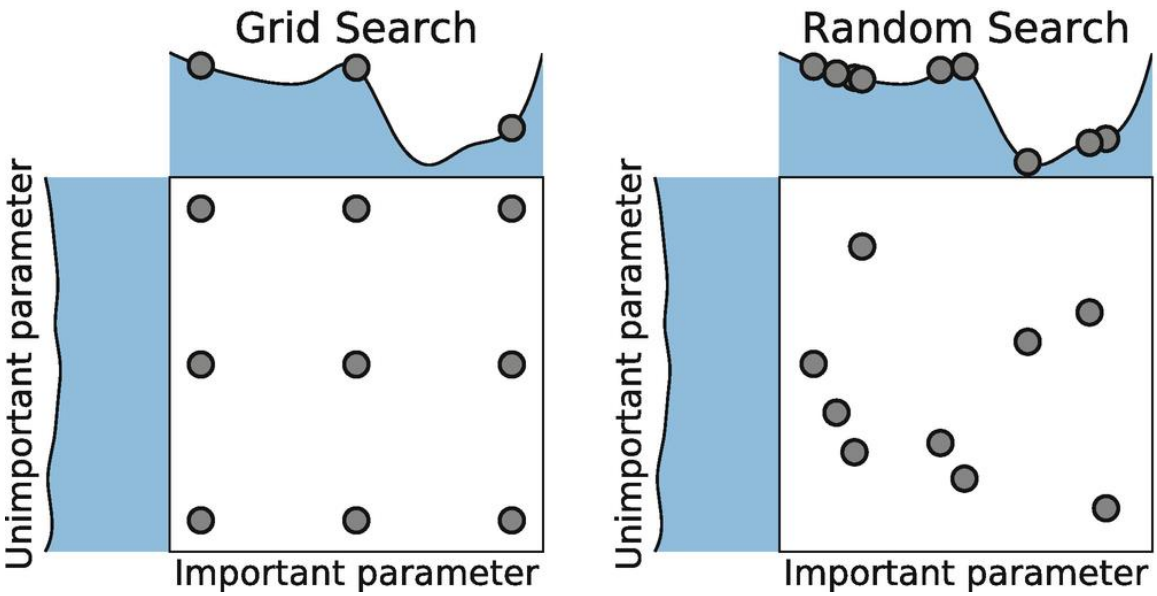

Figure 4-4: Grid Search vs. Randomized Search

4.3.2 Randomized Search

For handling big datasets, the Grid Search underperforms because if all combinations of hyperparameters are tested on big datasets, it will take a really long time and it will be tedious. In randomized search, all the hyperparameters are not considered important. Randomized search offers less processing time than Grid Search. Randomized fixed parameter settings are sampled. Randomized Search does not always give the optimal combination of hyperparameters. Hence there is always a trade-off between optimal hyperparameters and decreased processing time.

4.4 Choosing the Best-Performing Hyperparameter

Choosing a hyperparameter is the most important step in the training model. Once All the hyperparameters are adjusted, the best performing hyperparameter is chosen. The best performing hyperparameter gives more accuracy for the model. To do this, the model is trained using different values of the hyperparameters. The accuracy of the model is then evaluated using the validation set. Hyperparameter which results in highest F1 Score is

chosen as best-performing hyperparameter. Then, the metrics of the best hyperparameter in the current fold is stored. The metrics includes accuracy, loss, F1 Score, etc that are used to evaluate performance of the model. This information can be used to compare performance of different hyperparameters and the best hyperparameter that is chosen results in the highest F1 Score of the model.

4.5 Best Performing Model

After the data has been split into the training sets and validation sets there will be the next step where best scoring metrics is chosen for the model. In this step, evaluation performance of the model is carried out and the best set of hyperparameters are chosen which gives the best result. Evaluation process involves calculating accuracy, precision, recall and F1 score on the detection process. After the hyperparameter has been adjusted, the final model can be trained to make use of unseen data. After that the model is tested using 10% test data taken from the Data splitting process. These project's goal is to determine how well our model can be generalized into the new unseen data. After the test process the overall performance of the model is generated. After that model is chosen which gives the best performance metric (in this case best F1 Score).

4.6 Web Application

To develop a web application that utilizes a machine learning model to detect runtime errors in code, you can follow these steps: As for development framework because python is used to train the model, frameworks such as Flask and Fast API for web development framework are used. After that a user interface is created where a text box will be kept and other interactive elements are kept which allows users to enter code and interact with the application. Then, the web app is connected with the pre-trained ML model with the use of API to connect the model with our application. Then model analyzes code snippets entered by user to check if there are runtime errors. It returns analysis to the user displaying a message which indicates whether code is error-free or if there is an error that needs a fix.

5. IMPLEMENTATION DETAILS

5.1 Data Collection & Preparation

C code corpus is the dataset where the source code is collected written in C programming language.C code corpus contains around 2.5 GB of zipped C programming files which is around 12 GB when extracted. It contains around 720,000 codes of C programming language. It is the C files where preprocessing is not done.

5.2 Data Preprocessing

5.2.1 Dealing with Outliers

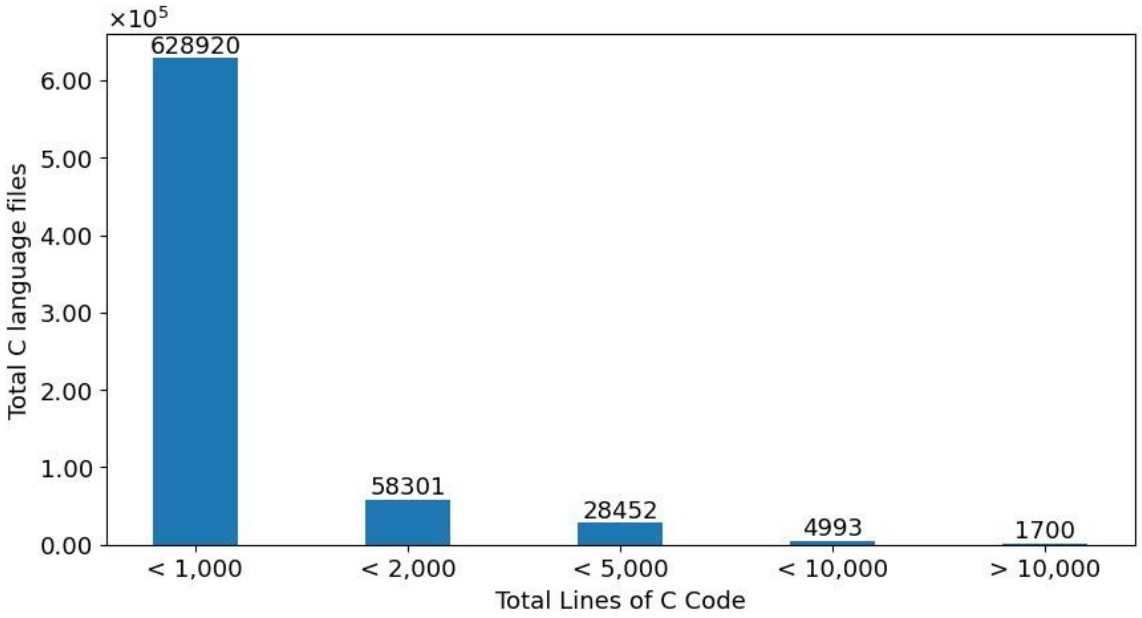

Figure 5-1: Outliers and Non-Outliers Visualization

Upon examining the figure presented above, it becomes evident that the majority of code snippets consist of less than one thousand lines of code. In light of this observation, it is deemed unethical and superfluous to generate Abstract Syntax Trees (ASTs) for the code snippets which had greater than 10k lines of code. As a result, any code snippets containing over ten thousand lines of code, numbering at 1700, are excluded. Only the code snippets with less than 10k lines of code are taken.

5.2.2 Representing Code as AST Paths

Objective C source code is parsed using a library provided by the Clang compiler i.e Libclang and generates an Abstract Syntax Tree (AST) from it. An Abstract Syntax Tree (AST) is a tree-like structure that represents the structure and meaning of source code. The process of generating an AST using libclang involves various steps. The first step in generating an AST using libclang is to initialize a Translation Unit (TU), a unit of source code that is parsed by the Clang compiler. Once the Translation Unit has been initialized, liblang's parsing functions can be used to parse the source code and build an AST. Each path in AST represents a sequence of nodes that correspond to a specific piece of code in the source code. The nodes along the path contain information about the code elements, such as their type, value, and location in the source as it can be traversed from its root cursor to children using libclang's API that allows us to access the nodes of the AST and their properties.

Here's how the general AST looks like along with several nodes:

Table 5-1: C Code with AST Nodes

Line no.	Code	AST Node
1	int add_abs (int a, int b){	Function Declaration Node with the start of CompoundStmt Node
2	int result;	VarDecl Node
3	if (a + b < 0) {	Ifstmt Node and BinaryOperator Node with the start of another CompoundStmt Node
4	result = - (a + b);	UnaryOperator Node
5	} else {	End of CompoundStmt Node
6	result = a + b;	BinaryOperator Node
7	}	
8	return result;}	

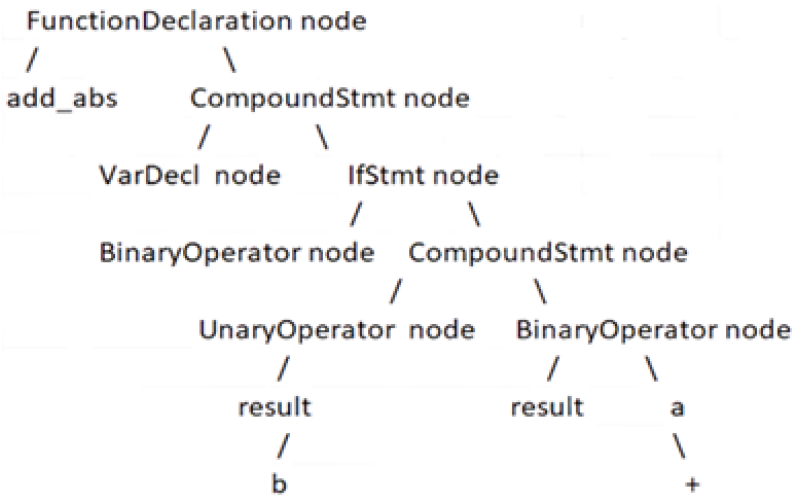

Figure 5-2: A General Structure of AST with Nodes

The general structure of an Abstract Syntax Tree (AST) for a C program typically includes the following types of nodes as referenced to the above Figure 5-2:

1. Translation Unit: The root node of the AST, representing the entire source code file.
2. Function Declaration nodes: Represent the declaration of a function, including the function signature and the body. Example: **FunctionDeclaration** node with 'add_abs'
3. Statement nodes: Represent individual statements in the function body, such as variable assignments or control flow statements. Example: **CompoundStmt** node
4. Expression nodes: Represent individual expressions in the source code, such as arithmetic operations or function calls.
5. Declaration nodes: Represent declarations within a function, such as local variable declarations. Example: **VarDecl** node where a variable 'result' is declared.
6. Control flow nodes: Represent control flow statements in the source code, such as loops or conditional statements. Example: **IfStmt** node where 'if' statement is declared.

7. Binary Operation nodes: Represent binary operations such as addition, subtraction, multiplication, etc. Example: **BinaryOperator** node which represents the expression ‘a + b’.
8. Literal nodes: Represent literals, such as integers, floating-point numbers, or string literals. Example: **IntegerLiteral** node which showcases any integer values assigned.

Now, here’s how the AST is meticulously used in the corpus of C codes for further proceedings. Here’s an example of AST generation of a c source code from code corpus:

Table 5-2: C Program from Code Corpus

Line no.	Code	Comments
1	#include <stdio.h>	
2	void setDimension(int x, int y){	Function declaration with parameters ‘x’ and ‘y’
3	printf("Width: %d, Height: %d\n", x, y);	Function body
4	}	
5	int main()	Main function
6	{	
7	int height = 7, width = 10;	Variable declaration
8	setDimension(width, height);	Function call with possible bug
9	return 0;	
10	}	

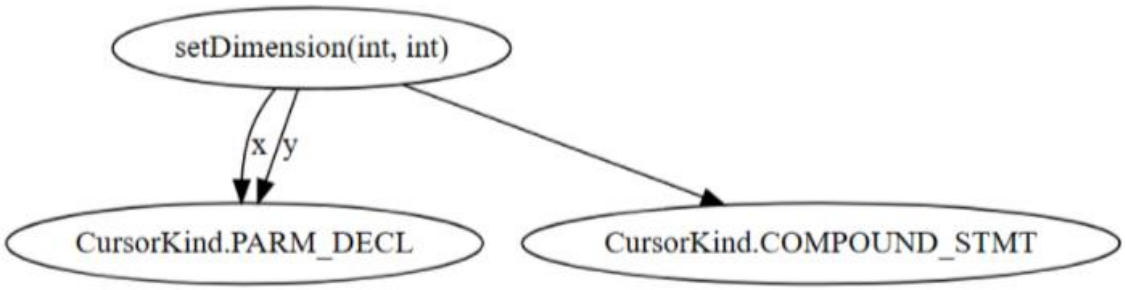

Figure 5-3: Graphical AST of setDimension Function Definition

```
FunctionDecl 0x20c0fbe13b8 <C:/msys64/home/ASUS/swap.c:3:1, line:6:1> line:3:6 used setDimension 'void (int, int)'
|-ParmVarDecl 0x20c0fbe1268 <col:19, col:23> col:23 used x 'int'
|-ParmVarDecl 0x20c0fbe12e8 <col:26, col:30> col:30 used y 'int'
`-CompoundStmt 0x20c0fbe1648 <line:4:1, line:6:1>
  `-CallExpr 0x20c0fbe15b0 <line:5:2, col:40> 'int'
    |-ImplicitCastExpr 0x20c0fbe1598 <col:2> 'int (*)(const char *, ...) __attribute__((cdecl))' <FunctionToPointer
    | `-DeclRefExpr 0x20c0fbe1468 <col:2> 'int (const char *, ...) __attribute__((cdecl))':'int (const char *, ...)
char *, ...)'
    |-ImplicitCastExpr 0x20c0fbe1600 <col:9> 'const char *' <NoOp>
    | `-ImplicitCastExpr 0x20c0fbe15e8 <col:9> 'char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x20c0fbe14c8 <col:9> 'char[23]' lvalue "Width: %d, Height: %d\n"
    |-ImplicitCastExpr 0x20c0fbe1618 <col:36> 'int' <LValueToRValue>
    | `-DeclRefExpr 0x20c0fbe14f8 <col:36> 'int' lvalue ParmVar 0x20c0fbe1268 'x' 'int'
    `-ImplicitCastExpr 0x20c0fbe1630 <col:39> 'int' <LValueToRValue>
      `-DeclRefExpr 0x20c0fbe1518 <col:39> 'int' lvalue ParmVar 0x20c0fbe12e8 'y' 'int'
```

Figure 5-4: AST of setDimension Function Definition

The aforementioned Figure 5-4 shows the simple representation of AST performed in command prompt for the function definition i.e. setDimension of C source code shown in Table 5-2 respectively. Libclang is used to parse the source code and generate AST from it. LibClang API allows traversing several nodes through the AST structures.The Clang cursor API explores the AST starting from the root node and traversing through its children nodes until the 'setDimension' node with the attribute **FunctionDec1**, function with two parameters, is encountered which is identified by cursor.kind property of libclang. **0x20c0fbe13b8** is the cursor object that contains information about a corresponding single node which is the 'setDimension' function in the AST. It provides features such as its type, location, and properties. In this case, its type is **void** followed by its two parameters of type **int** as given by **void (int, int)**. As for the location, it gives **line:6:1> line:3:6** which refers to the area the corresponding node is placed under. The cursor is then moved to its children as shown by indentations of the next line as it traverses down to the leaf nodes. **ParmVarDecl1** represents a parameter 'x' in a function declaration, specifically the first

parameter and **ParmVarDec2** represents a parameter i.e 'y' in a function declaration, specifically the second parameter. **0x20c0fbe1268** is their corresponding cursor object with its own features such as location and types. The location of 'x' is given by **<col:19, col:23> col:23** column index which starts at column no. 19 and ends at column no. 23. As for 'y', the location is **<col:26, col:30> col:30**. As it traverses along, it encounters **CompoundStmt** that simply refers to the body of the function. Its location is given by **<line: 4:1, line:6:1 >** which marks the start and end of the curly bracket ({}) as shown in Figure 5-4.

A graphical representation of the corresponding AST is shown in Figure 5-3 using the GraphViz library. For the same purpose, the Clang cursor API explores the AST starting from the root node and traversing through its children nodes until the 'setDimension' node, function with two arguments, is encountered. The 'setDimension' node becomes the root node of the cursor. It has two child nodes. The left node is the parameter declaration written as **CursorKind.PARM_DECL** where 'x' and 'y' are the declared parameters. On the right, the node is the **CursorKind.COMPOUND_STMT,** which is the compound statement which includes the content/body parts of the function inside the curly bracket.

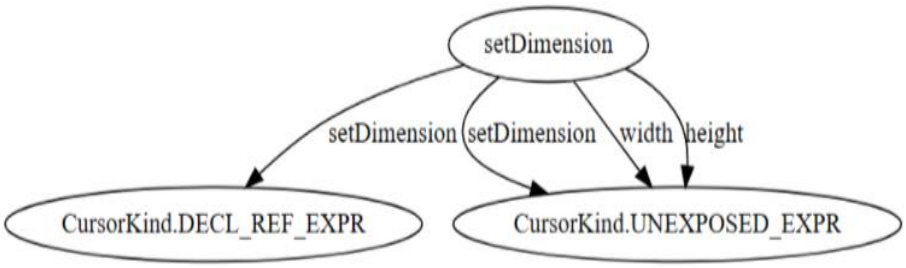

Figure 5-5: Graphical AST of setDimension Function call

```
-CallExpr 0x1ae9f91af50 line:11:2 col:28 'void'
 |-ImplicitCastExpr 0x1ae9f91af38 col:2 'void (*)(int, int)' FunctionToPointerDecay
 | `-DeclRefExpr 0x1ae9f91aea8 col:2 'void (int, int)' Function 0x1ae9f9553f8 'setDimension' 'void (int, int)'
 |-ImplicitCastExpr 0x1ae9f91af80 col:15 'int' LValueToRValue
 | `-DeclRefExpr 0x1ae9f91aec8 col:15 'int' lvalue Var 0x1ae9f9557a8 'height' 'int'
 `-ImplicitCastExpr 0x1ae9f91af98 col:23 'int' LValueToRValue
   `-DeclRefExpr 0x1ae9f91aee8 col:23 'int' lvalue Var 0x1ae9f91adf0 'width' 'int'
```

Figure 5-6: AST of setDimension Function Call with Two Arguments

The aforementioned Figure 5-6 shows the simple representation of AST performed in command prompt for the function call inside the main function i.e setDimension respectively. The Clang cursor API explores the AST starting from the root node and traversing through its children nodes until the ‘setDimension’ node with the attribute **callExpr**, the function with two arguments, is encountered which is identified by cursor.kind property of libclang. The information is stored in the object **0x1ae9f91af50 about** its location, type and propertiesThe format of the AST is a tree-like structure, with indentations indicating the parent-child relationship between nodes. The first line represents a "CallExpr" cursor, which is a node in the AST that represents a function call. It has a type of **void** and is located starting from line 11 ranging upto column 28 of the same line number as displayed by **line: 11:2** and **col 28**. The next lines, with indentations, represent child nodes of the "CallExpr" cursor. For example, the **ImplicitCastExpr** cursor at line 2 represents an implicit type cast operation used to represent conversions between data types, and it is a child of the "CallExpr " cursor. **FunctionToPointerDecay** refers to the implicit conversion of a function name to a pointer to the function. It allows the function call i.e setDimension to be passed as an argument to another function definition ‘setDimension’ which is located outside of the main function. As it further traverses, it reaches to the node of the children which are the two arguments, ‘height’ and ‘width’. **LValueToRValue** refers to passing lvalue arguments like ‘height’ and ‘width’ to functions. By converting lvalues to rvalues, the values stored in memory locations can be accessed and used as arguments, rather than local copies of the values.

5.2.3 AST for Wrong Binary Operator

Table 5-3: C Program for Binary Operator

Line no.	Code	Comments
1	#include <stdio.h>	
2	int main()	main function
3	{	
4	int a = 5, b = 7;	variable declaration i.e ‘a’ and ‘b’
5	float d = a * b;	binary operation with binary operator ‘*’
6	printf("Result is: %d", d);	
7	return 0;	
8	}	

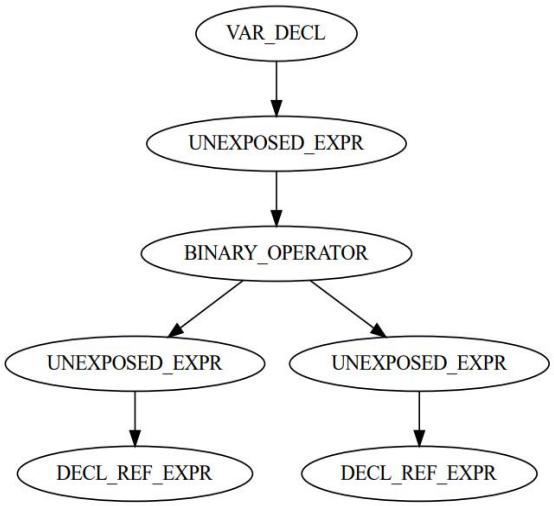

Figure 5-7: Graphical AST for Wrong Binary Operator

In the aforementioned figure, **VAR_DECL** node is the grandparent node and **UNEXPOSED_EXPR** is the parent node of the corresponding **BINARY_OPERATOR** which is of utmost importance in generating the negative sample for wrong binary operator and denoted by ‘*’ in the line number 5 of the above Table 5-3. The

‘BINARY_OPERATOR’ has two children denoted by another ‘UNEXPOSED_EXPR’ which further has a child of their own that is **DECL_REF_EXPR**. The ‘DECL_REF_EXPR’ node in Figure 5-7 provides the information about the operands. The operands are ‘a’ and ‘b’ as shown by left and right node of ‘DECL_REF_EXPR’ respectively.

5.2.4 Traversal on AST Tree

AST is constructed using a pre-order traversal. Pre-order traversal is a type of tree traversal that visits the current node before its children. In a pre-order traversal, the current node is processed first, then the left subtree is traversed recursively, and finally, the right subtree is traversed recursively. For example, in the Figure 5-4, the FunctionDeclaration node (FunctionDecl1) is visited first i.e., ‘setDimension’ from the translation unit which serves as the root cursor. Then, its left child ‘ParmVarDec1’ is traversed until both the parameters ‘x’ and ‘y’ are extracted. The attention is then moved to the right node which consists of the compoundStmt node. The traversal operation is then stopped because the arguments have already been extracted along with the function name, eventually extracting every needed feature for the generation of negative samples.

5.3 Generation of Negative Sample

5.3.1 Function Arguments Swap

Table 5-4: Negative Sample (1) and Positive Sample (0)

Function_name	**arg1**	**arg2**	**arg_type**	**param 1**	**param 2**	**labels**	**Remarks**
setDimension	height	width	int	x	y	0	Positive sample
setDimension	width	height	int	x	y	1	Negative sample

The AST represents the structure of the source code, including the relationships between different constructs such as classes, functions, variables, etc. This information has been used to understand the architecture and organization of source codes. In this case, it is the functions with two or more arguments that the project is working on. For example, the features like function name (setDimension), arguments (height, width), parameters (x, y) and their datatype (int) in the above illustrated Table 5-4 are extracted from the generated AST. The sample extracted is a positive sample which is labeled '0'. Positive samples are the foundation for the generation of negative samples. From the extracted positive sample, the arguments of the function are swapped and a new sample is created referred to as a negative sample. The negative sample is then labeled '1'.

For example, from the extracted information, the training data generator creates a negative sample for each positive sample as referenced to Michael Pradel's DeepBugs [6]:

$$X_{\text{pos}} = (\text{function\_name}, \text{arg1}, \text{arg2}, \text{arg\_data\_type}, \text{param1}, \text{param2}) \quad (5\text{-}1)$$

$$X_{\text{neg}} = (\text{function\_name}, \text{arg2}, \text{arg1}, \text{arg\_data\_type}, \text{param1}, \text{param2}) \quad (5\text{-}2)$$

That is, to create the negative example for swapped function arguments bug, simply swap the arguments are swapped w.r.t. the order in the original code. Eventually, both these positive and negative samples are combined to create a dataset.

5.3.2 Wrong Binary Operator

Table 5-5: Binary Operators with Corresponding Possible Swaps

Operator	Possible Swaps
‘==’	['>', '<', '!=', '>=', '<=', '=']
'<'	['>', '==', '!=', '>=', '<=', '<<', '^', '>>']
'+'	['-', '*', '/', '%', '+=']
'*'	['+', '-', '/', '%', '*=']
'-'	['+', '*', '/', '%', '-=']
'!=	['>', '<', '==', '>=', '<=', '=']
'>'	['==', '<', '!=', '>=', '<=', '<<', '^', '>>']
'&':	['|', '^', '<<', '>>', '&&', '||']
'>='	['>', '<', '!=', '==', '<=', '<<', '^', '>>']
'/'	['+', '-', '*', '%', '/=']
‘<='	['>', '<', '!=', '>=', '==', '<<', '^', '>>']
'&&'	['||', '&', '|']
'<<'	['|', '&', '^', '>>', '&&', '||', '<=', '<']
'>>'	['|', '&', '^', '<<', '&&', '||', '>=', '>']
'||'	['&&', '|', '&', '/']
'%'	['+', '-', '*', '/', '<', '>', '>=', '<=', '!=', '%=']
'|'	['&', '^', '<<', '>>', '&&', '||', '/']
'^'	['|', '&', '<<', '>>', '&&', '||', '<', '>']

For the generation of wrong binary operator’s dataset, the above table gives the information about the binary operators and their possible swaps. Using the above table, the operator from the positive sample is then swapped with one of the corresponding possible operators as represented in array. For instance, Less than (<) binary operator in the positive sample can be replaced with Greater than (>) binary operator and the negative sample can then be generated as shown in the below table. The label ‘0’ signifies the positive sample and the label ‘1’ signifies the negative sample.

Table 5-6: Negative Sample Generation of Wrong Binary Operators

left	operator	right	type_left	type_right	parent	grandparent	labels
i	<	linkCount1	int	int	FOR_STMT	COMPOUND_STMT	0
i	<=	linkCount1	int	int	FOR_STMT	COMPOUND_STMT	1

5.4 Dataset Exploration

5.4.1 Function Arguments Swap Bug Dataset

Table 5-7: Dataset Overview of Function Arguments Swap

function_name	arg1	arg2	arg_type	param1	param2	labels
strcmp	bin	val_format	const char *			1
pow	dc	ldc	double	__x	__y	1
strcmp	no	option	const char *			1
calloc	xin	sizeof(double)	unsigned long			0
printf	\n[%s]\n	architecture	const char *			0
enable_irq	1	num-32	int			1
check_nv	0xfe66	pa0b2	char *	name	value	1
remember	domain	node	char *	node	domain	1

Only certain features like function name, name of arguments, type of argument, name of parameters are extracted from the generated AST using LibClang library. Positive samples extracted from AST are combined with the negative sample adding up to the total of 486534 samples.

5.4.2 Handling Duplicate Data

Table 5-8: Duplicate Data

function_name	arg1	arg2	arg_type	param1	param2	labels
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0
av_sat_add32_c	a	av_sat_add32_c	int	a	b	0

In the aforementioned table, ‘av_sat_add32_c’ as function_name, ‘a’ as arg1, ‘av_sat_add32_c’ as arg2, ‘int as argument type’ ‘a’ as param1, ‘b’ as param2 and ‘0’ as labels is the information that is stored in multiple locations. Duplicated data can affect the quality of the dataset and can skew the results of any analysis performed on the dataset. For instance, the above record is duplicated, it will lead to an over-representation of that record, which may lead to incorrect conclusions about the data. Duplicate data can even lead to inaccurate results in analysis and modeling. Therefore, it is important to remove duplicate data to ensure that the data is accurate and reliable. Data Duplicates were removed by converting the dataset into pandas dataframe and removing duplicates by calling pandas inbuilt delete_duplicates method.

5.4.3 Wrong Binary Operator Dataset

Table 5-9: Dataset Overview of Wrong Binary Operator Dataset

left	operator	right	type_left	type_right	parent	grandparent	labels
low	&&	high	int	int	PAREN_EXPR	UNARY_OPERATOR	1
tlp	%	2	unsigned char *	int	PAREN_EXPR	UNARY_OPERATOR	1
o	==	0	int	int	IF_STMT	IF_STMT	0
p	-=	page	char *	char *	PAREN_EXPR	COMPOUND_ASSIGNMENT_OPERATOR	1
rd	-	1	int	int	BINARY_OPERATOR	FOR_STMT	0
c	==	'A'	int	int	IF_STMT	IF_STMT	0
attr	<<	8	int	int	UNEXPOSED_EXPR	VAR_DECL	0

The Wrong binary operator bug dataset of total 1111986 samples comprised of both positive and negative samples can further be split into following three categories with their approximation in composition:

Table 5-10: Wrong Binary Operator Dataset Composition

Dataset	Total Split
Train dataset	889590 [80% of total]
Validation dataset	111198 [10% of total]
Test dataset	111198 [10% of total]

5.4.4 Binary Operators in Dataset

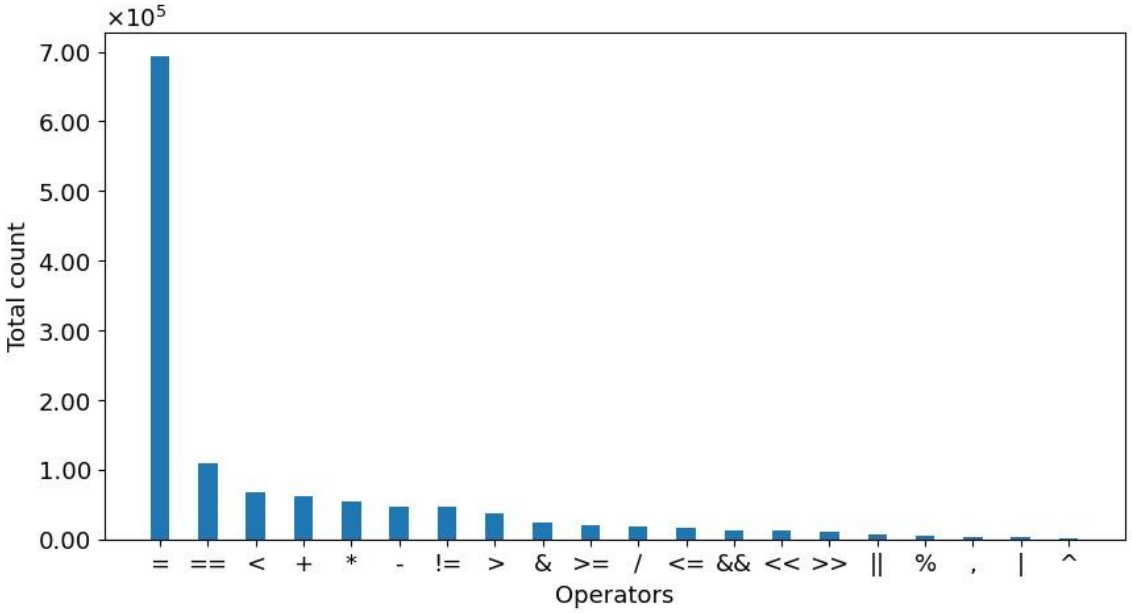

Figure 5-8: Bar Plot of Operator Count

The dataset contains distinct binary operators. Even within the several varieties of binary operators, the dataset contains some binary operators in huge numbers while some relatively sparse. The dataset is filled with the assignment operator '=' more than any other binary operators. It is logical as to why the assignment operator '=' is in abundance because of its huge importance and frequent uses. Every programming language undoubtedly contains the assignment operator '=' in excess amount as it's the most basic operator. Apart from this operator, all the other operators are close to each other in the quantity. For instance, equals-to operator (==) and less-than operator (<) are close to each other in terms of the frequency of count and the very same thing can be said with the other binary operators.

5.5 Missing Data Visualization

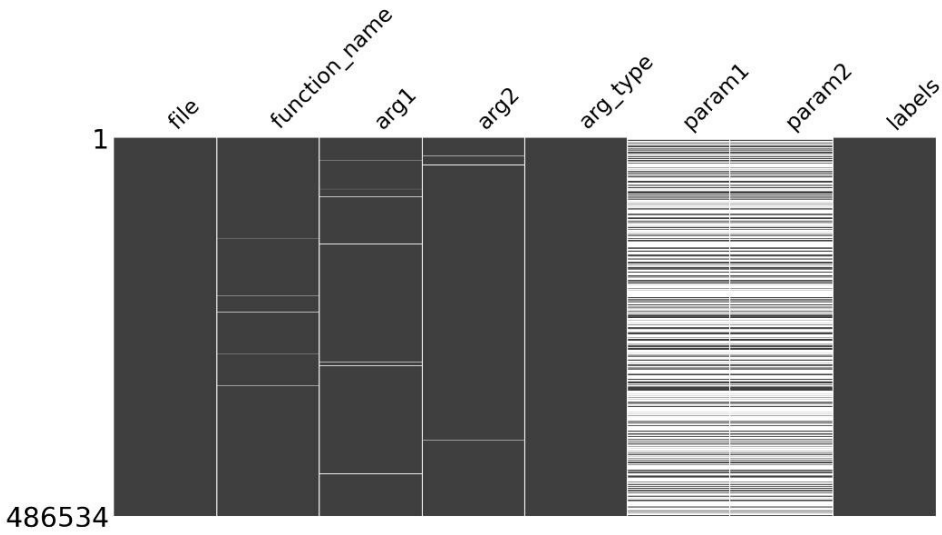

Figure 5-9: Visualizing Missing Data in a Grid

Missingno library has been used to generate a matrix plot that visualizes the presence or absence of missing values in the DataFrame. The plot shows the data as a grid, with each row representing a different sample and each column representing a different feature. Missing values are represented as white spaces in the grid, making it easy to spot patterns and trends in the missing data.

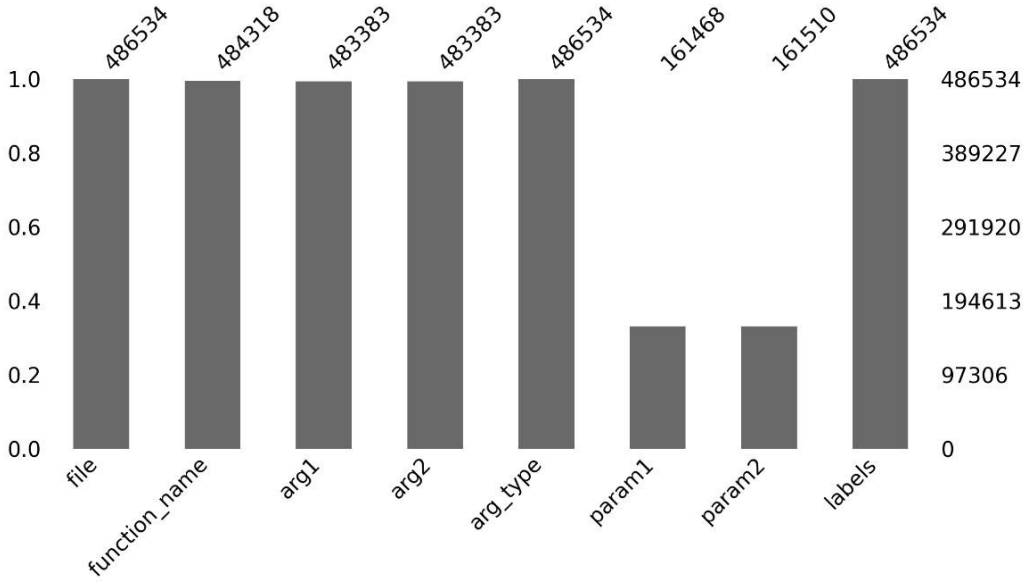

Figure 5-10: Bar Diagram Of Visualized Missing Data

The bar chart generated using Missingno library displays the columns of the data as bars and the height of each bar indicates the number of missing values in that column. In the bar chart visualization, columns with a high number of missing values (param1 & param2) are shown with smaller bars, while columns with a lower number of missing values (arg_type) are shown with taller bars.

In the total of 486534 samples, Missingno library found following missing data:

Table 5-11: Number of Missing Data

Features	Missing Data
function_name	2216
arg1	3151
arg2	3151
arg_type	0
param1	325066
param2	325024
labels	0

5.5.1 Handling Missing Data

Table 5-12: Missing Data

function_name	arg1	arg2	arg_type	param1	param2	labels
strcmp	bin	val_format	const char *	NaN	NaN	1
strcmp	no	option	const char *	NaN	NaN	1
printf	\n[%s]\n	architecture	const char *	NaN	NaN	0
enable_irq	1	num-32	int	NaN	NaN	1
outb	0x40b	0x30	int	NaN	NaN	1

Table 5-13: Replace Missing Data with ‘[UNK] ’ Token

function_name	arg1	arg2	arg_type	param1	param2	labels
strcmp	bin	val_format	const char *	[UNK]	[UNK]	1
strcmp	no	option	const char *	[UNK]	[UNK]	1
printf	\n[%s]\n	architecture	const char *	[UNK]	[UNK]	0
enable_irq	1	num-32	int	[UNK]	[UNK]	1
outb	0x40b	0x30	int	[UNK]	[UNK]	1

Dealing with missing or incomplete data is an important step in data preprocessing so depending on the type and extent of missing data, various methods were used to handle missing data, such as removing the missing data, replacing missing values with the unknown token ‘[UNK]’to fill in the missing values.

5.5.2 Reducing Size of Data by Removing Irrelevant Features

Table 5-14: Removing Irrelevant Data

function_ name	arg1	arg2	arg_type	param1	param2	labels	full_text
stricmp	.pngds	sExt	char *	[UNK]	[UNK]	1	stricmp .pngds sExt char * [UNK] [UNK]
strcasecmp	p	html	const char *	[UNK]	[UNK]	0	strcasecmp p html const char * [UNK] [UNK]
matches	*argv	reordering	char *	[UNK]	[UNK]	0	matches *argv reordering char * [UNK] [UNK]
strcmp	TC_5	sTestCase	const char *	__s1	__s2	1	strcmp TC_5 sTestCase const char * __s1 __s2

Table 5-15: Finalized Data

labels	full_text
1	stricmp .pngds sExt char * [UNK] [UNK]
0	strcasecmp p html const char * [UNK] [UNK]
0	matches *argv reordering char * [UNK] [UNK]
1	strcmp TC_5 sTestCase const char * __s1 __s2

Because High-dimensional data are difficult to analyze, and it can also be computationally expensive to process. Hence the size of the data was reduced by removing irrelevant features or columns like function_name, arg1, arg2, arge_type, param1 and param2. Instead, a single feature or column of full_text incorporates all the aforementioned feature's content as shown in the above Table 5-15.

5.6 Source Code Tokenization

The source code tokenization is done using the base version of CodeT5 tokenizer from Salesforce and was fine-tuned on C Code Corpus dataset of about 720,000 C language files to create a new tokenizer well-suited to C language dataset of our type. Subword level tokenization is used which includes BPE that merges most frequent characters into one. For example, if the dataset consists of: 'get', 'getting' and 'gets', then with byte pair encoding, it gives 'get' as one token and splits the other tokens as ['get', 'ting'] and ['get', 's']. Hence, getting, gets and get will not be considered semantically different words in the vocabulary. Lemmatization could be used but long and infrequent words could not be recognized hence more words wouldn't be considered into unknown [UNK] tokens. Each token from the code snippet is then converted into its corresponding embedding and then all the embeddings are grouped sequentially to generate the embedding vector.

Table 5-16: Generated Tokens from Default codeT5 Tokenizer

Code	**Extracted Function**	**Token**

#include <stdio.h>
void setDimension(int x, int y){
printf("Width: %d, Height: %d\n", x, y);}
int main(){
int height = 7, width = 10;
setDimension(width, height);
return 0;
}
	setDimension width height int x y	['set',
		'Dimension',
		'Ġwidth',
		'Ġheight',
		'Ġint',
		'Ġx',
		'Ġy']

Table 5-16 shows the tokens generated using default CodeT5 tokenizer from Salesforce. It doesn't give the desired outcome. This could be costly in the performance of the model. So, the better way is to fine tune the default CodeT5 tokenizer in such a way that overcomes it.

Table 5-17: Generated Tokens from Fine Tuned codeT5 tokenizer

Code	Extracted Function	Token

#include <stdio.h>
void setDimension(int x, int y){
printf("Width: %d, Height: %d\n", x, y);}
int main(){
int height = 7, width = 10;
setDimension(width, height);
return 0;
}
	setDimension width height int x y	[‘Start’,'set', 'Dim', 'ension', 'Ġwidth', 'Ġheight’, ‘Ġint', 'Ġx', 'Ġy',‘End’]

Table 5-17 shows the tokens generated using fine-tuned codeT5 tokenizer. It gives the desired outcome on most of the conditions.

5.7 Tokens to Input Ids

Table 5-18: Tokens and corresponding Input_ids

Tokens	**Input_ids**
[‘<s>’, ‘set’, ‘Dim’, ‘ension’, ‘Ġwidth’, ‘Ġheight’, ‘Ġint’, ‘Ġx’, ‘Ġy’, ‘</s>’]	[1, 492, 13570, 3722, 2891, 3808, 470, 695, 813, 2]

Tokens were generated of the code snippet of Table 5-17. ‘Input_ids’ in Table 5-18 are the numerical representations (integers) of the corresponding tokens in the input text. They are used as indices into the fine-tuned tokenizer’s vocabulary to retrieve the corresponding token embeddings, which are the vectors that capture the semantic information of the tokens. ‘Input_ids’ here provides a numerical representation of tokens in text prior to converting it into vectors. The ‘input_ids’ allow the Transformer models (DistilBERT) to handle input text data efficiently, as they reduce the size of the data and make it easy to process in a neural network. Moreover, they make it possible to fine-tune pre-trained models on new data, as the tokenization step is already handled by the tokenizer and the ‘input_ids’ provide a fixed-size input that can be used as input to the model.

5.8 Input Embedding

The input ids of a particular source code in Table 5-17 is obtained. Before training any transformer or ML model, the text data must be converted into a number of particular dimensions. Hence, various word and input embedding methods can be used for that. Code T5 tokenizer has been used to convert source code into tokens and those tokens are fed into DistilBERT Model which converts corresponding token into vector. a set of tokens with a maximum length of 100 is converted to a vector corresponding length. Because of Input Embedding, the same meaning and similar representation or similar d-dimensional vector value of source code is present. These embeddings are used to feed into the model.

Table 5-19: Input Embedding of Input_ids without positional Encoding

Input_ids	**Input Embeddings**
[1, 492, 13570, 3722, 2891, 3808, 470, 695, 813, 2]	[[-0.0156, -0.0650, -0.0179, ..., -0.0331, -0.0566, -0.0254], [-0.0170, -0.0533, -0.0213, ..., -0.0120, -0.0444, -0.0401], [-0.0616, -0.0178, -0.0302, ..., -0.0357, -0.0293, -0.0148], ..., [-0.0791, -0.0461, 0.0523, ..., -0.0320, 0.0109, -0.0170], [-0.0523, -0.0207, -0.0229, ..., -0.0400, 0.0124, -0.0736], [-0.0228, -0.0701, -0.0080, ..., -0.0318, -0.0715, -0.0258]] Dimension = ([10,768])

5.9 Positional Encoding

In transformer architecture, positional encoding is a technique used to represent the position of a word in a sentence in a continuous, dense vector representation. This information is important in tasks such as language modeling and machine translation, as the meaning of a word can change based on its position in the sentence.

One common way to add positional encoding to word embeddings is to use the sine and cosine functions. This is done by computing the values of these functions for different values of the position of the word in the sentence, and concatenating these values to the word embedding.

For example, let's say there is an input sequence of length N, and it is needed to add positional encoding to the word at “pos” position. The positional encoding is generated using a combination of sine and cosine functions of different frequencies, which are determined by the position of the element in the sequence and the embedding dimension. Specifically, the ‘i’th dimension of the positional encoding for the ‘j’th element in the sequence is defined as follows:

$$\mathrm{PE}_{(\mathrm{pos},2\mathrm{i})} = \sin\left(\frac{\mathrm{pos}}{10000^{\left(\frac{2\mathrm{i}}{\mathrm{d}_{\mathrm{model}}}\right)}}\right) \tag{5-3}$$

$$\mathrm{PE}_{(\mathrm{pos},2\mathrm{i}+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{\left(\frac{2\mathrm{i}+1}{\mathrm{d}_{\mathrm{model}}}\right)}}\right) \tag{5-4}$$

Where ‘$\mathrm{d}_{\mathrm{model}}$’ is the dimensionality of the embedding, ‘i’ is an integer in the range $\left[0,\ \left(\frac{\mathrm{d}_{\mathrm{model}}}{2}\right)\right]$, and 'PE’ is the matrix containing the positional encoding for each word.

The intuition behind this formula is that the sine and cosine functions have periodicity, and the values of the functions for different positions will be distinct. By encoding the position as a combination of these functions, we can add positional information to the embedding without changing the values of the embedding itself.

After computing the positional encoding, it is added to the word embedding, usually by element-wise summing the two vectors. This combined vector can then be used as input to a neural network for tasks such as language modeling or machine translation.

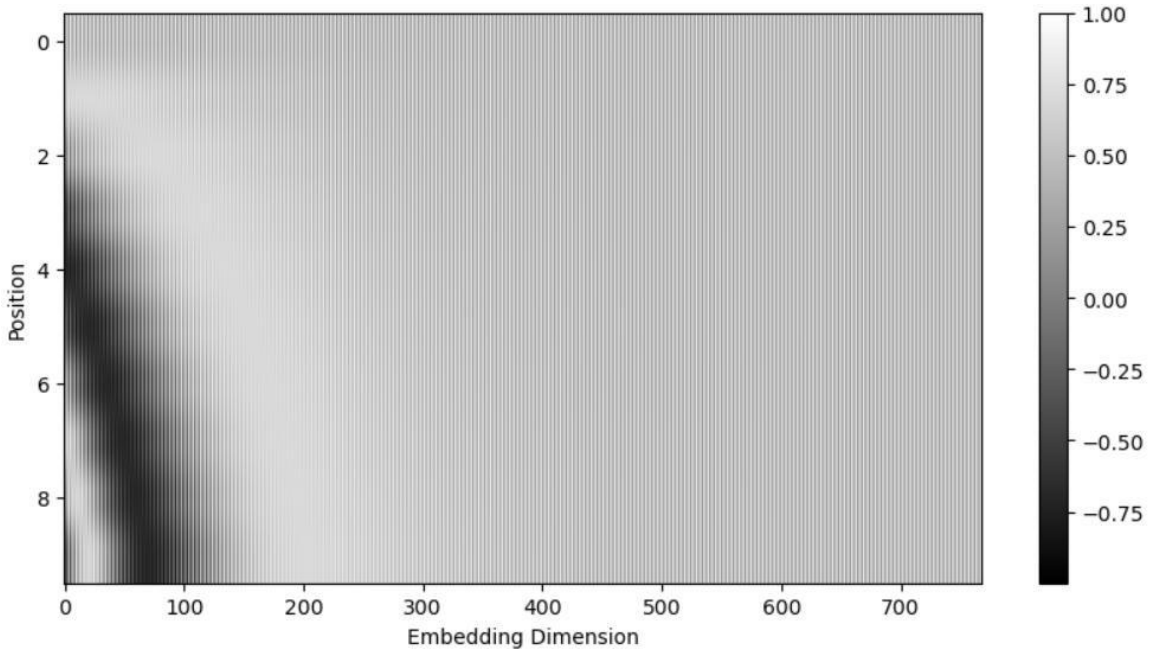

Figure 5-11: 10 Position Vector vs. 768 Word Embedding Dimension

This code which generates above figure from positional encoding matrix ‘pe’ for a sequence of length 10, where each element in the sequence is associated with an embedding vector of dimension d=768. The purpose of the positional encoding is to add positional information to the embeddings so that the model can differentiate between the positions of the elements in the sequence. The shape of the positional encoding vector is $([10,768])$

The frequency of the sine and cosine functions increases exponentially with the embedding dimension, so that each dimension captures information at a different scale. The choice of the value 10000 in the exponent is arbitrary but has been found to work well in practice.

The resulting ‘pe’ matrix is visualized using a grayscale heatmap, where the x-axis represents the embedding dimensions and the y-axis represents the position in the sequence. The values in the matrix indicate the strength of the positional encoding at each position and embedding dimension.

Similarly, for the 50 and 128 position vectors, the following figures give a more meticulous view.

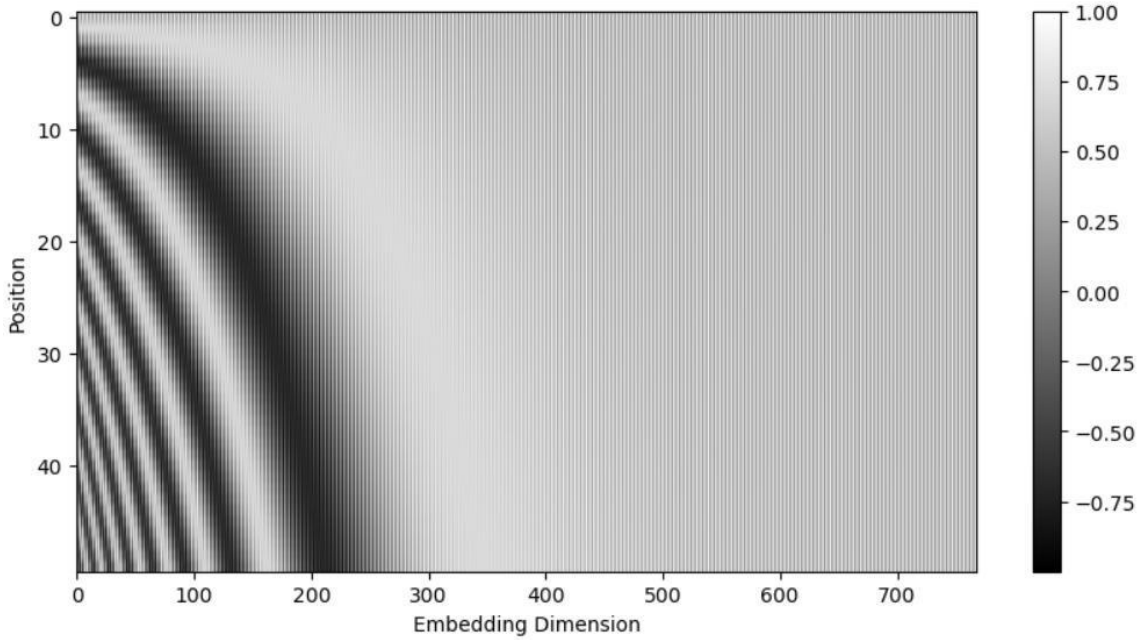

Figure 5-12: 50 Position Vector vs. 768 Word Embedding Dimension

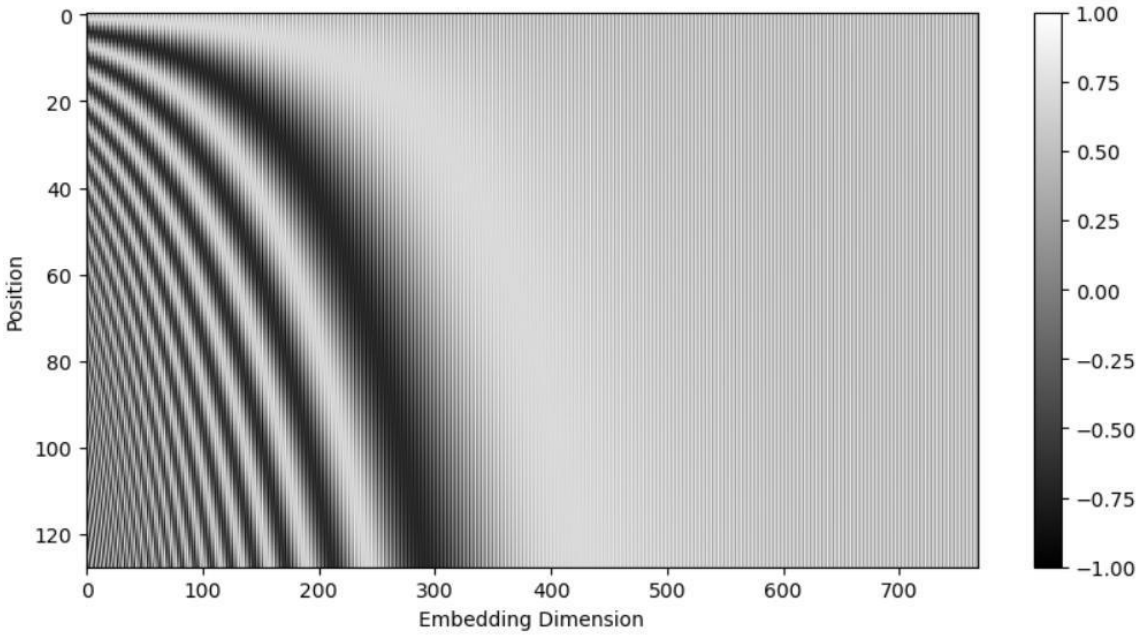

Figure 5-13: 128 Position Vector vs. 768 Word Embedding Dimension

Table 5-20: Positional Encoding Vectors

Positional Encoding Vectors	Shape
[[1.4731e-02, -2.3781e-02, …., -1.2616e-03, 1.5584e-02], [-6.2504e-04, 7.2108e-03, ..., 2.2016e-02, 2.5742e-02, -5.0380e-03], ..., [3.7319e-04, -1.2221e-02, -1.5817e-02, ..., 1.5544e-02, 1.2627e-02, 4.27 18e-03], [6.1085e-03, -2.6334e-03, -1.4536e-02, ..., 2.1069e-02, -2.6085e-05, 4.0 873e-03]]	Shape = ([10, 768])

Table 5-21: Input Embedding with Positional Encoding

Input Embedding with Positional Encoding	Shape
[[0.2408, -0.6792, 0.1108, ..., -0.2514, -0.5733, -0.0571], [0.1915, -0.4417, 0.0954, ..., 0.1202, -0.3246, -0.4156], [-0.4101, 0.4758, 0.2190, ..., -0.0252, 0.2059, 0.4225], ..., [0.1172, -0.5053, 0.1390, ..., -0.1887, -0.5760, -0.3693], [0.2721, -0.5572, 0.2448, ..., -0.1350, -0.5042, -0.2015], [0.0926, -0.7528, 0.3252, ..., -0.1825, -0.8538, -0.0894]]	Shape = ([10, 768])

After the dropout layer is applied:

Table 5-22: Final Embedding Vectors

Final Embedding Vectors	Shape
[[0.2408, -0.6792, 0.1108, ..., -0.2514, -0.5733, -0.0571], [0.1915, -0.4417, 0.0954, ..., 0.1202, -0.3246, -0.4156], [-0.4101, 0.4758, 0.2190, ..., -0.0252, 0.2059, 0.4225], ..., [0.1172, -0.5053, 0.1390, ..., -0.1887, -0.5760, -0.3693], [0.2721, -0.5572, 0.2448, ..., -0.1350, -0.5042, -0.2015], [0.0926, -0.7528, 0.3252, ..., -0.1825, -0.8538, -0.0894]]	Shape = ([10, 768])

5.10 Multi Head Attention

Multi Head Attention allows a model to attend to multiple positions in a sequence simultaneously, by computing attention scores between a query vector and a set of key-value pairs. These key-value pairs are derived from the input sequence, where each position in the sequence is mapped to a key-value pair. The query vector is also derived from the input sequence, but it corresponds to a specific position that the model is currently trying to predict. Multiple attention scores for each query vector, using different sets of learned projections for the query, key, and value vectors. These scores are then weighted and combined to produce a single embedding weight vector. Then, different aspects of the input sequence are maintained by attending to different sets of key-value pairs, which are obtained by applying different linear projections to the same input sequence. This is achieved by splitting the query, key, and value vectors into multiple heads 8 in the case of this project, and applying different linear projections to each head. Each head then computes its own attention scores and outputs a separate weighted combination of the value vectors. These weighted combinations are concatenated and projected back into the original vector space to produce the final attention weight of the multi-head attention layer.

For each input element, the attention weight from the DistilBERT Transformer after it is achieved is assigned which indicates the importance of that element to the output.

5.10.1 Self Attention

Self-attention determines the importance of the different parts of the input sentence. The sentence is represented as a sequence of tokens. By using a self-attention, network learns which tokens are related to each other to make predictions.

$$\text{Attention}\,(\mathrm{Q},\mathrm{K},\mathrm{V}) = \text{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V} \tag{5-5}$$

Query, Keys and Values are the three main inputs to the attention mechanism. If ‘n’ is the number of tokens and ‘d’ is the dimensionality of those tokens, then ‘n x d’ dimensional Query and ‘n x d’ dimensional Keys are matrices multiplied where ‘n x n’ dimensional vectors are formed. The ‘n x n’ dimensional matrix is called Attention Matrix. This

attention matrix multiplied with ‘n x d’ dimensional Value after which we obtain final ‘n x d’ dimensional matrix.

In order to capture the dependency of different sequences, the attention mechanism is highly beneficial when multiple queries, keys and values are used. This can be performed by using multiple attention operations in parallel. Query, key and values are first transformed using independently learnt linear projections, after that each set is used for separate attention. The final output is obtained by concatenating self-attention outputs and transforming them with another learnt linear projection. The design is multi-head attention where attention output is considered head.

Table 5-23: Transformer Key, Query and Value

Key	**Query**	**Value**
[[-0.4766, 0.7820, -0.2919, ...,0.1013, 0.9453, 3.2119], [-0.5228, 0.7572, -0.3394, ..., -0.2841, 0.8090, 2.2906], ..., [-0.7253, 0.7913, -0.1017, ..., 0.0826, 0.8226, 2.2313], [-0.6373, 0.6967, -0.5178, ..., 0.2399, 1.4333, 3.0199]] Key Shape = ([10, 768])	[[-0.2172, 0.2293, 0.4491, ..., -2.4213, -0.7798, 0.9900], [0.1515, 0.2009, 0.2343, ..., -2.6056, -0.7436, 0.1482], ..., [0.1662, 0.2199, 0.3928, ..., -2.2737, -0.7834, 0.3770], [-0.0541, 0.1729, 0.3899, ..., -1.9409, -0.7215, 0.8775]] Query Shape = ([10, 768])	[[0.3440, 0.3317, ..., 0.0692, 0.4051], [-0.5627, 1.0878, ..., -0.5766, -0.7924], ..., [-0.0247, -0.0227, ..., 0.1139, 0.2509, 0.0445], [0.4120, 0.1782, 0.2826, ..., 0.5687, 0.2055, 0.4816]] Value Shape = ([10, 768])

Table 5-24: Tokens and their Corresponding Attention Weights

Token	Attention Weights
‘Start’	[0.10558134, 0.09808115, ……,0.10317525, 0.11041001]
‘set’	[0.10713033, 0.09781968,……, 0.10553581, 0.1136036]
‘Dim’	[0.10244723, 0.10347213,……, 0.10069145, 0.09535287]
'ension',	[0.09925384, 0.09357029,……, 0.09870452, 0.10444252]
'Ġwidth',	[0.10621226, 0.10646176,……, 0.1049436 , 0.10431095]
'Ġheight',	[0.09913301, 0.09063389,……, 0.10110793, 0.10376085]
'Ġx',	[0.10417756, 0.09705792,……, 0.10368904, 0.11080562]
'Ġy',	[0.10628998, 0.09768552,……, 0.10506797, 0.1123969]
‘End’	[0.10642418, 0.09745905,……, 0.10553119, 0.11279049]
	Attention Weights shape = ([10,10])

Table 5-25: Output Vectors with Shape

Output vectors	Shape
[[[-0.0553, -0.0544, 0.0388, ..., 0.0915, 0.0666, 0.0574], [-0.0565, -0.0552, 0.0400, ..., 0.0912, 0.0635, 0.0570] [-0.0639, -0.0448, 0.0430, ..., 0.0898, 0.0644, 0.0528] ..., [-0.0566, -0.0554,0.0405, ..., 0.0942, 0.0667, 0.0569], [-0.0553, -0.0557, 0.0406, ..., 0.0939, 0.0647, 0.0567], [-0.0523, -0.0592, 0.0393, ..., 0.0922, 0.0663, 0.0573]]]	Output vectors shape = ([1, 10, 768])

When the value of self-attention is 1, it means that the model is assigning full attention to the corresponding token, which indicates that the token is highly relevant to the other token. In the self-attention heatmaps as presented subsequently, self-attention weight of 1 for a particular token indicates that the word is critical to understanding the meaning of the sentence.

When the value of self-attention is 0, it means that the model is not attending to the corresponding input token at all, indicating that the token is not relevant to the output. This happens if the input token is a padding token that has been added to make all input sequences the same length.

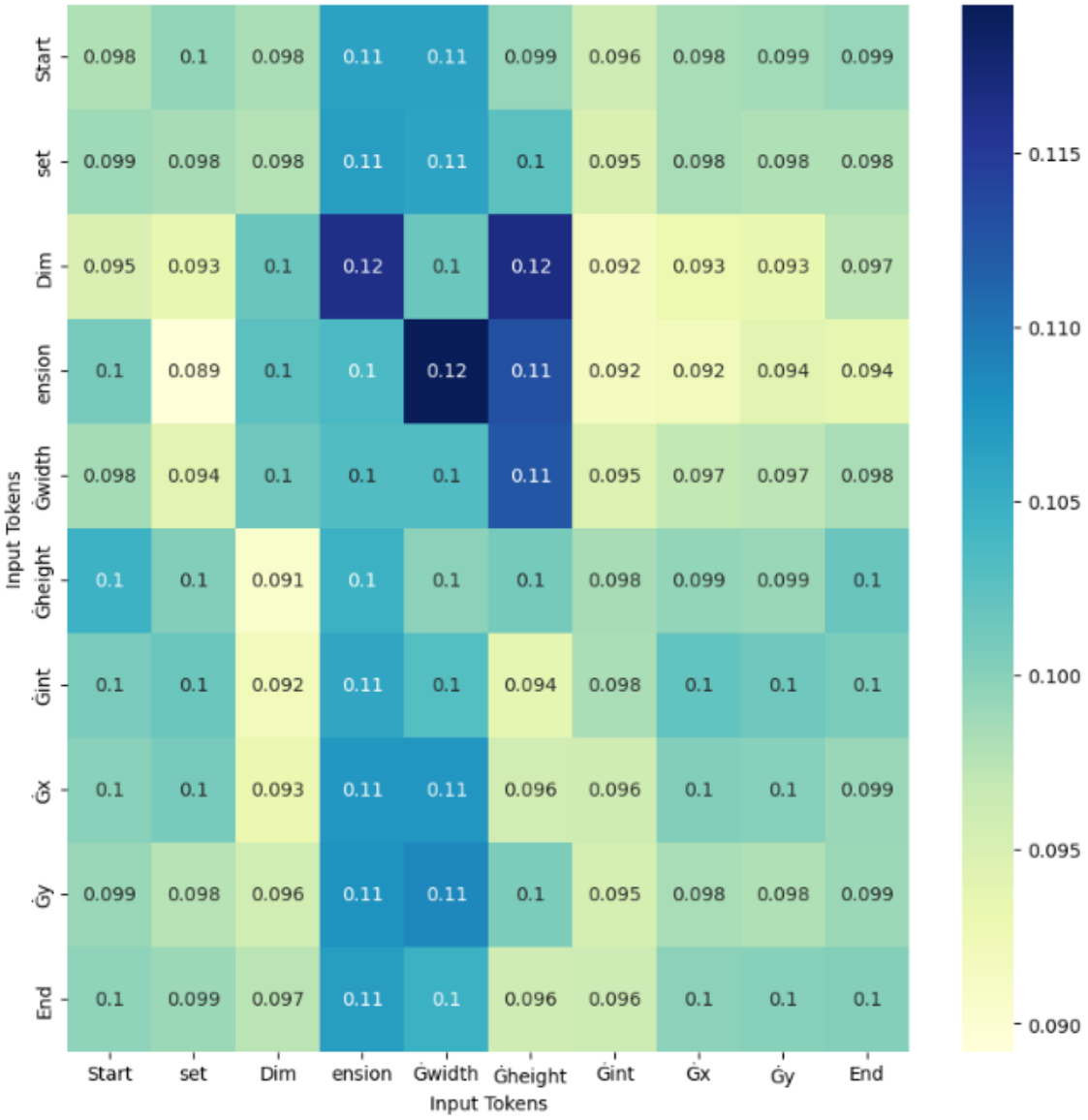

Figure 5-14: Self Attention 0

The value of self attention ranges from 0.09 to 0.12 as shown by the color indicator in the above figure. The ‘Dim’ token and ‘ension’ token has the largest attention of 0.12. It gives the insight that they are closely related in a particular way in the attention mechanism. They

exhibit great inter-connectedness as compared to 0.091 which is between ‘Dim’ and ‘Gheight’ token which means they exhibit relatively lower association.

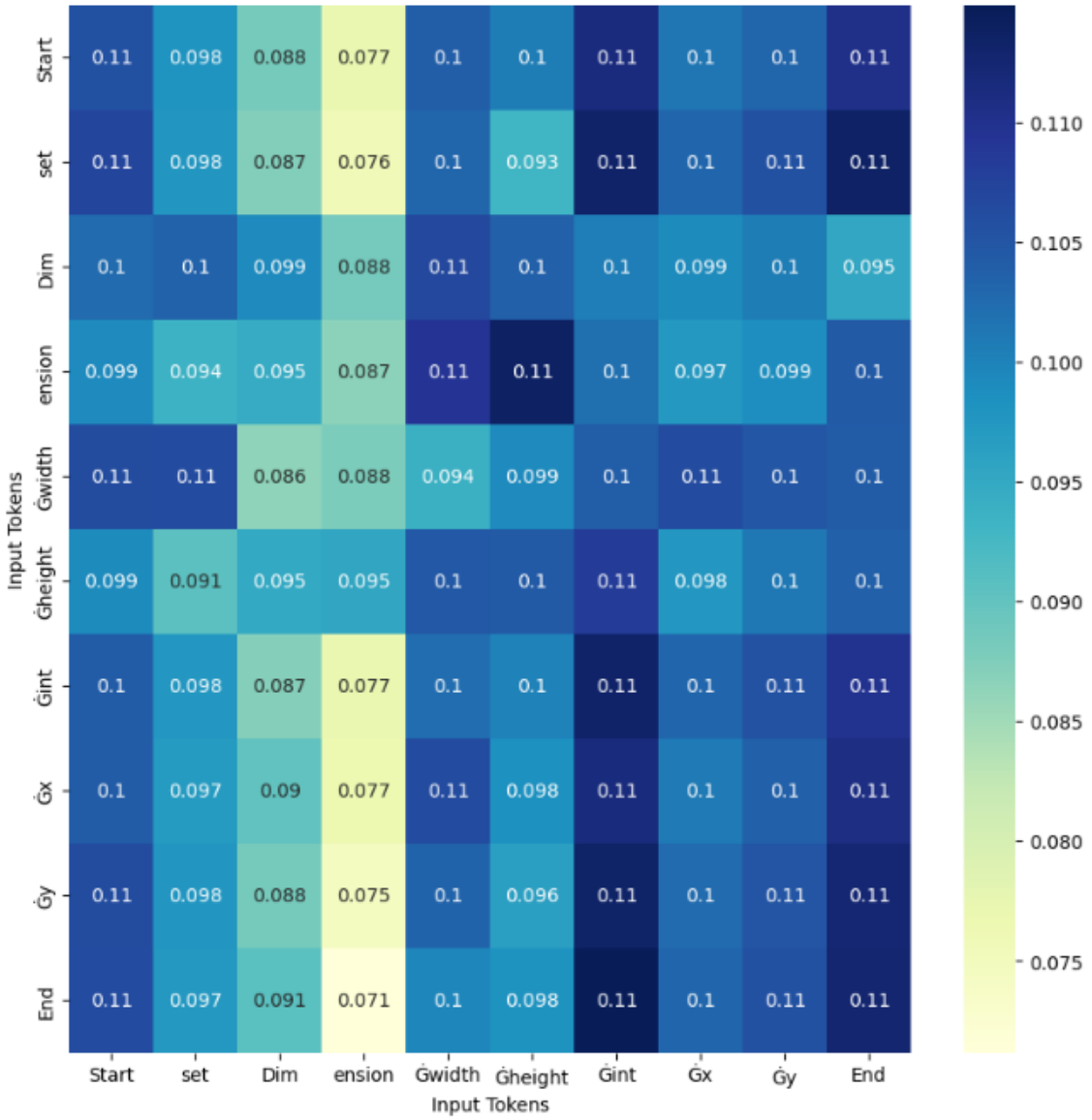

Figure 5-15: Self Attention 1

For this particular case, the self-attention value ranges from 0.071 to 0.11. Hence the token having a value of attention weight closer to 1 gives information that the token is highly relevant to the another token. In above figure ‘Gint’ and ‘Gx’, ‘Gint’ and ‘Gy’, ‘Gint’ and Gint has the maximum attention of 0.11 which indicates they are related in a particular way

in this attention mechanism. Similarly, ‘ension’ token and ‘End’ token are less related to each other as their attention weights are 0.071 which is nearer to zero.

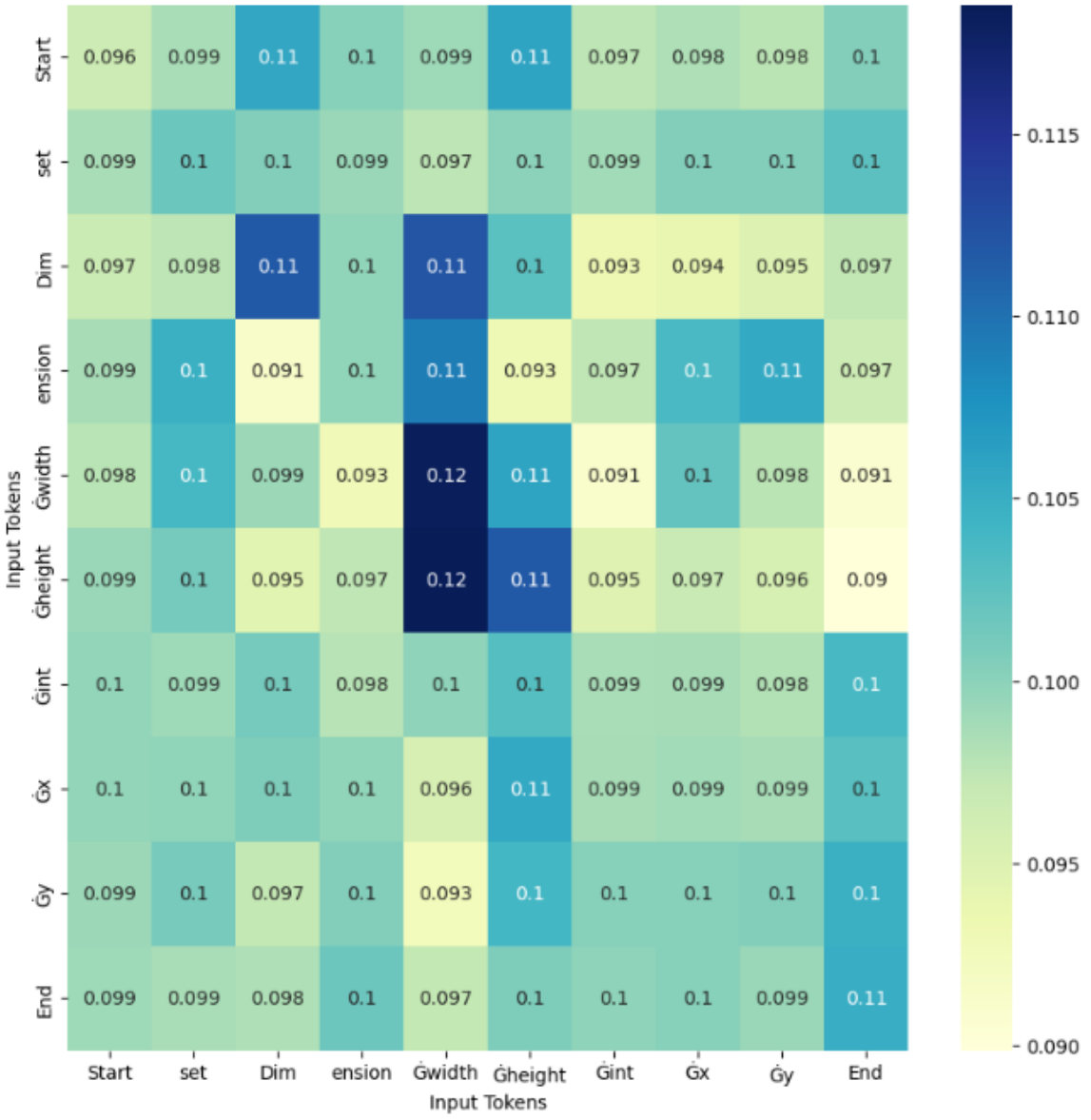

Figure 5-16: Self Attention 2

The self-attention score ranges from 0.090 to 0.12 in this instance. As shown in the diagram above, Gwidth's attention score reaches a maximum of 0.12 when paired with ‘Glength’ and itself, suggesting a distinct correlation within this attention mechanism. Conversely, the ‘End’ token and ‘Gheight’ token exhibit relatively lower association, with attention weights of 0.09 approaching zero.

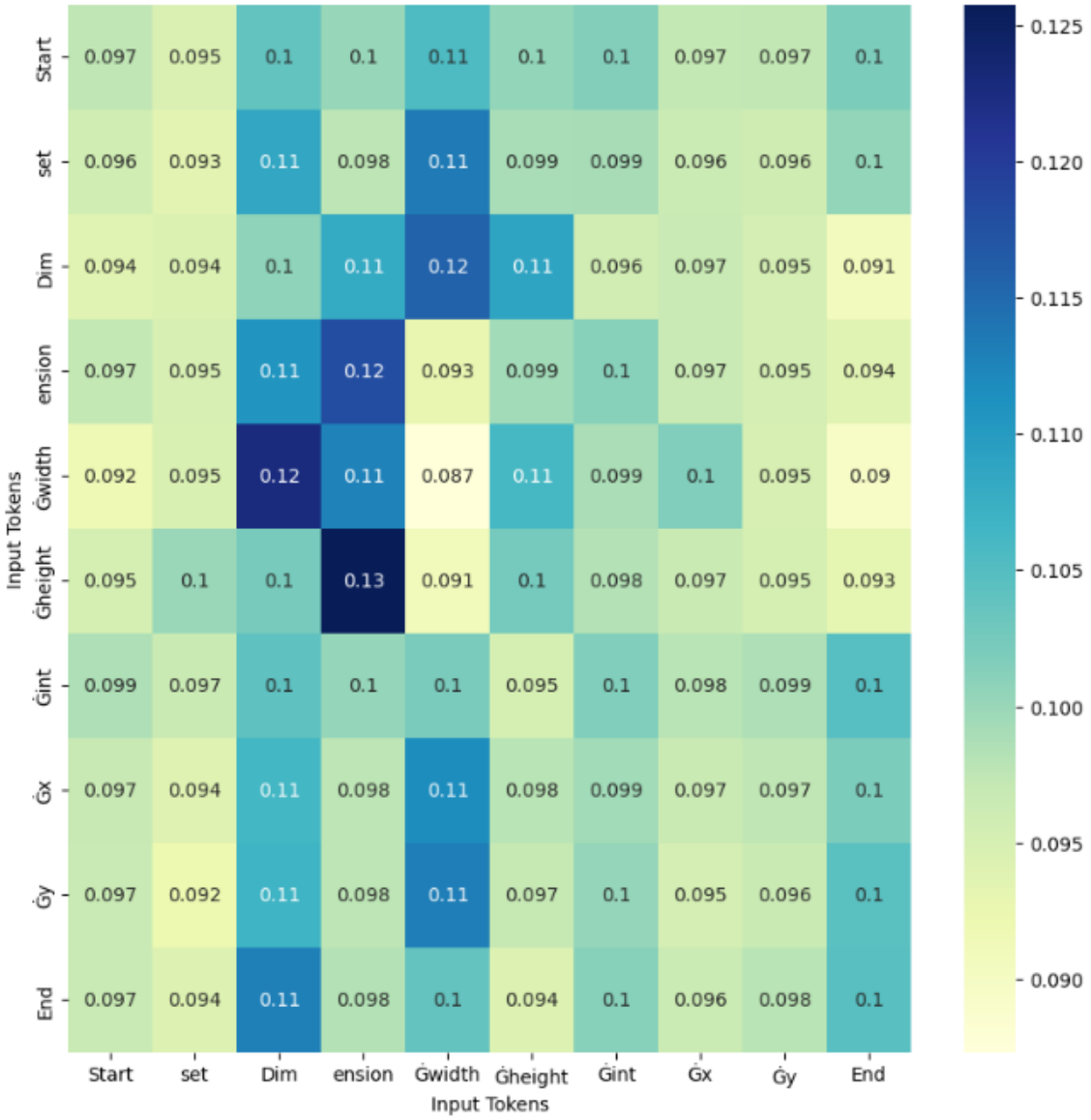

Figure 5-17: Self Attention 3

In the above diagram, there are some items or tokens that are related to each other in a certain way. It is being measured by how much attention or focus each item has on the other items. This score ranges from 0.090 to 0.13 for different pairs of tokens. When ‘ension’ token is taken with ‘Gheight’, the highest attention score of 0.13 is attained which suggests that they are closely related and share some common characteristics. On the other hand, the ‘End’ token and ‘Gwidth’ token have a relatively lower score of 0.09, which is almost close to zero. This suggests that they are not as closely related and have fewer things in common.

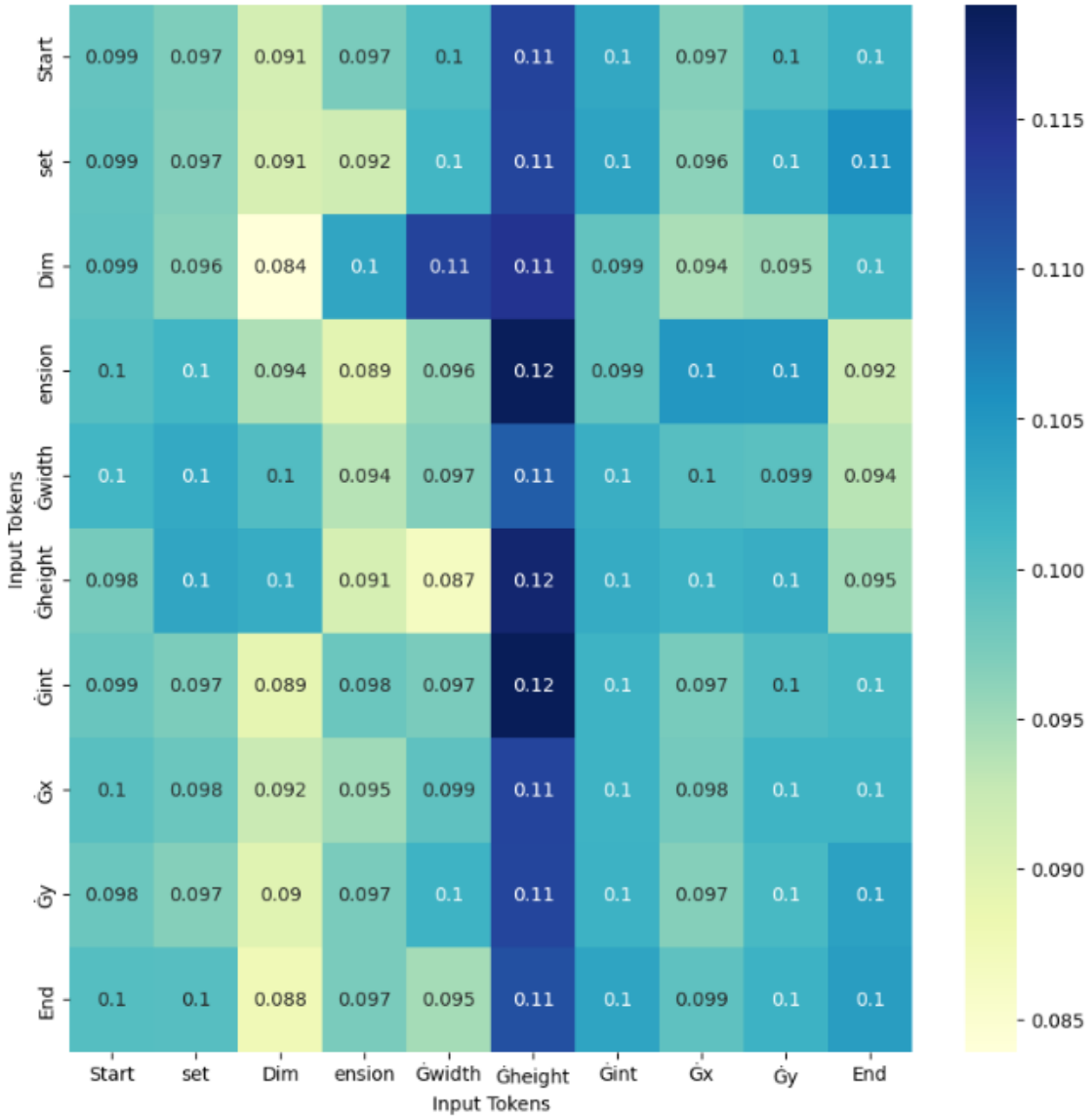

Figure 5-18: Self Attention 4

The diagram above presents a collection of tokens whose interconnectedness is being evaluated based on their level of attention towards each other. The attention scores for the various token pairs range from 0.085 to 0.12. Of note, the ‘Gheight’ token and ‘Gint’ token exhibit the highest score of 0.12, which is similar with ‘Gheight’ token with itself indicating a strong correlation and substantial shared characteristics. In contrast, the ‘Dim’ token with itself have a relatively lower score of 0.084, implying a weaker relationship and fewer shared features.

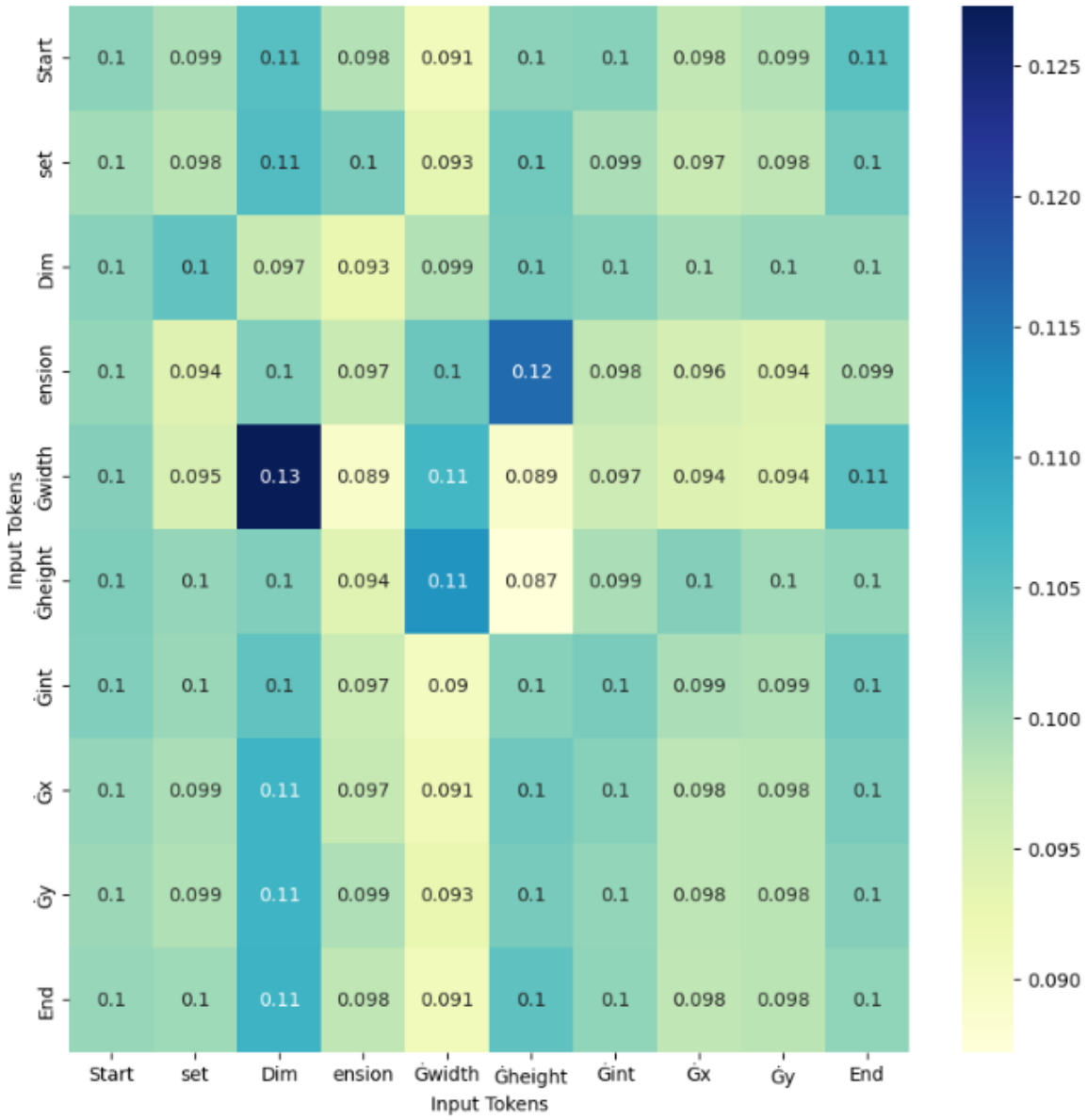

Figure 5-19: Self Attention 5

The attention scores for the various token pairs range from 0.089 to 0.13, indicating a range of different strengths of correlation and shared characteristics. Of particular note, the tokens ‘Dim’ and ‘Gwidth’ display the highest attention score of 0.13 which suggests a particularly strong correlation and a significant amount of shared attributes between these tokens. But, ‘ension’ and ‘Gwidth’ token has a relatively lower self-score of 0.084, indicating a weaker relationship and fewer shared features. This stands in contrast to the other tokens in the diagram, which exhibit a wider range of attention scores and levels of interconnectedness.

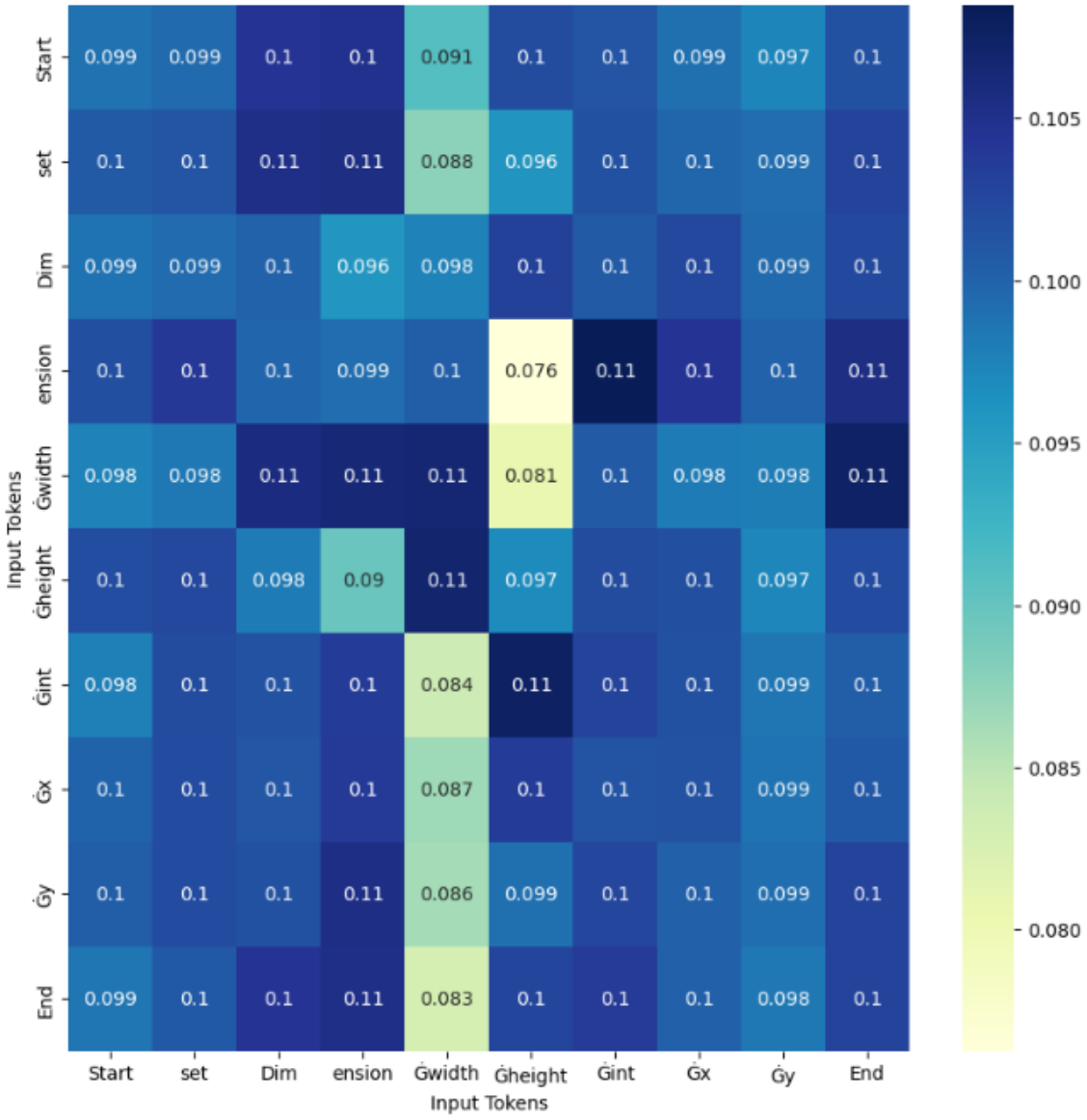

Figure 5-20: Self Attention 6

The range of attention scores assigned to various token pairs denotes varying degrees of correlation and shared characteristics, ranging from 0.076 to 0.11. Specifically, the tokens ‘Gint’ and ‘ension’, similarly ‘Dim’ and ‘Gwidth’ showcase the highest attention score of 0.11, signifying a notably strong correlation and a substantial amount of shared features. In contrast, the ‘Gheight’ and ‘ension’ token possess a relatively lower self-score of 0.076, indicating a weaker relationship and fewer shared features compared to other tokens in the diagram, which demonstrate a broad range of attention scores and levels of interconnectedness.

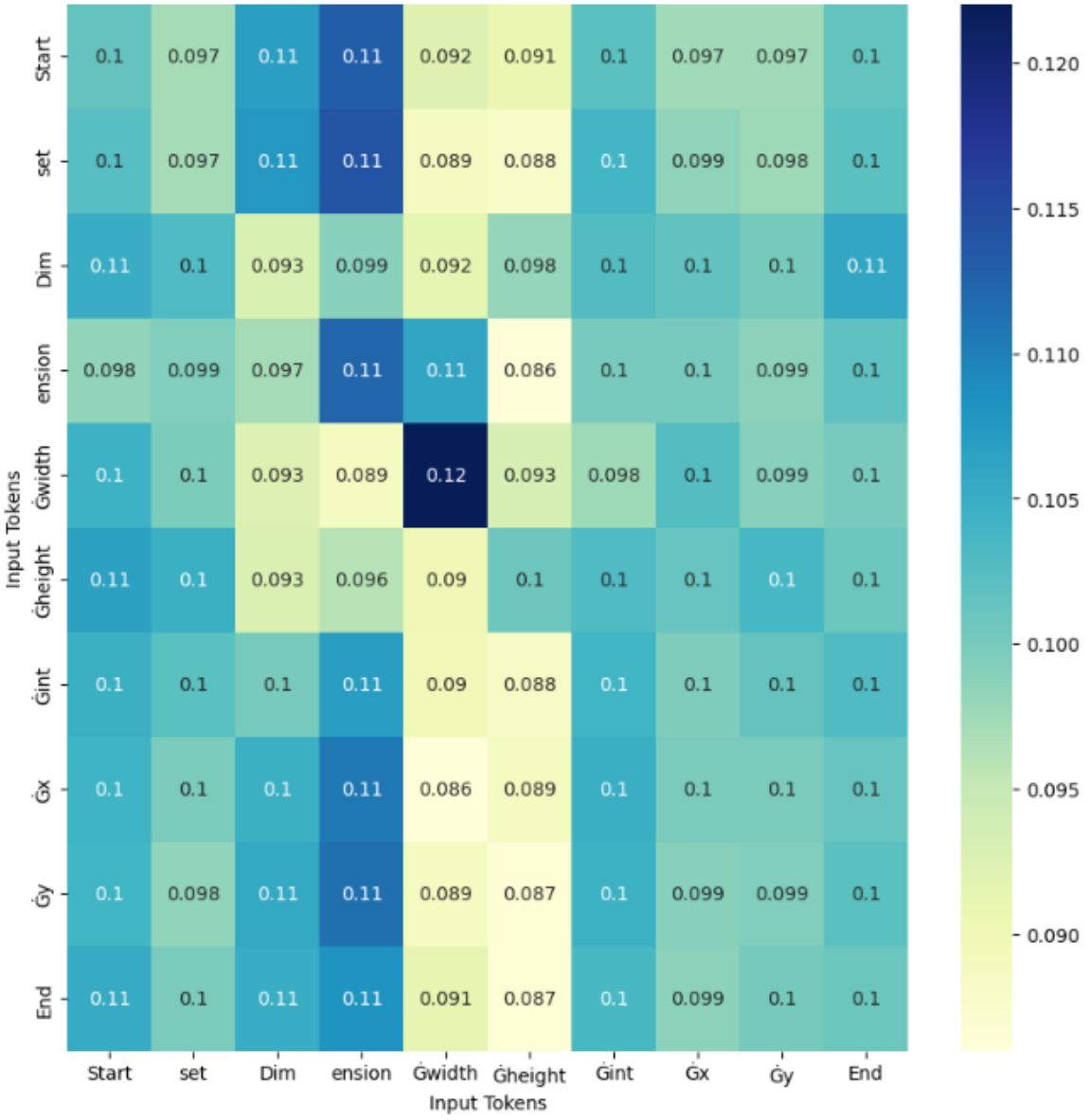

Figure 5-21: Self Attention 7

The attention scores assigned to token pairs indicate the level of similarity and correlation between them, which can range from 0.089 to 0.12. Specifically, the token ‘Gwidth’ with itself exhibit the highest attention score of 0.12, which implies a strong interrelationship and a significant degree of shared features. In contrast, the tokens ‘Gheight’ and ‘ension’ display a comparatively lower self-score of 0.086, indicating a weaker correlation and fewer shared traits among each other than other tokens in the diagram. The range of attention scores and levels of interconnectedness between tokens illustrate the complex nature of their interdependence and the varying degrees of similarity between them.

5.11 Architecture of DistilBERT Model

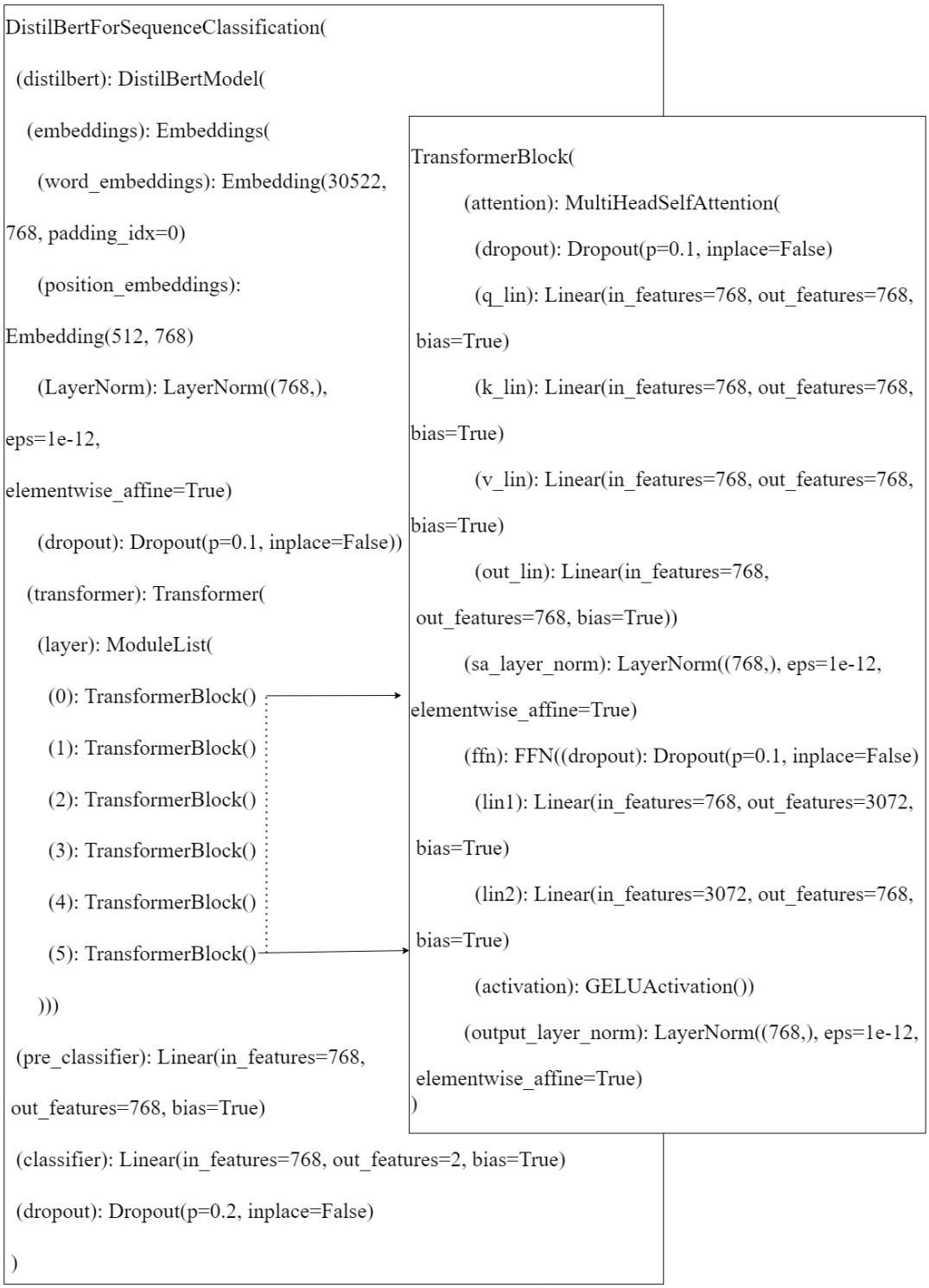

Figure 5-22: Internal View of DistilBERT Architecture

The DistilBERTForSequenceClassification model is a smaller variant of BERT, which is a transformer-based neural network architecture for natural language processing tasks. It is trained on a large amount of text data and can be fine-tuned on various NLP tasks such as text classification, question-answering, and named-entity recognition. In the project, the text classification variant of the model is used.

The model takes in a sequence of tokens as input and outputs a probability distribution over a set of predefined classes. It consists of two main components: The DistilBERTModel and a classification layer.

The DistilBERTModel component of the architecture is responsible for processing the input sequence and producing a sequence of contextualized representations for each token in the input. The input sequence is first tokenized and then passed through an embedding layer, which converts each token into a fixed-size vector representation. The embedding layer is composed of three sub-layers: word embeddings, position embeddings, and layer normalization.

The Transformer layer is the main component of the DistilBERTModel architecture. It consists of multiple transformer blocks, where each block contains a multi-head self-attention mechanism and a feed-forward network (FFN). The multi-head self-attention mechanism is used to compute a weighted sum of the input embeddings, where the weights are learned based on the similarity between each token in the input sequence. The output of the multi-head self-attention mechanism is then fed through a feed-forward network, which applies non-linear transformations to the input representation.

The classification layer is a linear layer that takes the final hidden state of the DistilBERTModel as input and produces a probability distribution over a set of predefined classes.

The model consists of five transformer blocks, where each block contains a multi-head self-attention mechanism and a feed-forward network. Each transformer block has a sa_layer_norm layer and an output_layer_norm layer for normalization. The

MultiHeadSelfAttention module computes the self-attention of the input embeddings by projecting them into query, key, and value spaces, computing the attention scores, and returning the weighted sum of the values. The FFN module applies two linear transformations with a GELU activation function in between. Finally, the output of each transformer block is normalized using layer normalization.

5.12 Hyperparameter Tuning

The hyperparameters tuned during these steps are:

5.12.1 Learning Rate

Learning rate determines the particular step size at which these project optimizers update the model parameter during the training phase. It is denoted by the Greek alphabet alpha (α). A larger learning rate causes the model to learn more quickly but it can face higher convergence. If the α is large then it can cause overshooting but if it is really small then the convergence of the model into the global minimum takes a really long time, that is it has slow convergence.

Table 5-26: Learning Rate Values

Learning Rate	2e-3	2e-4	2e-5	2e-6

5.12.2 Batch Size

Batch size determines the number of samples that are processed in a single iteration of the training process. In batch gradient descent, the model parameters are updated based on the average of the gradients computed from the entire batch of samples. A larger batch size results in slower convergence, but also reduces the variance in the gradients, making the optimization process more stable. Conversely, a smaller batch size results in faster convergence, but also increases the variance in the gradients, making the optimization process less stable.

Table 5-27: Batch Size Values

Batch Size	32	64

5.12.3 Number of Epochs

The number of epochs determines the number of times the model sees the entire training data during the training process. An epoch consists of one full pass through the training data. More epochs result in more training time, but also increase the likelihood of the model converging to the optimum solution. On the other hand, too few epochs can result in underfitting, where the model does not learn the underlying patterns in the data.

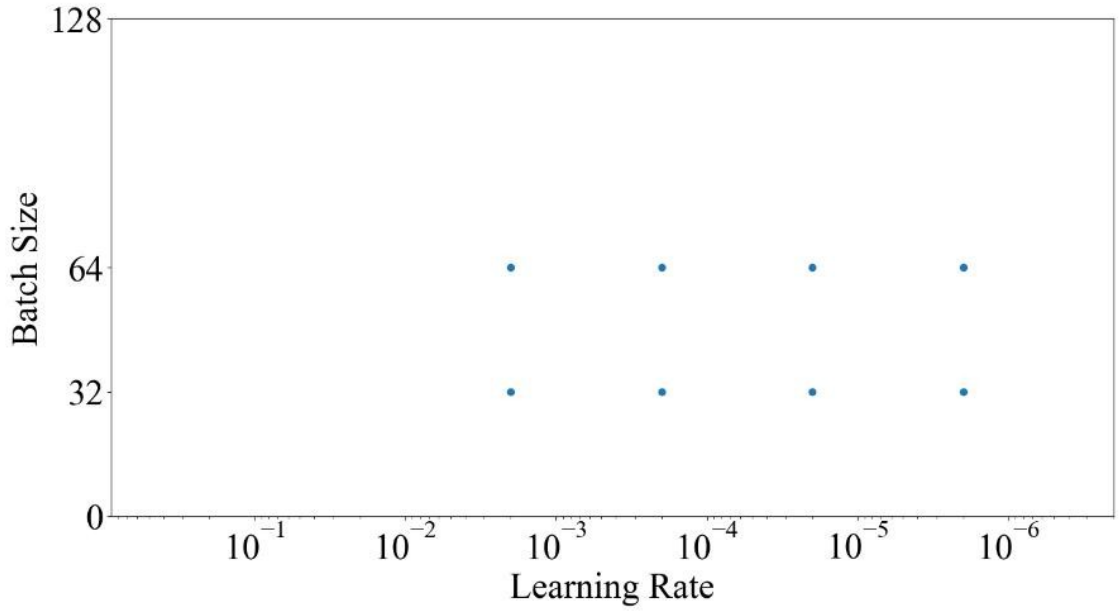

Figure 5-23: Various Hyperparameter Visualization

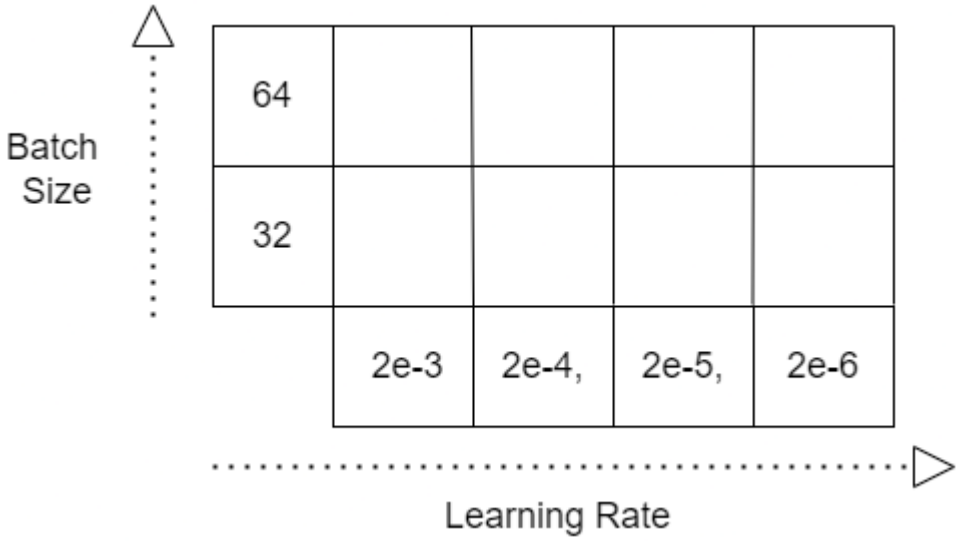

Figure 5-24: Hyperparameter Grid

5.13 Adam Optimizer

A good optimizer algorithm finds global minimums really fast, and does not get stuck in local minima, plateau region of saddle points. Adam optimizer algorithm is an extension to stochastic gradient descent algorithm. It works upon the AdaGrad and RMSProp mechanism. Stochastic gradient descent algorithm' learning rate (α) does not change at training and it makes use of a single learning rate.

$$\theta = \theta - \eta . \nabla \theta J(\theta; x^{(i)}; y^{(i)}) \quad (5\text{-}6)$$

In the paper [12] Adam: A Method for Stochastic Optimization authors talked about Adaptive Gradient Algorithm (AdaGrad) which adjusts learning rate which in turn improves performance of CV and NLP applications. Similarly, RMSProp also adjusts the learning rate and chooses a different learning rate for each parameter. Hence, it converges faster than GD or SGD. Hence Adam optimizer adapts the benefit of both AdaGrad and RMSProp.

5.13.1 Momentum

Taking an example from physics, in a frictionless bowl if a ball slides from top of it to bottom it doesn't go to the bottom and stop. But instead of stopping, momentum of the ball

pushes it back and forth. Similarly, for each step in model training a momentum from the previous step is added which has a best advantage of escaping local minima as the momentum may take it out of a local minima.

Hence for each step, in addition to the regular gradient, it also adds on the movement from the previous step.

$$w_{t+1} = w_t - \alpha\, m_t \tag{5-7}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\left[\frac{\delta L}{\delta w_t}\right] \tag{5-8}$$

Where,

Table 5-28: Symbols Description in Momentum

symbol	Description
m_t	aggregate of gradients at time t [current] (initially, $m_t = 0$)
m_{t-1}	aggregate of gradients at time t-1 [previous]
w_t	weights at time t
w_{t+1}	weights at time t+1
α_t	learning rate at time t
δL	derivative of Loss Function
δw_t	derivative of weights at time t

5.13.2 AdaGrad

Some features are sparse (scattered) compared to others. Features get trained at sparse features at much slower rate when the same learning rate is applied for all features. Setting a different learning rate for each feature can be applied but it can be messy very soon. By using an AdaGrad, for more time a feature is updated at present, the less it will be updated in future which gives a chance for sparse features to get updated. Hence the property allows AdaGrad to escape the saddle point. It takes a straight path whereas GD takes the approach

of sliding the steep slope first and takes care of a slower direction later. Hence, AdaGrad can take a straight path whereas GD takes a steeper path which in turn makes GD go in saddle points.

5.13.3 RMSProp

AdaGrad is slow hence RMSProp fixes that issue by adding a decay factor. Without decay, very small learning rates have to be set so the loss won't begin to diverge after decreasing to a point. The decay rate means that only the recent gradient matters and gradients are forgotten which are there from a long time. For example: if the decay rate is 0.99, the sum of gradient squared will be $\sqrt{(1-0.99)} = 0.1$ to that of AdaGrad, hence step is 10 times larging for the same learning rate. AdaGrad's sum of gradient squared accumulates very fast which in turn becomes humongous. Hence they take heavy toll and hence AdaGrad stops moving, but RMSProp keeps square under manageable size because of decay rate.

Adam (Adaptive Moment Estimation) combines Momentum and RMSProp. Speed of Adam is from momentum and ability to adapt gradients in different directions is from RMSProp. Sum of gradient (first moment) and sum of gradient squared (second moment)

are two common tools to improve optimizers. The Momentum method uses the first moment with decay rate to gain speed and AdaGrad uses the second moment with no decay to deal with sparse features. RMSProp uses a second moment with decay rate to speed AdaGrad whereas Adam uses both first and second moment which makes it really powerful.

$$v_t = \beta_2 \, v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta \omega_t}\right]^2 \tag{5-9}$$

$$w_{t+1} = w_t - \frac{\alpha_t}{(v_t + \varepsilon)^{\frac{1}{2}}} * \frac{\delta L}{\delta \omega_t} \tag{5-10}$$

where,

Table 5-29: Symbol Description in RMSProp

Symbol	Description
w_t	weights at time t
w_{t+1}	weights at time t+1
α_t	learning rate at time t
δL	derivative of Loss Function
$\delta\omega_t$	derivative of weights at time t
v_t	sum of square of past gradients. [i.e sum $\left(\frac{\delta L}{\delta \omega_{t-1}}\right)$] (initially, $v_t = 0$)

Finally, the entire equation for adam includes:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\left[\frac{\delta L}{\delta w_t}\right] v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left[\frac{\delta L}{\delta \omega_t}\right]^2 \quad (5\text{-}11)$$

Hyperparameters for Adam includes:

Table 5-30: Hyperparameter Description

Hyperparameter	Description
Learning Rate (α):	Larger values results in faster initial learning but risk overshooting
β_1 (from Momentum):	Exponential Decay rate for first moment
β_2 (from RMSProp):	Exponential decay rate for second moment
Epsilon (ε):	Used to prevent any division by zero in implementation which is a very small number

The choice of hyperparameter for Adam is:

Table 5-31: Hyperparameter Choice

Hyperparameter	Value
Learning Rate (α)	2e-5
Decay Rate from Momentum (β_1)	0.9 (default)
Decay Rate from RMSProp (β_2)	0.999 (default)
Epsilon (ε)	10e-8 (default)

5.14 Loss Function

5.14.1 Binary Cross Entropy (BCE)

Binary Cross Entropy (BCE) is a commonly used loss function for training machine learning models in binary classification problems where positive or negative is predicted. In binary classification, the predicted output is a probability that an instance belongs to one of the two classes. The BCE loss measures the dissimilarity between the predicted probability distribution and the actual distribution.

Binary cross Entropy is used to classify whether the program is buggy or not as this project only involves 2 classes. Mathematically, the BCE loss between the predicted probability y' and the true label y is given by:

$$\text{BCE} = -(\text{y} \times \log(\text{y}') + (1 - \text{y}) \times \log(1 - \text{y}')) \quad (5\text{-}12)$$

Here, y is the true label, which is either 0 or 1, and y' is the predicted probability that the instance belongs to class 1 or 0.

5.15 UML Diagrams

5.15.1 Class Diagram

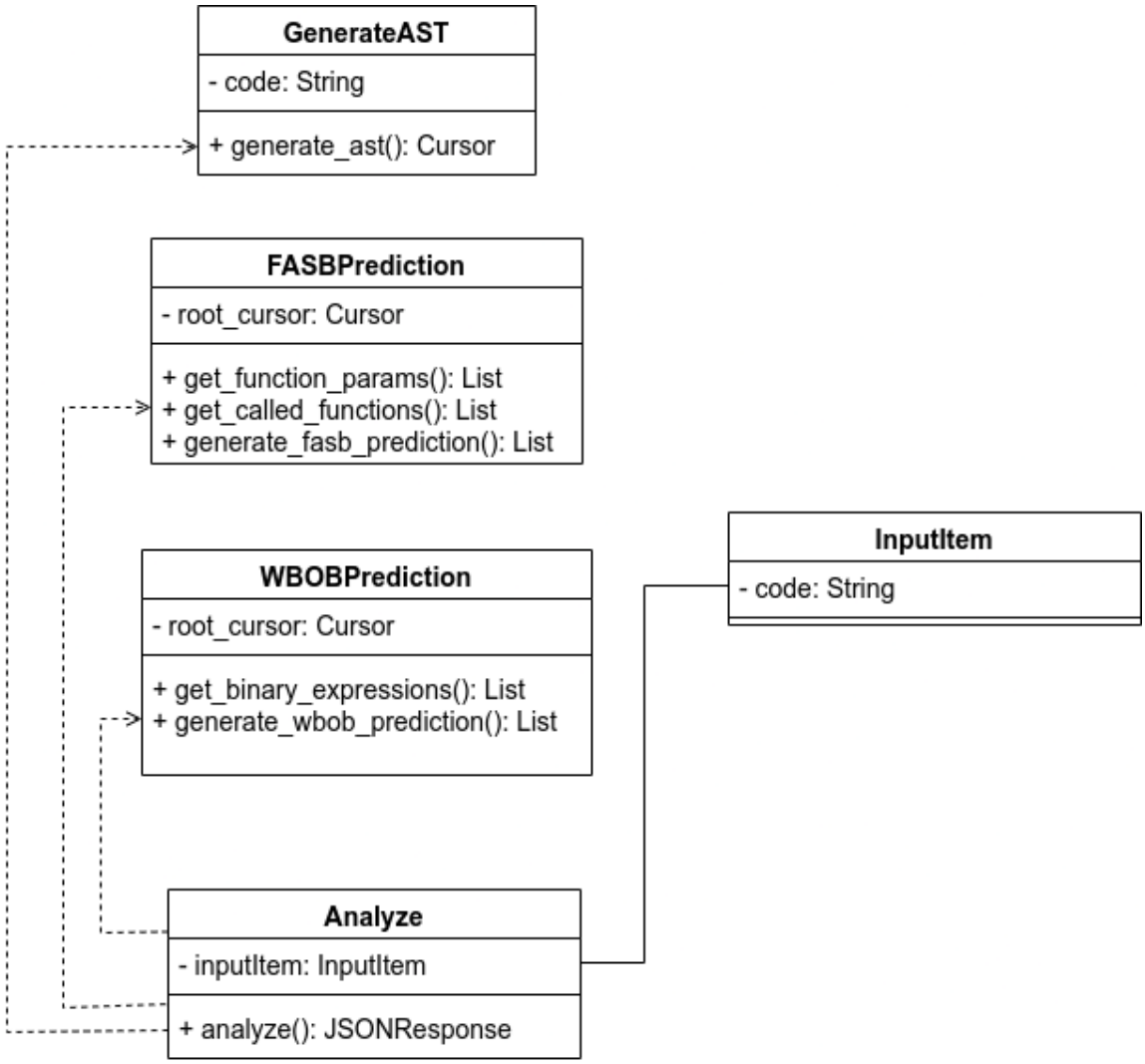

Figure 5-25: Class Diagram

As seen in the class diagram, we have several classes with their own task. GenerateAST class has a field named "code" and is used to get cursor of root node of Abstract Syntax Tree (AST). FASBPrediction class is used for the prediction of function arguments swapped bug and WBOBPrediction is used for the prediction of wrong binary operator bug. InputItem class contains a field named "code" which holds the data coming from

frontend of web application. Analyze class used GenerateAST, FASBPrediction and WBOBPrediction classes and needs InputItem class as a field.

5.15.2 Use-case Diagram

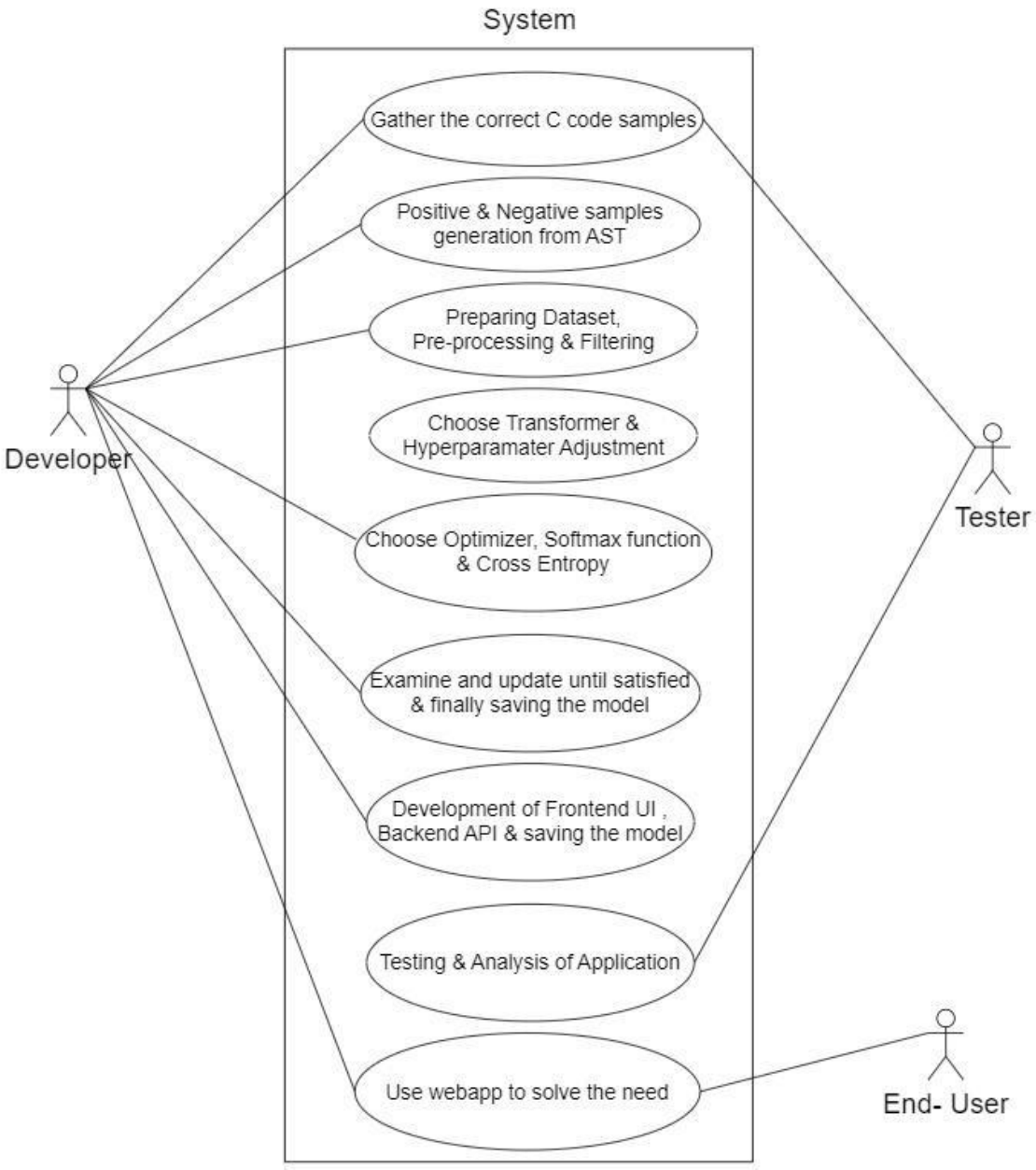

Figure 5-26: Use-case Diagram

In the context of the overall system, it is the collective responsibility of the Developer, Tester, and End User to ensure proper use case implementation. The Developer plays a crucial role in performing a multitude of tasks such as gathering C code snippets, generating positive and negative sample data, and carrying out data preprocessing and filtering procedures, amongst other duties. The Tester, on the other hand, works in tandem with the Developer to gather accurate C code samples and conducts a thorough analysis and testing of the program. Finally, the End User's primary responsibility is to interact with the web application and meet the project's specific needs. In essence, the collaboration between the Developer, Tester, and End User is critical to achieving the intended goals of the system. By engaging in their respective roles and responsibilities, they can ensure that the system functions as intended, and any issues or defects are addressed promptly. Ultimately, this collaborative effort enables the overall system to operate smoothly and efficiently, delivering maximum value to its end-users

5.15.3 Activity Diagram

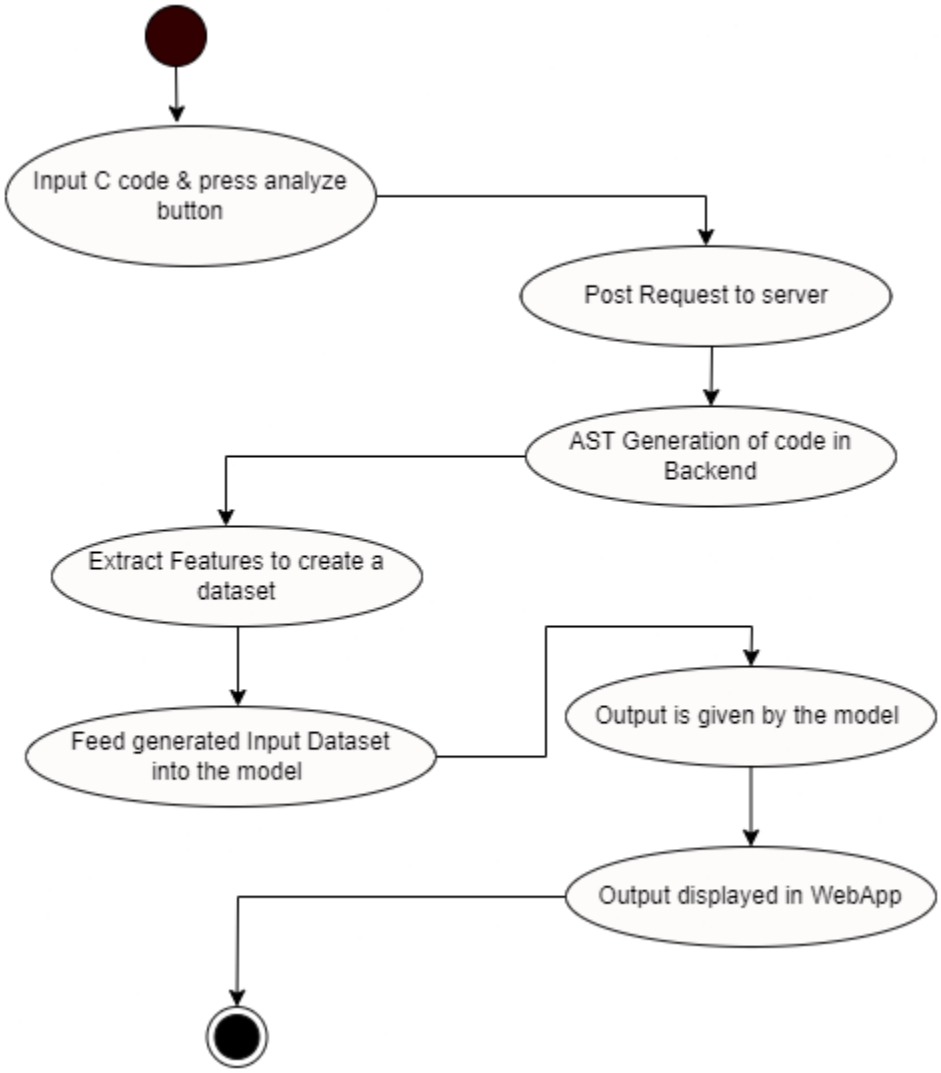

Figure 5-27: Activity Diagram

The described workflow involves a process for analyzing C code through the use of a web application, server, and machine learning model. The user interacts with the web application by writing C code and pressing an "analyze" button, which triggers a post request to the server. Upon receiving the post request, the server generates an Abstract Syntax Tree (AST) from the input C code. The AST generated by the server is then used to form an input dataset, which is fed to the machine learning model. The model performs its analysis of the code and produces an output, which is sent back to the web application

and displayed to the user. Overall, the workflow allows for the automated analysis of C code.

5.15.4 Sequence Diagram

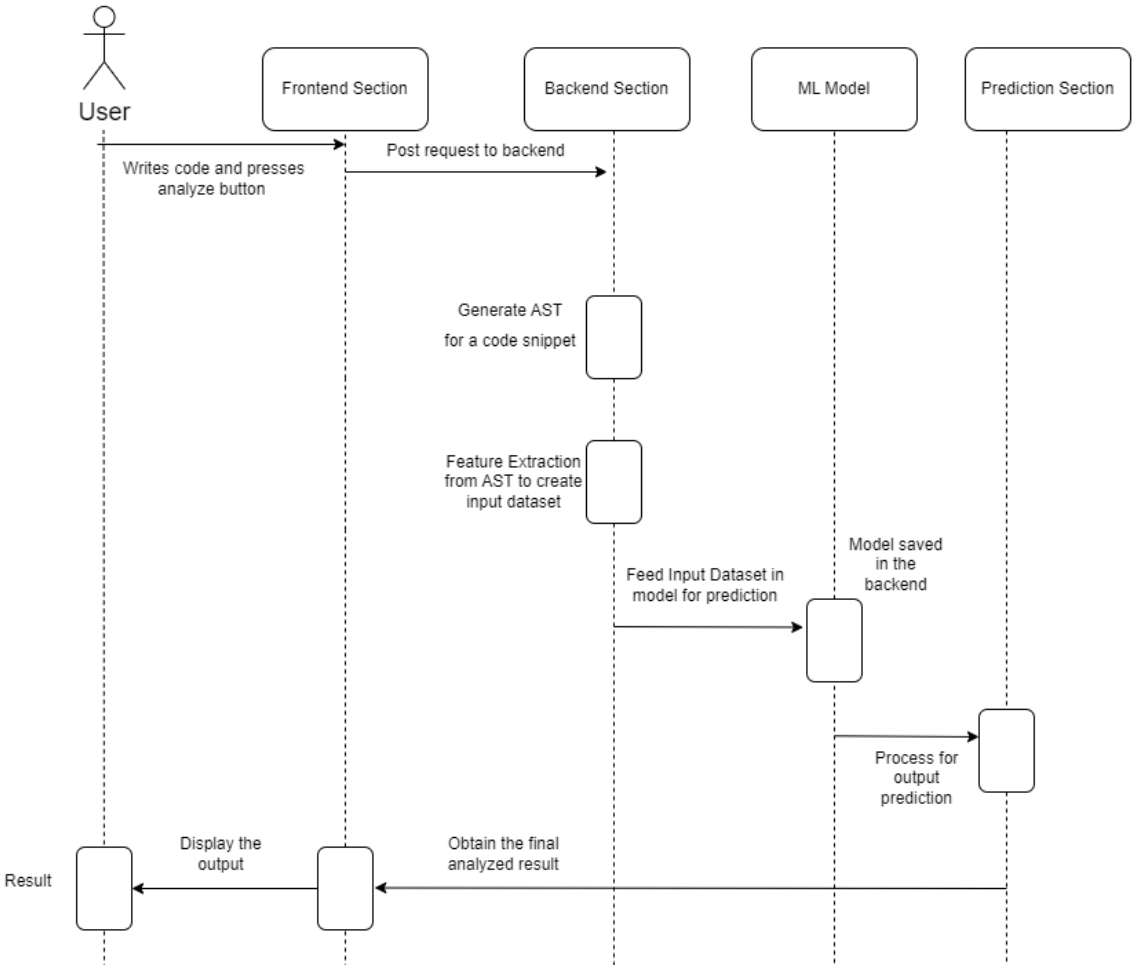

Figure 5-28: Sequence Diagram

In the realm of software development and testing, there exists a multitude of diverse responsibilities that must be undertaken by both the Tester and the Developer, while the end user is primarily concerned with receiving the desired outcome. Nevertheless, it is worth noting that at the core of their roles, all Testers and Developers are themselves users of the software they create. The user initiates the software application by writing code and triggering the analysis process through the activation of the analyze button, which is then transmitted to the frontend component. Subsequently, a post request is dispatched to the backend module, where the Abstract Syntax Tree (AST) of the code is generated. The AST

is then subject to Feature Extraction in order to create an input dataset that will serve as the foundation for further processing. This input dataset is subsequently fed into a predictive model. The model in the backend carries out computations to generate output predictions. Upon completion of this prediction process, the analyzed results are retrieved and subsequently displayed to the user via the frontend module.

5.16 Back-end Web Application using FastAPI

5.16.1 Overview

The Back-end (i.e. server) of the web application which was developed using FastAPI framework consists of a RESTful API endpoint on the path "/analyze" which accepts a post request where the C code is accepted through the post request body. After the backend gets the post request, AST of the sent C code is generated and required nodes of the AST are visited to form the input dataset. There can be multiple input datasets and each of them are passed to the Transformer model in the batch of 32 to predict the output. The probability of correctness or bugginess is obtained through the softmax function. Output is sent to the frontend web application in JSON format consisting of function name, start line, start column, end line, end column, prediction (0 or 1) and the prediction probability.

5.16.2 RESTful API Guiding Principles

REST (Representational State Transfer) is a popular architectural style for building web services, and RESTful APIs are APIs that adhere to the principles of REST. The guiding principles of RESTful APIs are as follows:

1. **Client-Server Architecture**: RESTful APIs separate the client and server responsibilities, allowing each to evolve and scale independently. This project also uses client-server architecture in which the client app made from React.js makes post request to the server for the analysis of C code.

2. **Statelessness**: RESTful APIs do not maintain client state on the server. All information needed to handle a request is included in the request itself. Every

request from clients is completely independent form the previous requests and the server doesn't keep track of the state of the client requests.

3. **Cacheability**: RESTful APIs should be designed to allow responses to be cached, to improve performance and scalability. At the current stage, cacheability hasn't been implemented directly by the developers.

4. **Layered System**: RESTful APIs can be composed of multiple layers, such as load balancers, cache servers, and application servers, to improve scalability and security. The layers such as load balancers, cache servers and application servers haven't been explicitly implemented.

5. **Code-On-Demand** (optional): RESTful APIs may allow for optional code to be executed on the client, such as JavaScript, to improve the user experience. The project executes JavaScript code on client-side web application.

6. **Uniform Interface**: RESTful APIs should have a uniform interface, including standard HTTP methods (GET, POST, PUT, DELETE, etc.), resource identification through URIs, self-describing messages, and hypermedia as the engine of application state. The project embodies all the required HTTP methods for the client-side web application. Currently, only POST method is present as the client-side application requires only one endpoint to accomplish the task.

By following these principles, RESTful APIs can provide a scalable and maintainable way to expose data and services to clients over the web.

5.16.3 POST Request

A POST request is an HTTP method used to send data to a server for processing. It is often used to submit form data to a server, but it can also be used to send data for other purposes, such as uploading a file, creating a resource, or updating a resource on the server. A POST request typically includes a payload, which is the data being sent to the server. The payload

can be included in the request body, and it can be formatted as plain text, JSON, XML, or any other data format.

5.16.4 JSON Conversion for Input and Output

JSON payload from the frontend contains only one key-value pair with key named “code” which is converted into a Python class named “Item” with one field named “code” in the backend and is accessed from there. While sending the output from backend to frontend, JSON with one key-value pair with key named “analysis” is used. The value of JSON output consist of two one-dimensional arrays having keys "function_args_swap_bug" and "wrong_binary_operator_bug" with their length same as the total possible buggy elements in each of the two categories in the given C code. Each element in the one-dimensional array contains information about possible bug in an element of the code which is shown in the figure below:

```
{
  "analysis": {
    "function_args_swap_bug": [
      {
        "function_name": "div",
        "arg1": "dividend",
        "arg2": "divisor",
        "start_line": "16",
        "start_column": "18",
        "end_line": "16",
        "end_column": "40",
        "label": 0,
        "probability": 0.9997864365577698
      }
    ],
    "wrong_binary_operator_bug": [
      {
        "left": "dividend",
        "operator": "/",
        "right": "divisor",
        "start_line": "18",
        "start_column": "36",
        "end_line": "18",
        "end_column": "54",
        "label": 0,
        "probability": 0.9394147992134094
      },
      {
        "left": "dividend",
        "operator": "%",
        "right": "divisor",
        "start_line": "19",
        "start_column": "36",
        "end_line": "19",
        "end_column": "54",
        "label": 0,
        "probability": 0.5474639534950256
      }
    ]
  }
}
```


Figure 5-29: JSON Output Response from Server

5.17 Front-end Web Application using React.js

5.17.1 Overview

The front-end (i.e. client-side) of web applications was made using React.js framework. It is a single page application consisting of an input section to write the C code, an “analyze” button to detect possible bugs in the given code and an output description section which gives pointwise description of the predicted results of function arguments swapped bugs.

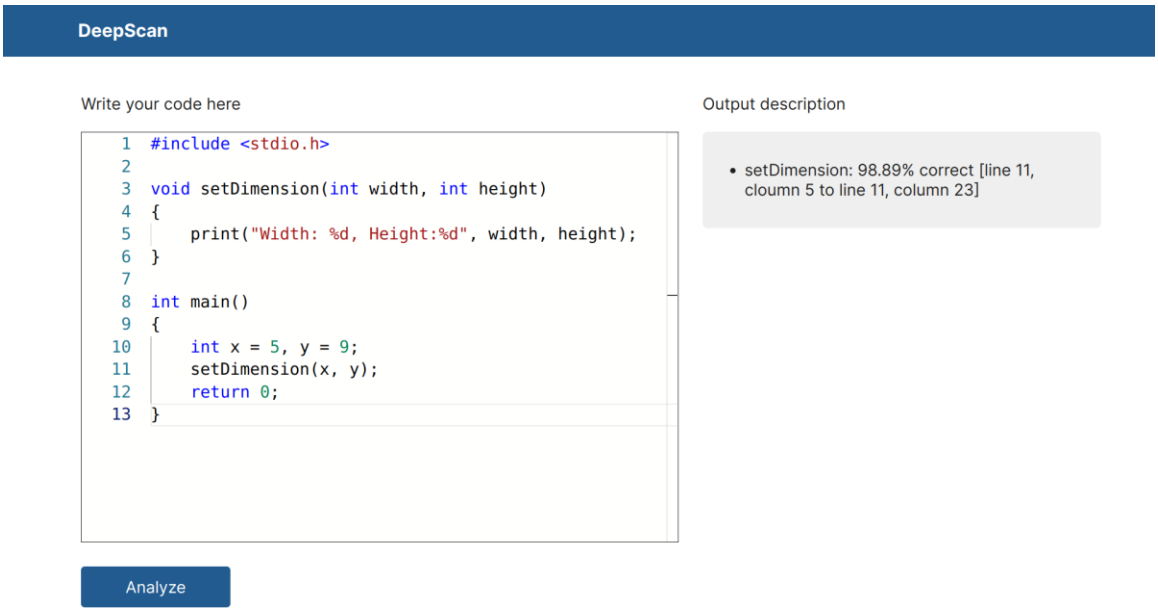

Figure 5-30: Front-end Web Application

As seen in the output description section, the function name, probability of correctness or bugginess and the start line number, start column number, end line number and lastly the end column number is given for users to exactly localize the function call and edit it accordingly. When the “analyze” button is clicked, the input section is checked if it’s empty and the request to the backend is not initiated if it’s empty. In general, it takes less than 1 second to generate the output on a 2GB NVIDIA GP108M [GeForce MX250] GPU.

5.17.2 DOM Structure

The HTML Document Object Model (DOM) is a hierarchical tree-like structure that represents the contents of a HTML document. Each element in the document is represented

as a node in the tree and the relationships between elements are represented as parent-child relationships in the tree.

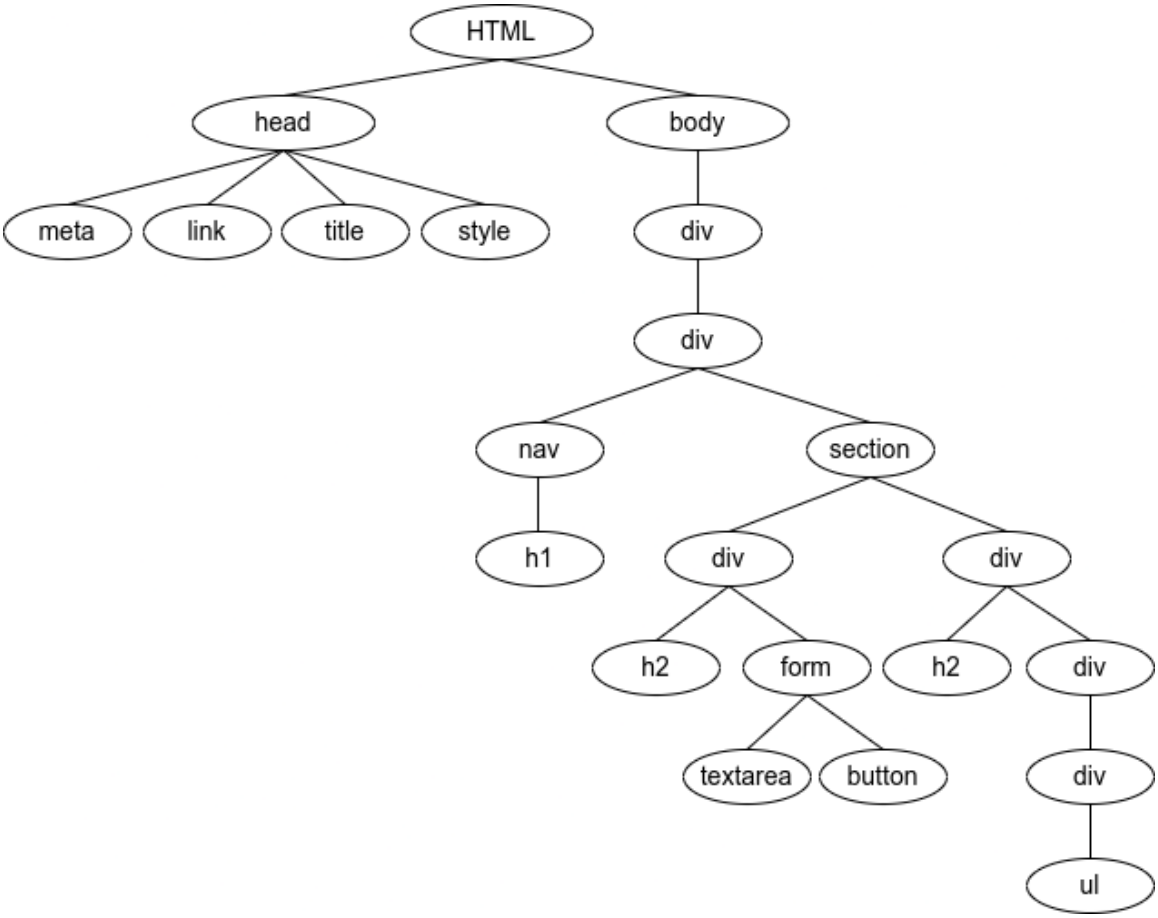

Figure 5-31: DOM Structure of Web Application

The root of the tree is the <html> element, which contains two main branches: the <head> element and the <body> element. The <head> element contains information about the document, such as its title, metadata, and links to external resources. The DOM of <head> element in our web application is shown in the figure below:

```
<!DOCTYPE html>
<html lang="en">
▼<head>
    <meta charset="utf-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="theme-color" content="#000000">
    <meta name="description" content="DeepScan: Automatic Bug Detection Through Na
    me Analysis">
    <link rel="apple-touch-icon" href="/logo192.png">
    <link rel="manifest" href="/manifest.json">
    <title>DeepScan: Automatic Bug Detection Through Name Analysis</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@300;400;500;60
    0;700&display=swap" rel="stylesheet">
    <script defer src="/static/js/bundle.js"></script>
  ▶<style>…</style>
  ▶<style>…</style>
  </head>
```

Figure 5-32: DOM Structure of <head> element

The <body> element contains the actual content of the document, such as headings, paragraphs, images, and links. Each HTML element in the document can have its own set of children, forming a nested structure. For example, a <p> (paragraph) element might contain several text nodes and other elements, such as or for emphasized text. The DOM of <body> element in our web application is shown in the figure below:

```
<body data-new-gr-c-s-check-loaded="14.1095.0" data-gr-ext-installed data-gr-ext-disabled="forever">
▼<div id="root">
  ▼<div class="App">
    ▼<nav class="App-Navbar">
       <h1 class="App-Navbar-title">DeepScan</h1>
     </nav>
    ▼<section class="App-code-analysis-section"> flex
      ▼<div class="App-code-input-div">
         <h2 class="App-code-input-title">Write your code here</h2>
        ▼<form>
           <textarea name="input-code" type="text" id="input-code" rows="22" required></textarea>
           <button class="App-code-input-analyze-button" type="submit">Analyze</button>
         </form>
       </div>
      ▼<div class="App-code-output-div">
         <h2 class="App-code-output-title">Output description</h2>
        ▼<div class="App-code-output">
          ▼<div class="App-code-output-text">
             <ul></ul>
           </div>
         </div>
       </div>
     </section>
   </div>
  </div>
</body>
```

Figure 5-33: DOM Structure of <body> element

The DOM provides a way for scripts and programs to dynamically access and modify the contents of a HTML document. It allows developers to interact with the document structure, modify its styles, and respond to events like user input. By manipulating the DOM, developers can create dynamic, interactive web pages and applications.

Instead of the actual DOM, React.js internally uses a virtual DOM and then reflects the changes in it to the actual DOM. As the frontend web application contains only one page with only one React component named “App”, virtual DOM is relatively simpler for this project. The virtual DOM in React is faster than updating the actual DOM for the following reasons:

1. **Batch updates**: The virtual DOM allows React to batch multiple updates into a single update, which is then propagated to the actual DOM. This means that instead of making multiple, separate updates to the actual DOM, which can be slow, React makes a single, optimized update.

2. **Minimal DOM manipulation**: The virtual DOM allows React to determine the minimum set of changes necessary to update the actual DOM. By only making the necessary changes, React.js minimizes the amount of DOM manipulation, which is one of the slowest parts of web development.

3. **Improved memory usage**: The virtual DOM uses a lightweight data structure to represent the actual DOM, which reduces the amount of memory used. In addition, React's garbage collector can reclaim memory used by virtual DOM nodes that are no longer needed, further reducing memory usage.

5.17.3 DOM Traversal

The web application was tested on chromium-based web browsers. The Chromium project, which is the open-source project behind Google Chrome, uses a depth-first search algorithm for DOM traversal. This means that the algorithm visits the children of a node before visiting its sibling nodes. It's worth noting that while Chromium uses a depth-first search algorithm for DOM traversal, other browsers or rendering engines may use different algorithms. The important thing is that the algorithm provides a way to access and manipulate the contents of a HTML document in a predictable and consistent way. React also uses the same algorithm for virtual DOM traversal.

5.17.4 Diffing Algorithm in React.js

React.js uses a diffing algorithm to efficiently update the user interface of a web application. The algorithm is called "reconciliation" and it's a key part of React's virtual DOM mechanism.

In React, the virtual DOM is a lightweight in-memory representation of the actual DOM. When the state of a React component changes, the virtual DOM is updated to reflect those changes. React's diffing algorithm then compares the updated virtual DOM with the previous virtual DOM to determine the minimum set of changes that need to be made to the actual DOM.

The diffing algorithm looks at the tree structure of the virtual DOM and identifies which components have changed, added, or removed. React then updates the actual DOM with the minimum number of operations necessary to bring it in line with the new virtual DOM. This process is optimized for speed and efficiency, allowing React to update the user interface smoothly and quickly, even as the state of the components changes.

5.17.5 Use of React.js Hooks

The React.js hook that has been used in this project is: useState. The useState hook is a built-in hook in React.js that allows adding state to functional components. State is a way to store and manage data that can change over time and affects the rendering of the component.

```
const [inputCode, setInputCode] = useState()
const [output, setOutput] = useState([])
const [analyzing, setAnalyzing] = useState(false)
const [emptyResult, setEmptyResult] = useState(false)
```

Figure 5-34: React.js useState Hooks Declaration

The four React.js hooks used in this project's client-side web application are: inputCode, output, analyzing and emptyResult. "inputCode" useState hook have been used to track the C code input by user, "output" to store the output result, "analyzing" to track the API request and response (i.e. it is set true when frontend initiates API request and finally set false when the server responds back or error occurs) and "emptyResult" hook have been used to track the empty output response from server.

```
<textarea
    name="input-code"
    type="text"
    id="input-code"
    rows="22"
    value={inputCode}
    onChange={(e) => setInputCode(e.target.value)}
    required
/>
```

Figure 5-35: React.js useState Hook (inputCode) Example Use Case

For example, “inputCode” useState hook have been used on a textarea (i.e. C code input box) in two-way binding (i.e. changes made to the textarea are reflected in the inputCode variable and vice versa) as shown in the Figure 5-35.

5.17.6 HTTP Headers and Payload

HTTP headers are metadata included in an HTTP request or response that provide additional information about the requested resource or the client requesting the resource. Headers are key-value pairs and can be used for a variety of purposes. The HTTP headers used in this project are:

Table 5-32: HTTP Headers Used

Key	Value
Accept	application/json
Content-Type	application/json

Accept header specifies the preferred format of the response and Content-Type header specifies the format of the request body.

HTTP payload refers to the data that is sent from the client to the server in an HTTP request. The payload is typically included in the body of an HTTP POST or PUT request, and it can contain various types of data, such as text, images, files, JSON, or XML. The payload is used to send data to the server, which then processes the request and returns a response. JSON format of payload having one key-value pair with key named “code” have been used for sending C code from frontend to analyze it in the backend of web applications.

6. RESULTS AND ANALYSIS

6.1 Detecting Swapping of Function Arguments

6.1.1 Epoch vs. Loss Plot

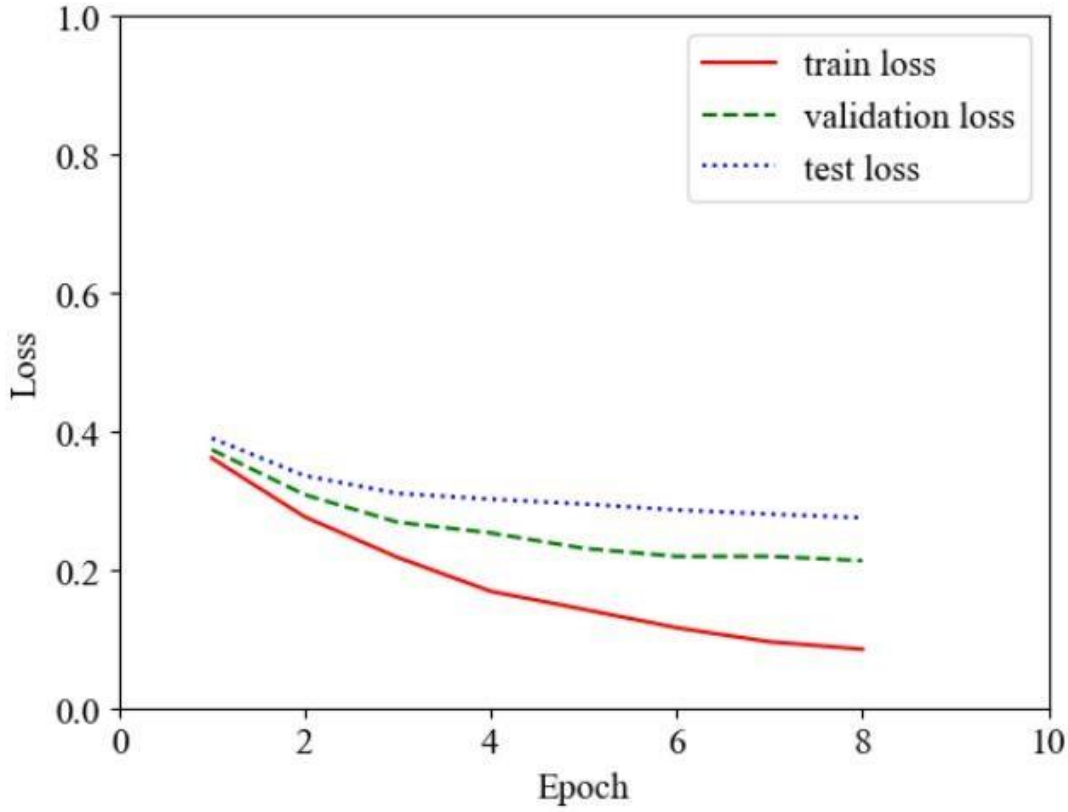

Figure 6-1: Epoch vs. Loss Plot

During the 8th epoch, there is a noticeable reduction in loss when the arguments are swapped for the function. For training, the decrease in loss is from around 0.36 to 0.085. For validation, the reduction is from 0.373164 to 0.212908, and for testing, it is from 0.3897 to 0.2750 which suggests that the model has successfully learned the underlying patterns of the function.

Hence, above values describes the performance of a machine learning model and how its performance improved over multiple training epochs.

6.1.2 Epoch vs. Accuracy Plot

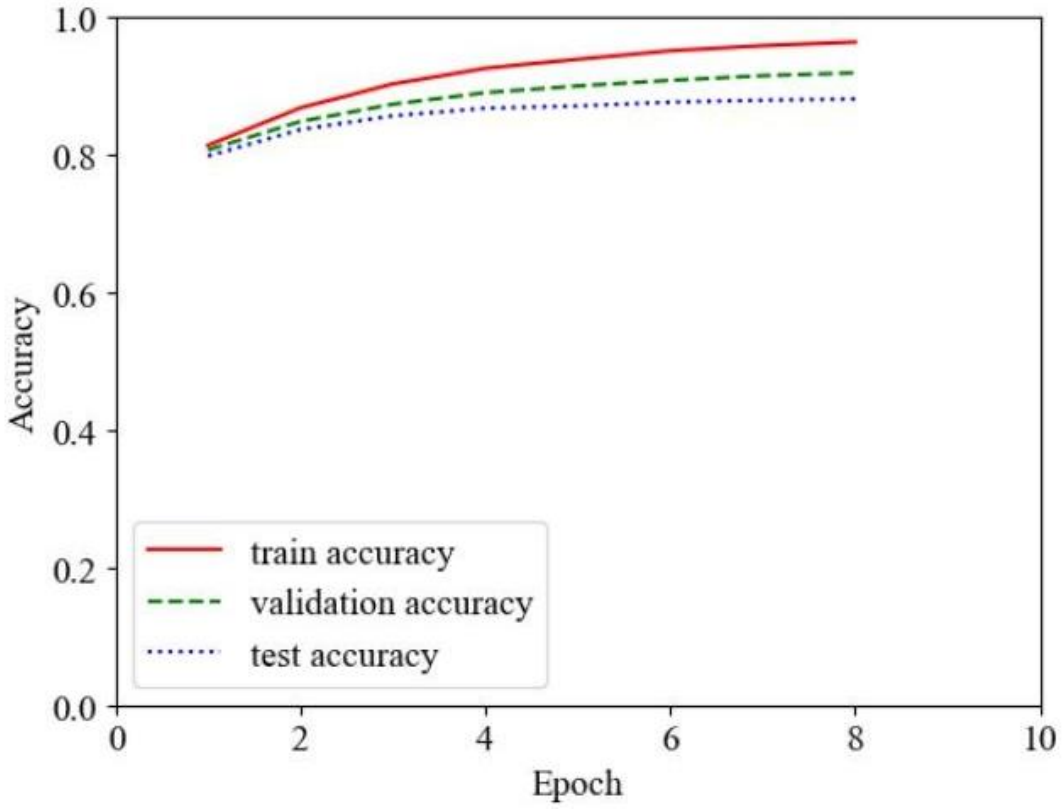

Figure 6-2: Epoch vs. Accuracy Plot

In regard to the operation of swapping arguments in the eighth epoch, one may observe a rise in accuracy from approximately 0.8135 to 0.9585 during the course of training. Moreover, in relation to the validation set, there is a noticeable increase in accuracy from 0.806947 to 0.9149, and the accuracy of the test set undergoes a corresponding elevation from 0.798124 to 0.879327 upon the completion of the eighth epoch.

6.1.3 Epoch vs. Precision Plot

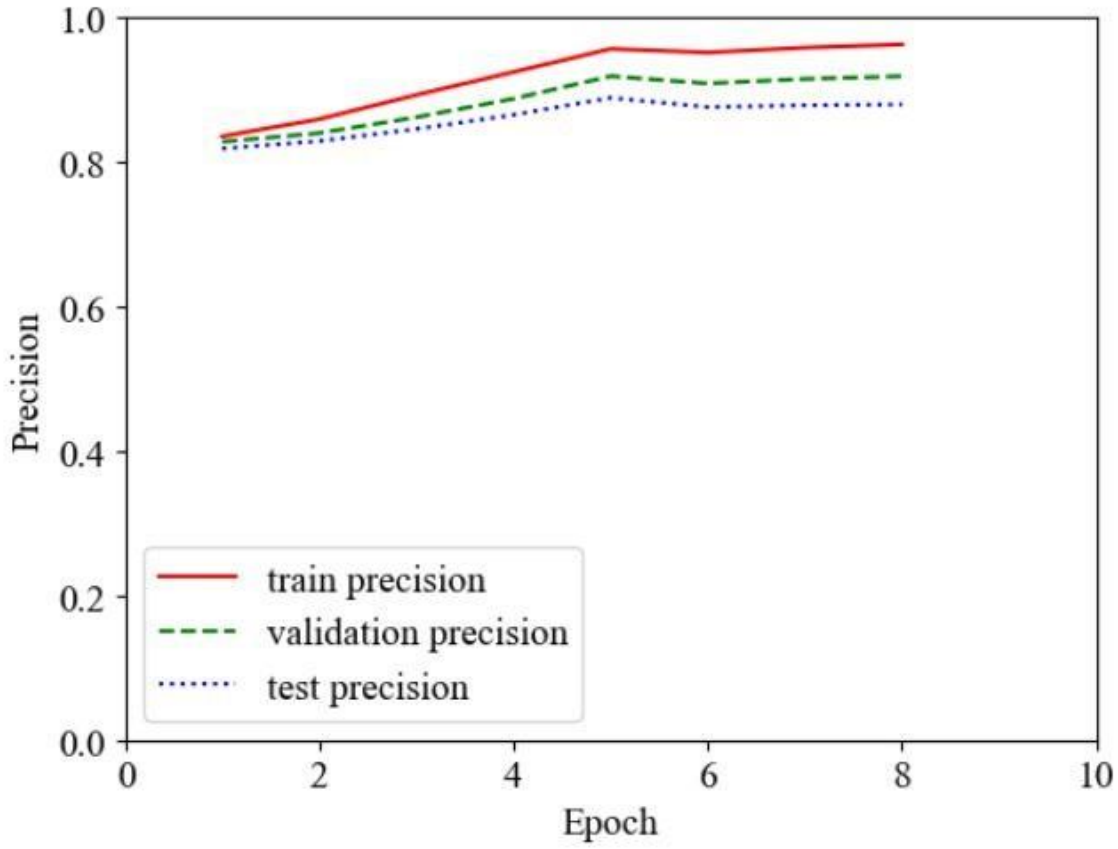

Figure 6-3: Epoch vs. Precision Plot

In light of the evidence presented, it becomes evident that the swapping of arguments after the eighth epoch of the function yields a marked increase in precision during training. This is reflected by a discernible rise from approximately 0.835346 to 0.962655. Furthermore, by the conclusion of the eighth epoch, the validation precision increases from 0.827957 to 0.918511, and the test precision escalates from 0.818518 to 0.879506.

6.1.4 Epoch vs. Recall Plot

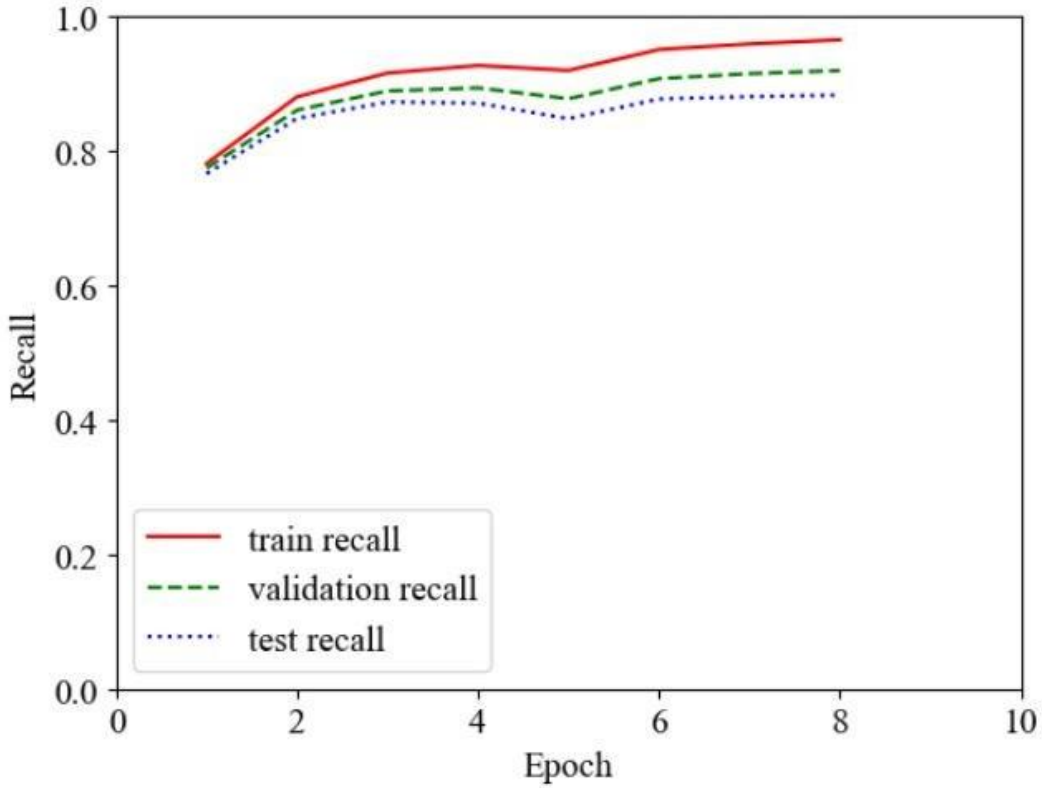

Figure 6-4: Epoch vs. Recall Plot

Regarding the function that swaps arguments, it is worth noting that an interesting pattern emerges during the training process, wherein the recall score starts to demonstrate a significant upward trend from approximately 0.781187 to 0.965338 by the 8th epoch. This implies that the model was able to learn the data more effectively and improved its recall performance by a significant margin. Additionally, the recall scores for the validation and test sets also exhibited an upward trend, increasing from 0.774613 to 0.919445 and 0.766111 to 0.883018, respectively. This observation suggests that the model's performance generalizes well on previously unseen data, indicating a good fit.

6.1.5 Epoch vs. F1 Plot

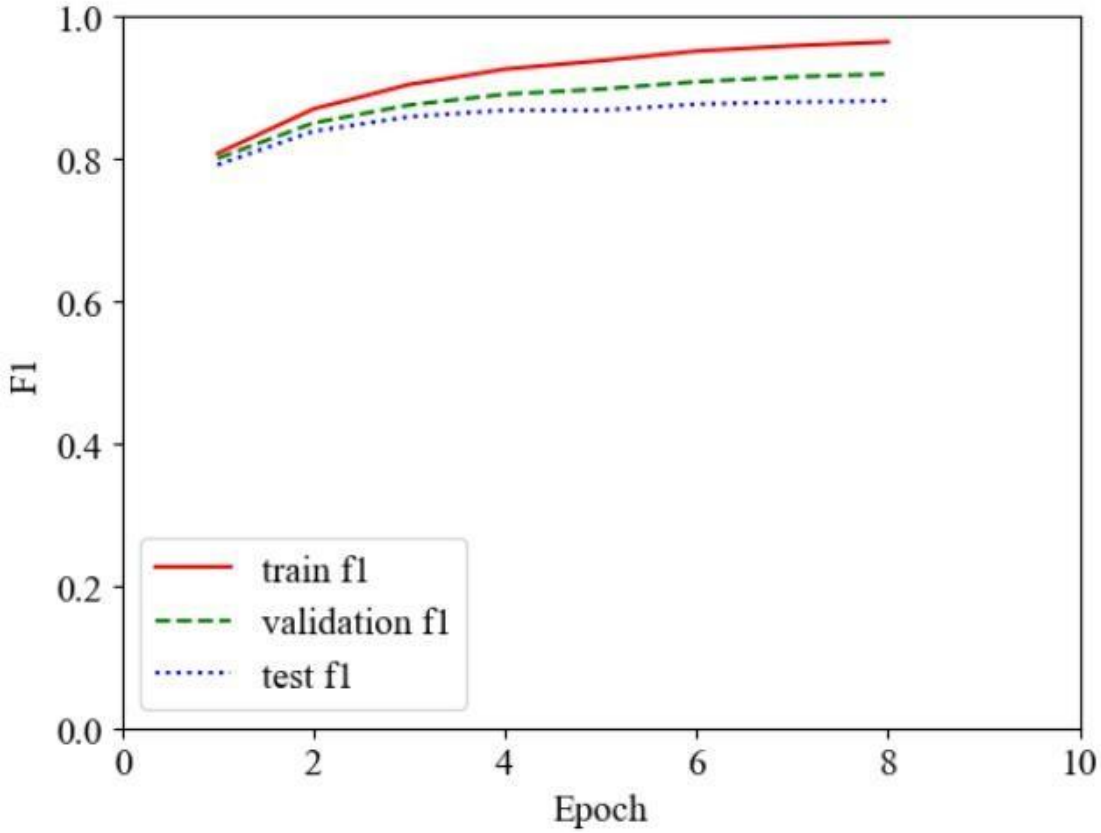

Figure 6-5: Epoch vs. F1 Plot

In the realm of programmatic functions, it has been observed that the action of swapping arguments following the 8th epoch yields discernible increases in F1 scores. Specifically, during the training phase, the metric exhibits an ascent from approximately 0.807359 to 0.963995. Moreover, the F1 scores of the validation data set demonstrate a similar trend, with the value increasing from 0.791448 to 0.881258. Furthermore, the recall metric of the test data set also exhibits a notable improvement, rising from 0.800397 to 0.918978 by the conclusion of the 8th epoch.

6.1.6 ROC Curve

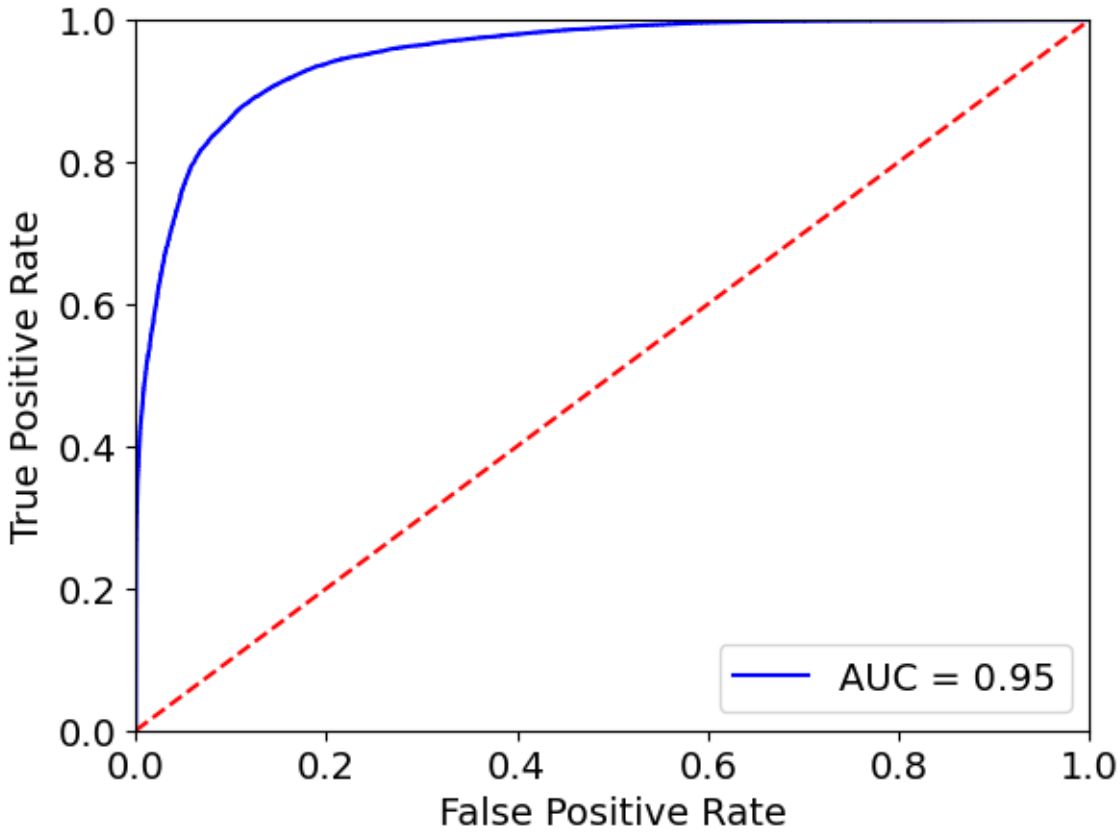

Figure 6-6: ROC curve of Function Arguments Swap

An AUC (Area Under the Curve) value of 0.95 indicates that the model has very good performance in distinguishing between the positive and negative instances. The AUC value ranges between 0 and 1. An AUC of 0.95 means that the model has a high probability of ranking a randomly chosen positive instance higher than a randomly chosen negative instance. Thus, the model is able to correctly classify a high proportion of positive instances while keeping the false positive rate relatively low. The ROC curve is closer to the top-left corner of the plot which implies the better the performance of the model.

6.1.7 Precision-Recall Curve

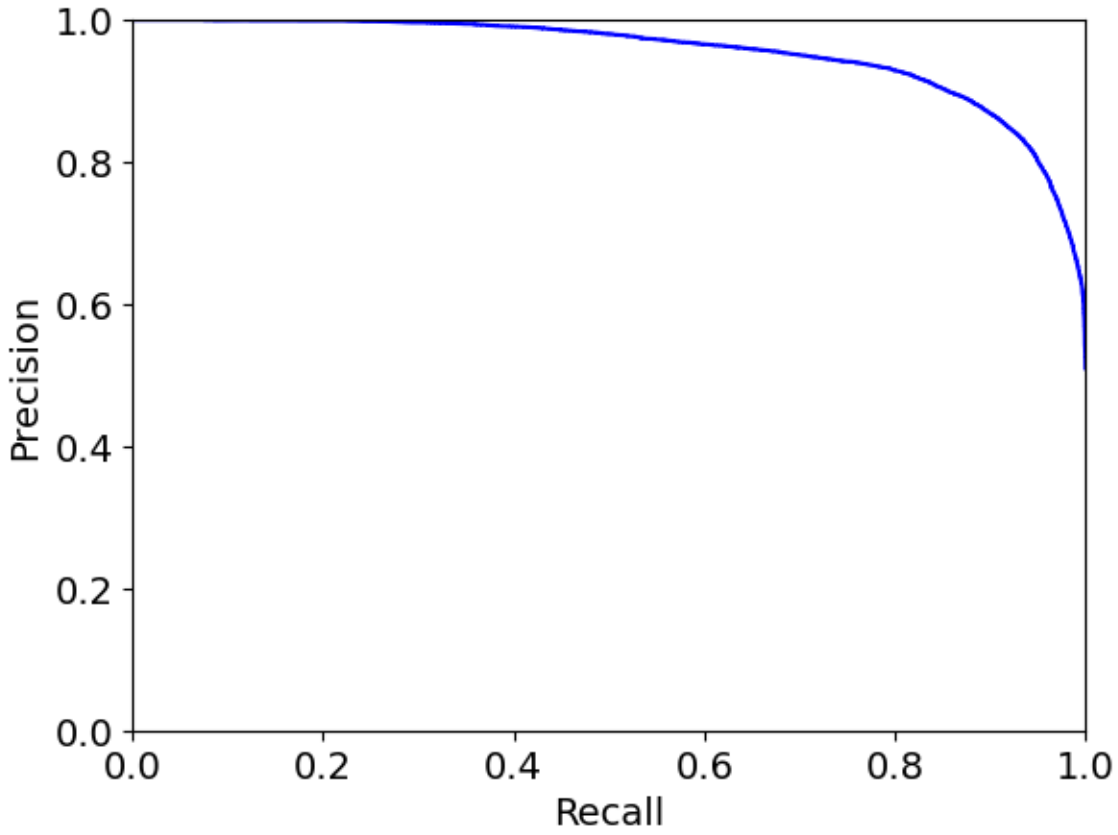

Figure 6-7: Precision-Recall Curve of Function Arguments Swap

The above precision-recall curve shows how well the model is able to correctly identify positive instances (precision) while minimizing the number of false negatives (recall). The curve is closer to the top-right corner of the plot. Hence, the performance of the model is better.

The precision-recall curve is particularly useful when the classes are imbalanced, that is, when the number of negative instances greatly outnumber the positive instances. In such cases, the ROC curve may not be as informative, and the precision-recall curve can provide a more accurate picture of the model's performance.

6.1.8 Confusion Matrix

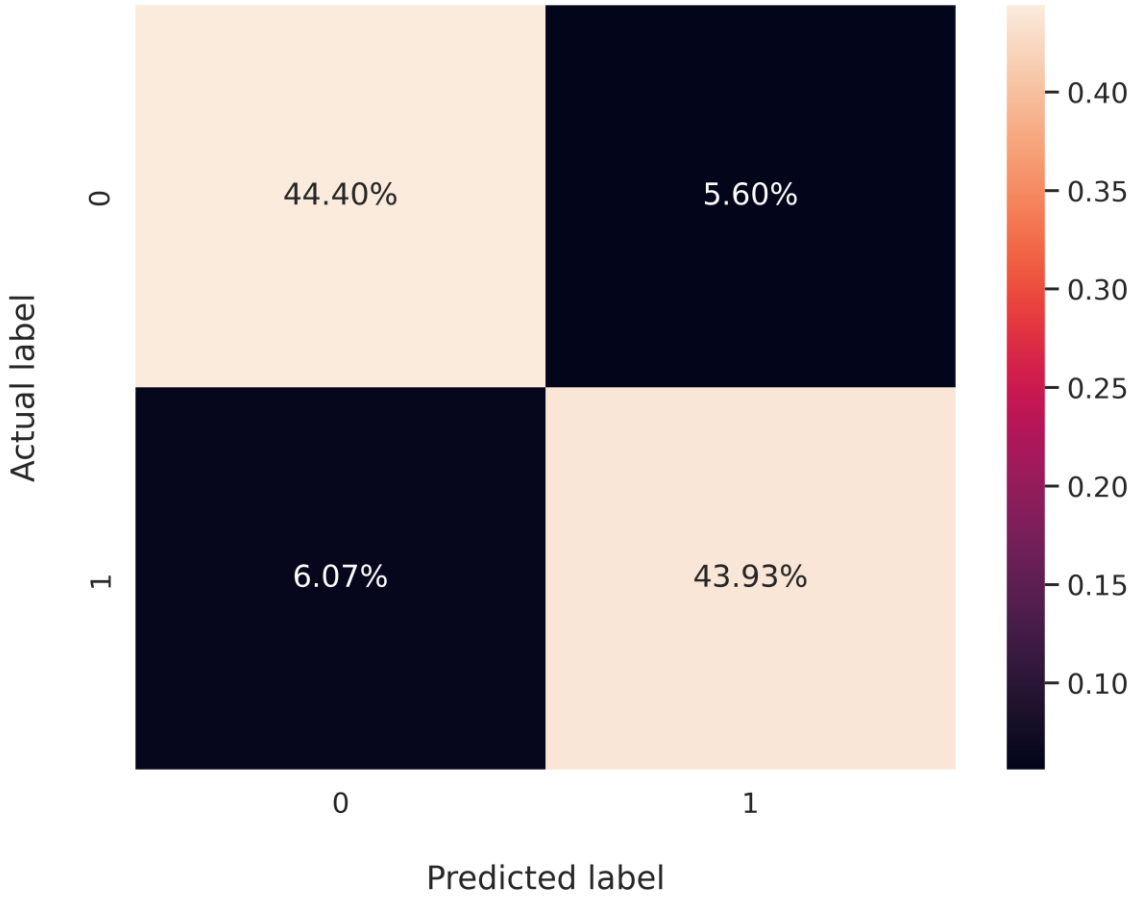

Figure 6-8: Confusion Matrix for Function Args Swap

Around 44.40% of correct code was classified as correct by our model (TP) and 43.93% of buggy code was classified as buggy by our model (TN). Around 5.60% code was correct but these models were classified as incorrect (FP) and 6.07% code was incorrect but these project's model classified it as correct (FN).

6.1.9 Hyperparameter Tuning

Table 6-1: Hyperparameter Tuning for Function Args Swap

Learning rate ($\times 10^{-3}$)	**Batch size**	**loss**	**epoch**	**accuracy**	**Precision**	**Recall**	**F1**
2	32	0.6931	1	0.49928	0.4992	1	0.6660
2	64	0.6931	1	0.49928	0.4992	1	0.6660
0.2	32	0.6931	1	0.5007	0	0	0
0.2	64	0.6931	1	0.5007	0	0	0
0.02	**32**	**0.2129**	**8**	**0.9189**	**0.9185**	**0.9194**	**0.9189**
0.02	64	0.2086	8	0.9127	0.9144	0.9105	0.9124
0.002	32	0.4109	9	0.7830	0.7875	0.7744	0.7809
0.002	64	0.3701	4	0.8062	0.7837	0.8449	0.8132

The table above displays the diverse hyperparameter values and their corresponding outcomes. It has been observed that when the learning rate is maintained at a higher level, the accuracy does not converge to the global minimum. Remarkably, the learning rate of 2×10^{-3} and 0.2×10^{-3} exhibit suboptimal performance, as the model fails to attain maximum learning and overshoots even when the batch size is altered. However, the adoption of a learning rate of 0.02×10^{-3} leads to a remarkable surge in accuracy. Moreover, when the batch size is set at 32, superior metrics are obtained as compared to the utilization of a batch size of 64. Similarly, the implementation of a learning rate of 0.002×10^{-3}, the performance of the model is not as good compared to previous results because of slow learning. Ultimately, based on the aforementioned observations, the hyperparameter combination of a learning rate of 0.02×10^{-3} and a batch size of 32 is selected, as it yields the best performance after being trained for 8 epochs. Therefore, this hyperparameter combination is deemed optimal for achieving the best possible results.

6.1.10 True Positive Prediction Analysis

As seen in the confusion matrix (Figure 6-8), the model has made 43.93% of the time's true positive prediction which is quite good in a balanced dataset. To analyze what type of code snippets have been categorized as true positive by the Transformer model, an example is presented below:

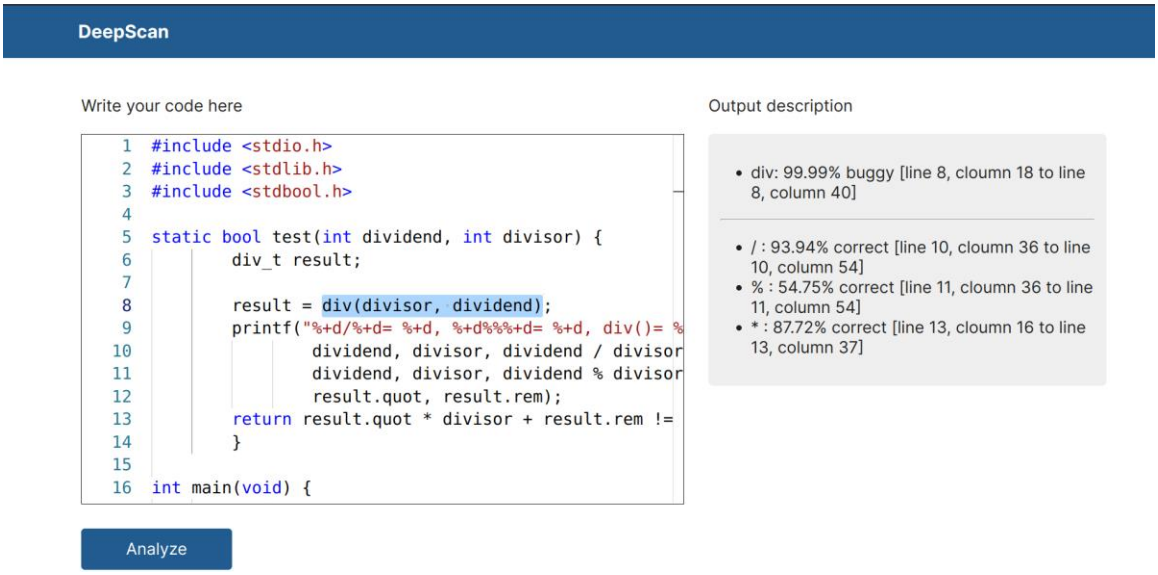

Figure 6-9: True Positive Code Snippet

The dataset extracted from code snippet in the above figure which resulted in true positive prediction is: X_{input} = (div, divisor, dividend, int, _numer, _denom), Y_{actual} = 1 (i.e. buggy) and $Y_{predicted}$ = 1 (i.e. buggy) where X_{input} is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This true positive prediction is due to the presence of names relating semantically with each other but in reverse order i.e. “div” function arguments "divisor" and "dividend" and the parameters “__numer” and “__denom” are exactly opposite with the passed arguments name semantically as in general programming scenario “divisor” is related with “__denom” and “dividend” with “__numer” in the case of division. Also, the function name “div” and the order of arguments “divisor” and “dividend” are alone sufficient for the detection of bugs by this model.

6.1.11 True Negative Prediction Analysis

As seen in the confusion matrix (Figure 6-8), the model has made 44.40% of the times true negative prediction which is quite good in a balanced dataset. To analyze what type of code snippets have been categorized as true negative by the Transformer model, an example is presented below:

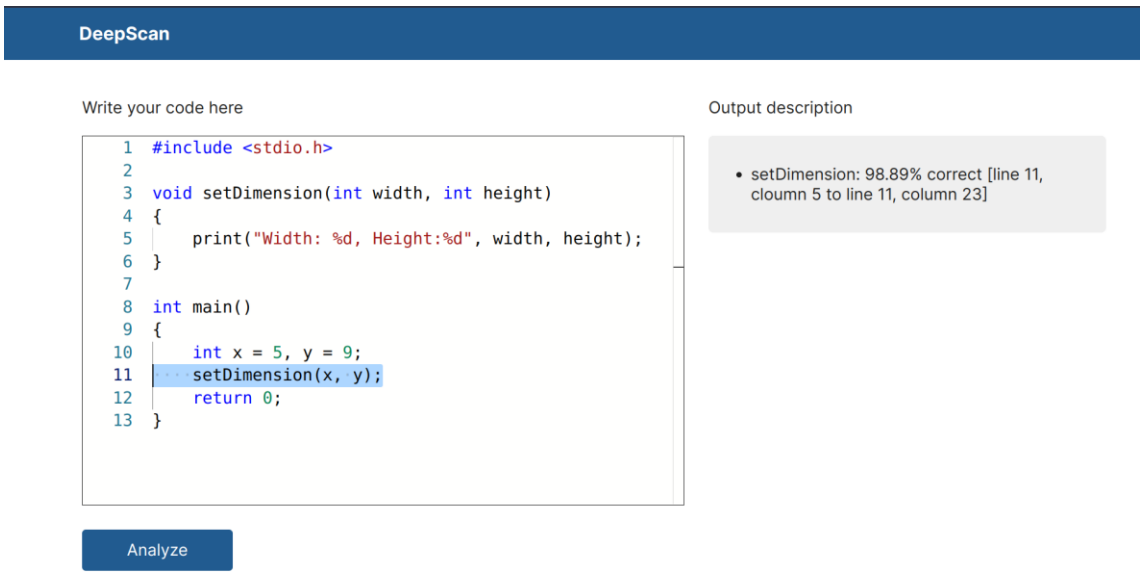

Figure 6-10: True Negative Code Snippet

The dataset extracted from code snippet in the above figure which resulted in true negative prediction is: X_{input} = (setDimension, x, y, int, width, height) , Y_{actual} = 0 (i.e. correct) and $Y_{predicted}$ = 0 (i.e. correct) where X_{input} is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This true negative prediction is due to the presence of names relating semantically with each other i.e. “setDimension” function arguments "x" and "y" and the parameters “width” and “height” which exactly match with the passed arguments name semantically as in general programming scenario “x” is related with “width” and “y” with “height” in the case of setting dimension. Also, the function name “setDimension” and the arguments “x” and “y” are alone sufficient for the correct prediction.

6.1.12 False Negative Prediction Analysis

As seen in the confusion matrix (Figure 6-8), the model has made 6.07% of the times false negative prediction which is not a bad number as it's significantly less as compared to true positive and true negative values. To analyze what type of code snippets have been categorized as false negative by the Transformer model, an example is presented below:

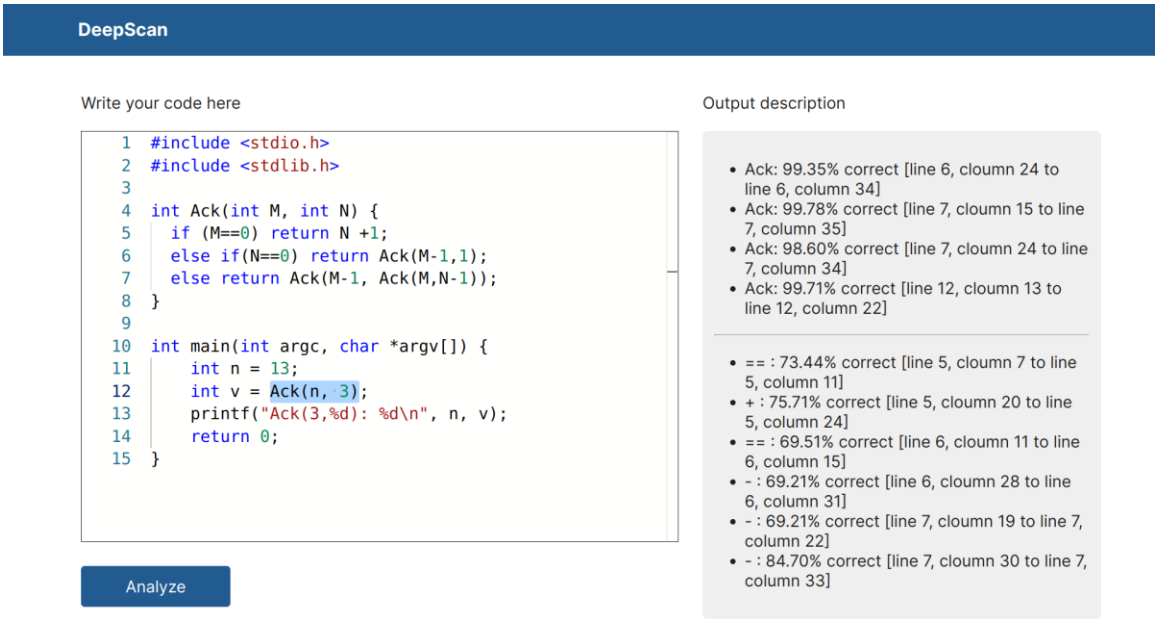

Figure 6-11: False Negative Code Snippet

The dataset extracted from code snippet in the above figure which resulted in false negative prediction is: $X_{pos} = (Ack, n, 3, int, M, N)$, $Y_{actual} = 1$ (i.e. buggy) and $Y_{predicted} = 0$ (i.e. correct) where $X_{pos}$is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$= is the predicted output value.

This false negative prediction is due to the lack of semantic meaning of arguments "3" and "n" and also the parameters which don't exactly match with the passed arguments name semantically. The function name "Ack" and the arguments "3" and "n" don't have any connection if we relate them on the basis of their name. As the ML model for detection of swapped function arguments highly depends on the names of function and variable, this input was predicted as correct even if it was buggy.

6.1.13 False Positive Prediction Analysis

As seen in the confusion matrix (Figure 6-8), the model has made 5.6% of the times false positive prediction which is not a bad number as it's significantly less as compared to true positive and true negative values. To analyze what type of code snippets have been categorized as false positive by the Transformer model, an example is presented below:

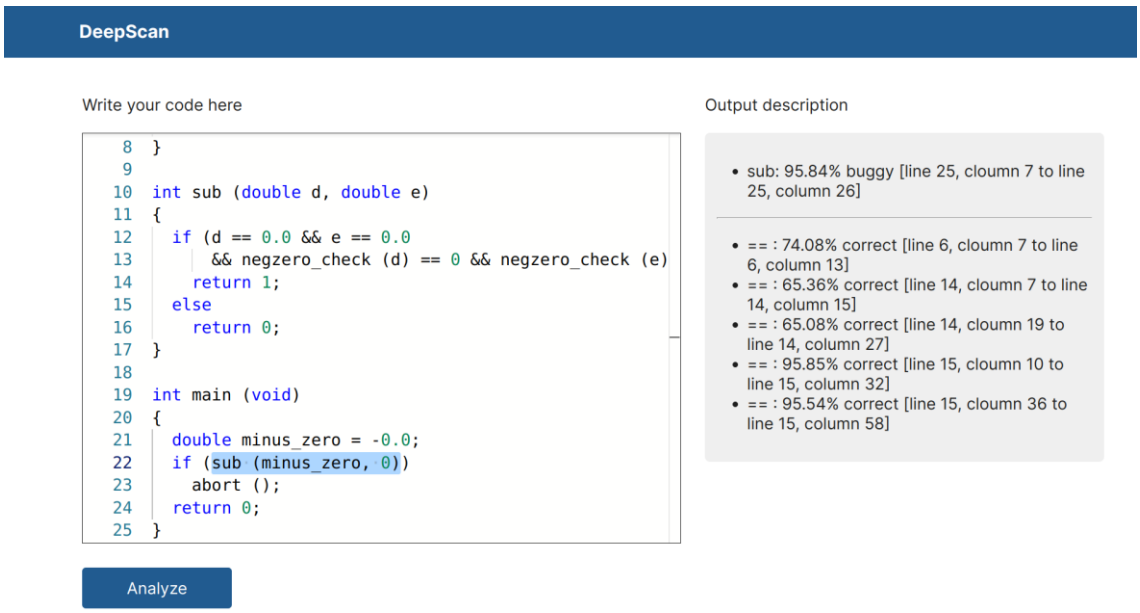

Figure 6-12: False Positive Code Snippet

The dataset extracted from code snippet in the above figure which resulted in false positive prediction is: X_{input} = (sub, minus_zero, 0, double, d, e), Y_{actual}= 0 (i.e. correct) and $Y_{predicted}$= 1 (i.e. buggy) where Xinput is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This false negative prediction can be due to the lack of semantic meaning of arguments "minus_zero" and "0" and also the parameters "d" and "e" which don't exactly match with the passed arguments name semantically. The function name "sub" is well-named and gives the meaning semantically while the arguments "minus_zero" and "0" don't have any connection with function names if we relate them on the basis of their name. As the ML model for detection of swapped function arguments highly depends on the names of function and variable, this input was predicted as buggy even if it was correct.

6.2 Detecting Swapping of Wrong Binary Operators

6.2.1 Epoch vs. Loss Plot

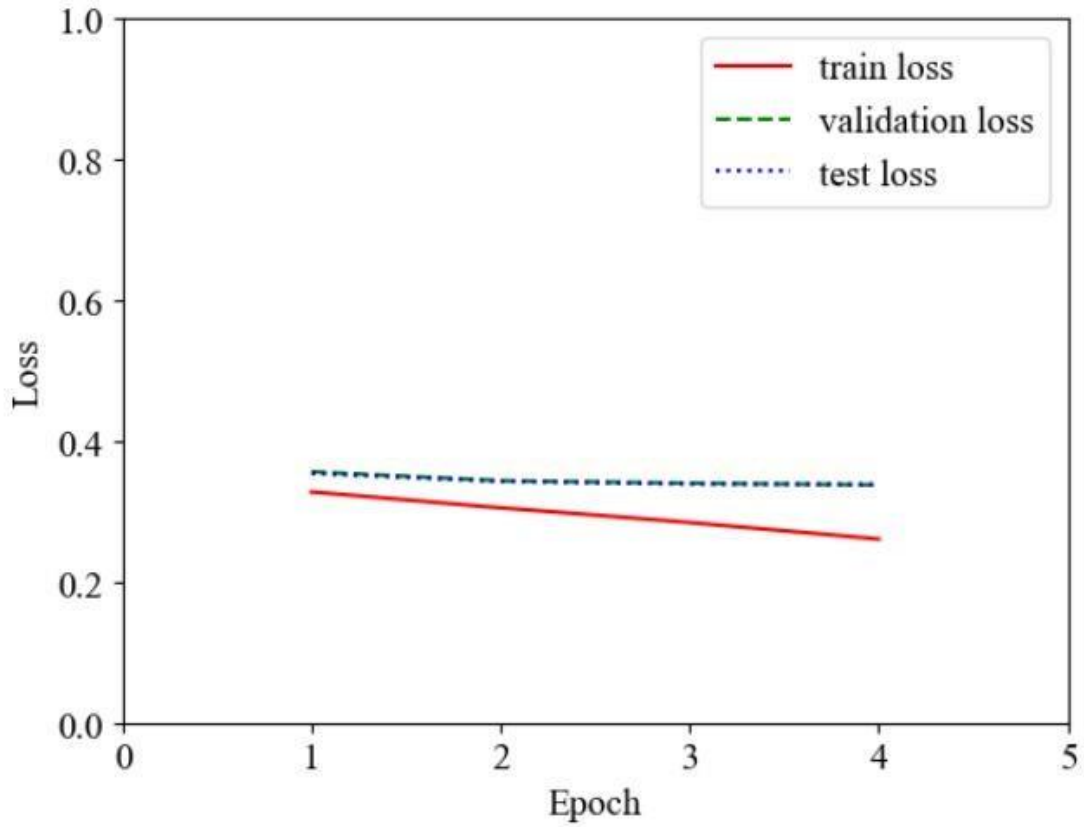

Figure 6-13: Epoch vs. Loss Plot

The model was trained only up to the 4th epoch, and early stopping was employed to prevent overfitting. This means that the model was trained only for a limited number of iterations and was stopped early to avoid the model from memorizing the training data and performing poorly on new data. The training loss decreased from 0.3276 to 0.2607 by the end of the 4th epoch. Similarly, for validation, the loss decreased from 0.356363 to 0.337736, and for testing, the loss decreased from 0.354083 to 0.338415.

6.2.2 Epoch vs. Accuracy Plot

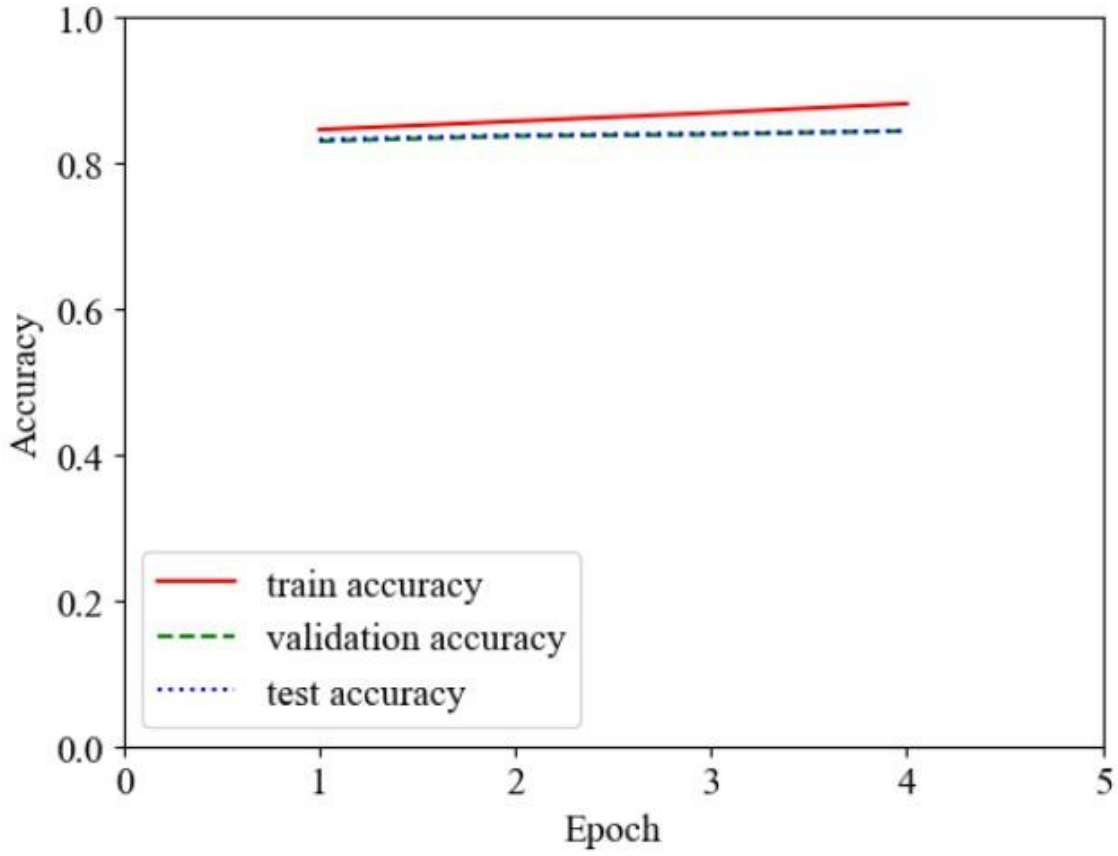

Figure 6-14: Epoch vs. Accuracy Plot

As for the incorrect utilization of the binary operator, the model undergoes training until the fourth epoch, whereby the early stopping mechanism is implemented to prevent overfitting. The accuracy of the training set displays an increase from 0.845748 to 0.881418 at the culmination of the fourth epoch, and the validation and test sets indicate an improvement in accuracy from 0.830015 and 0.832050 to 0.843945 and 0.844020, respectively.

6.2.3 Epoch vs. Precision Plot

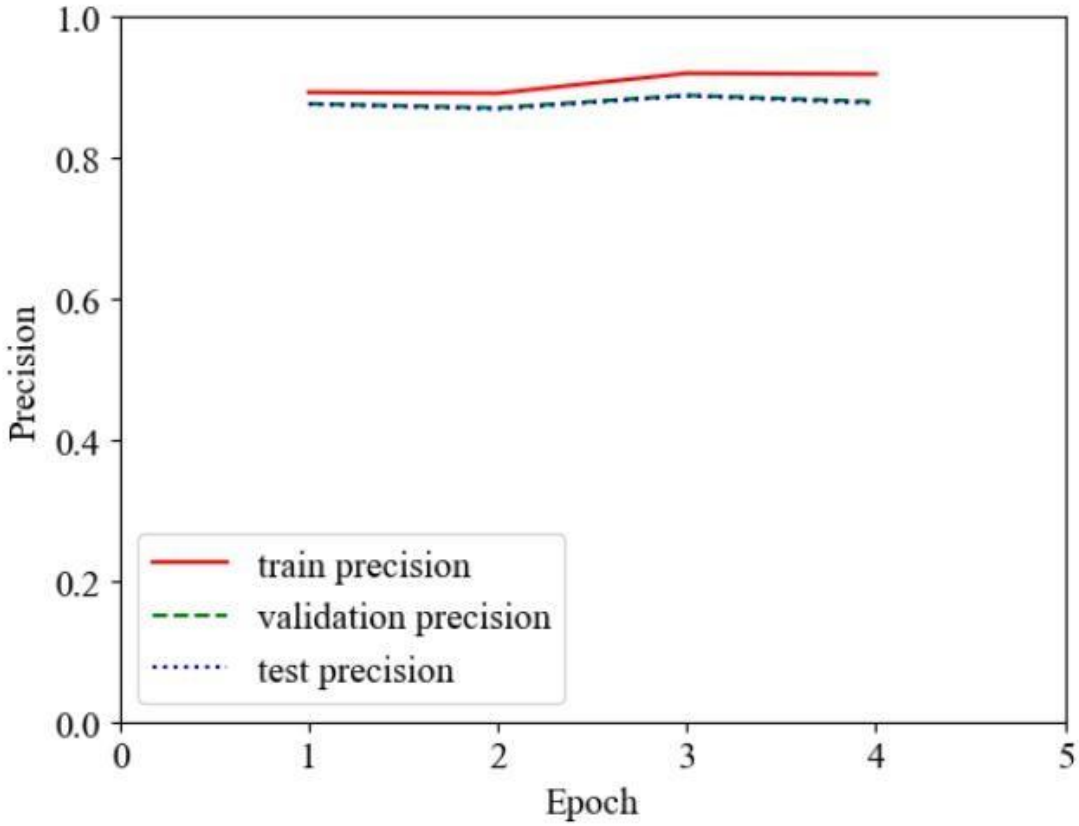

Figure 6-15: Epoch vs. Precision Plot

With regard to the Wrong Binary operator, it appears that the model's training leads to a significant increase in precision by the conclusion of the fourth epoch. This indicates that the model is able to learn from the data it is being fed, resulting in more accurate predictions. Additionally, the fact that both validation and test precision also increase suggests that the model is capable of generalizing to new data.

Overall, these results suggest that the model is capable of learning and generalizing well, which is essential for its success in various real-world applications.

6.2.4 Epoch vs. Recall Plot

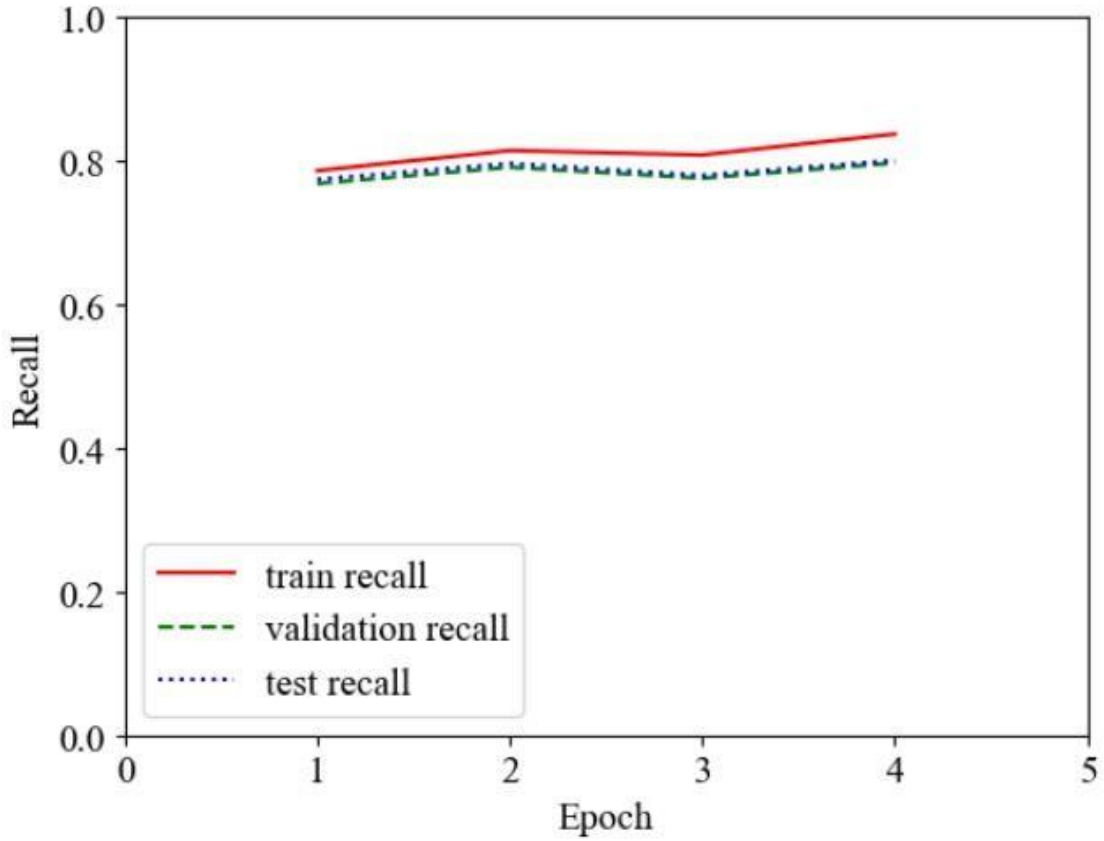

Figure 6-16: Epoch vs. Recall Plot

In contrast, for the Wrong Binary operator function, the model was only trained until the 4th epoch, followed by the application of early stopping measures to prevent overfitting. This approach proved to be effective, as the recall score for the training set demonstrated a notable increase, rising from 0.786018 to 0.837181 by the end of the 4th epoch. The validation and test recall scores also exhibited a similar upward trend, with values increasing from 0.768072 and 0.773777 to 0.796842 and 0.800432, respectively. These results imply that the model was able to learn the task well within a limited amount of time, and the early stopping technique helped avoid the problem of overfitting, thereby improving the model's generalization ability.

6.2.5 Epoch vs. F1 Plot

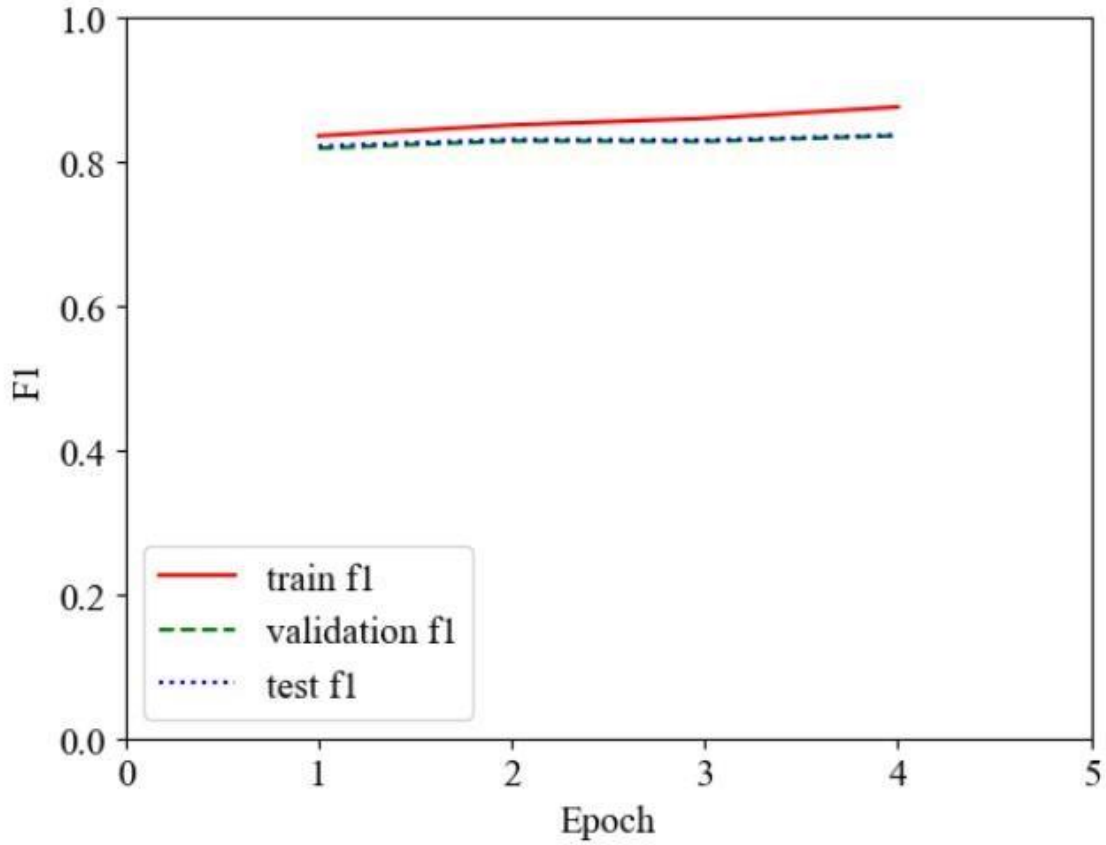

Figure 6-17: Epoch vs. F1 Plot

With respect to the Wrong Binary operator, it appears that the model undergoes training solely up to the 4th epoch, with early stopping techniques being implemented to avoid overfitting. Within this limited training window, the F1 score for the training data set exhibits a visible increase, ascending from 0.835966 to 0.875943 by the conclusion of the 4th epoch. Moreover, the F1 scores for both the validation and test data sets display a marked improvement, increasing from 0.818641 and 0.821658 to 0.836093 and 0.836911, respectively.

In essence, the empirical findings suggest that the aforementioned alterations to programmatic functions can lead to statistically significant improvements in the performance of machine learning models. The present study serves to contribute to a growing body of literature concerning the optimization of these models, and it is hoped

that the information conveyed herein may be of utility to practitioners and researchers alike.

6.2.6 ROC Curve

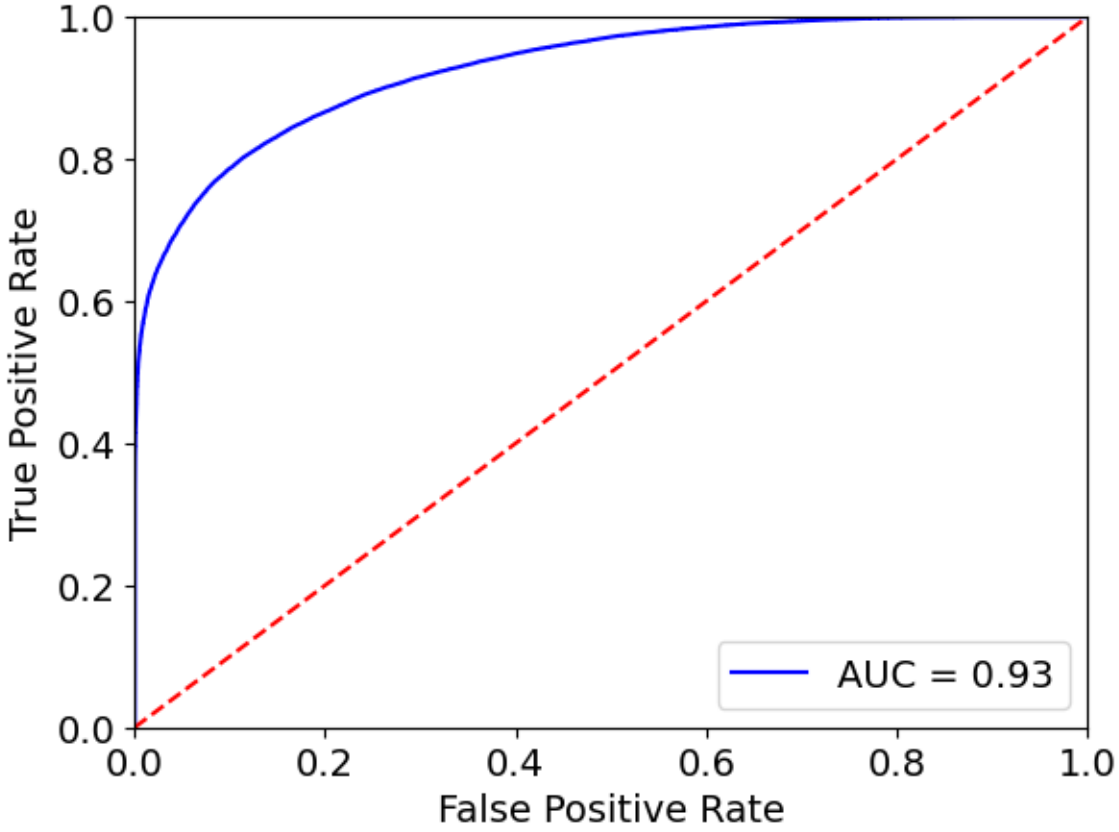

Figure 6-18: ROC curve of Wrong Binary Operators

An AUC score of 0.93 suggests that the model has a high discriminatory power and is able to distinguish between positive and negative samples with high accuracy. Specifically, it means that the model is able to correctly rank 93% of the positive samples higher than the negative samples, across all possible thresholds. It indicates that the model is reliable and accurate.

6.2.7 Precision-Recall Curve

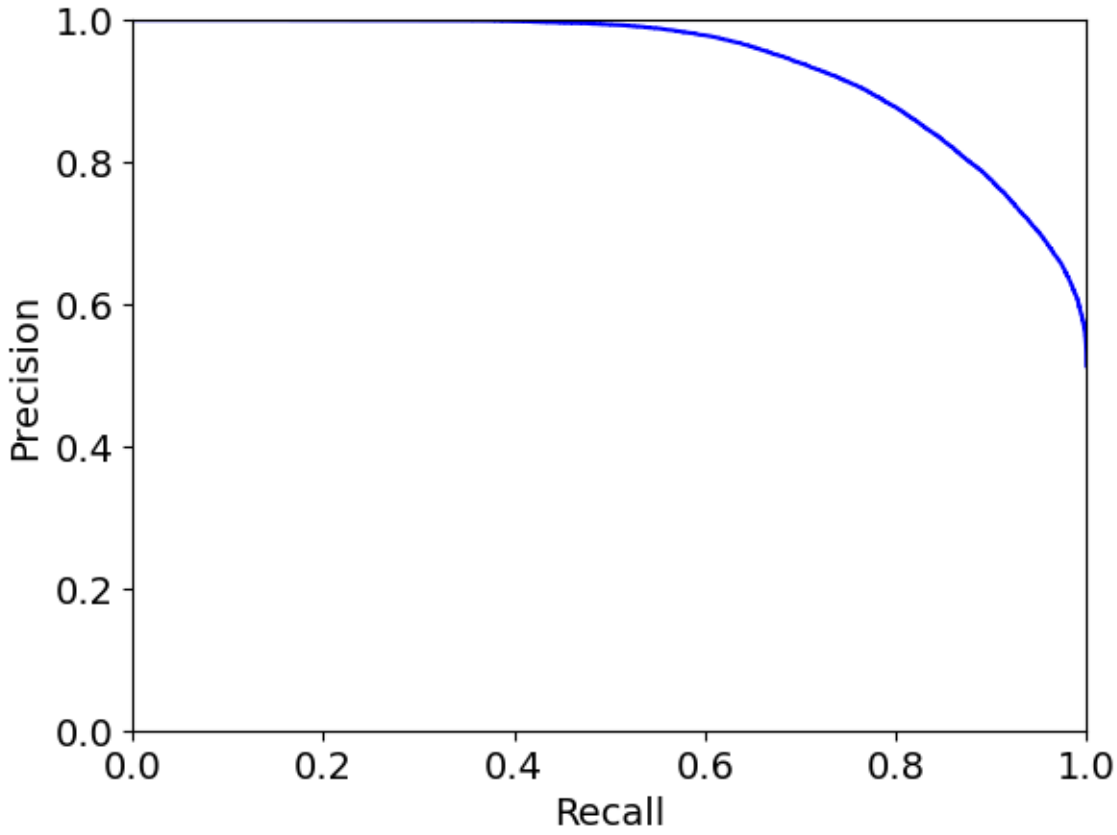

Figure 6-19: Precision-Recall Curve of Wrong Binary Operator

The above precision-recall curve shows how well the model is able to correctly identify positive instances (precision) while minimizing the number of false negatives (recall). The curve is closer to the top-right corner of the plot. Hence, the performance of the model is better.

The precision-recall curve is particularly useful when the classes are imbalanced, that is, when the number of negative instances greatly outnumber the positive instances. In such cases, the ROC curve may not be as informative, and the precision-recall curve can provide a more accurate picture of the model's performance.

6.2.8 Confusion Matrix

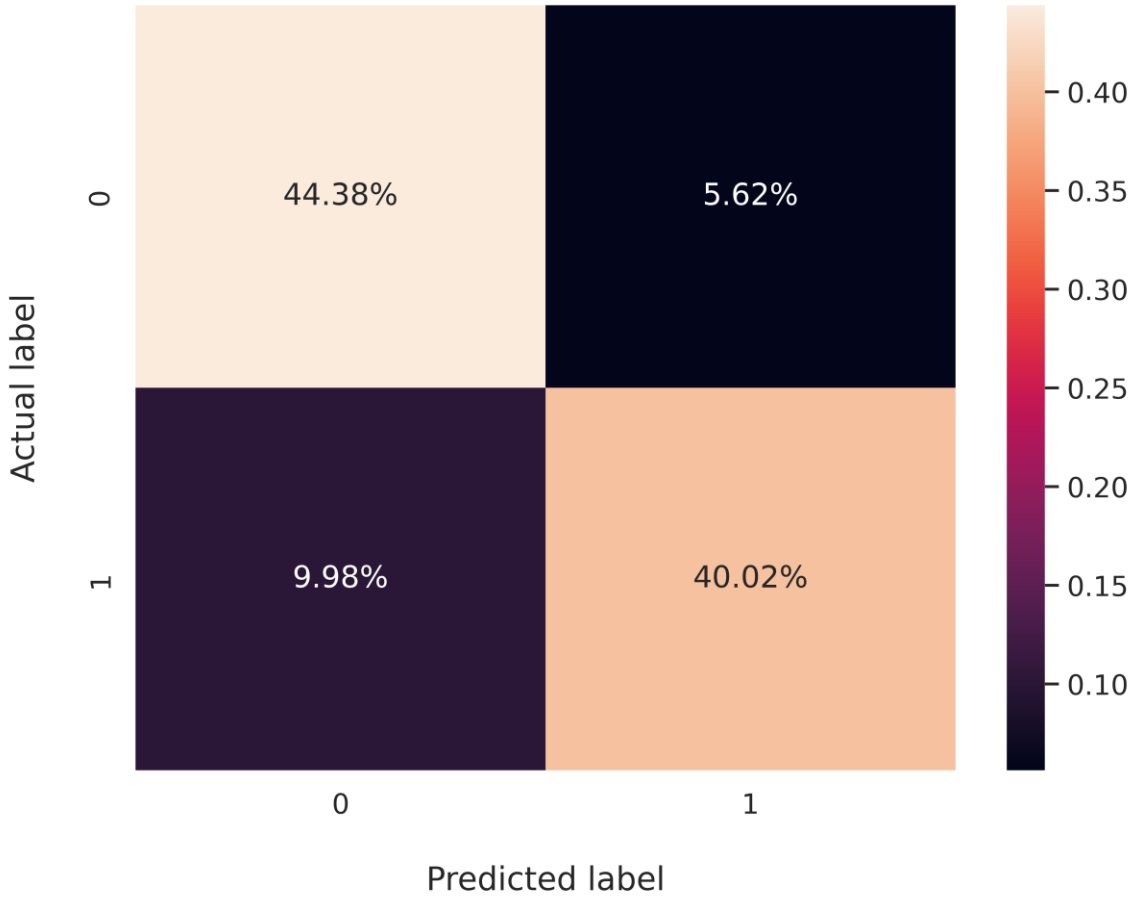

Figure 6-20: Confusion Matrix for Wrong Binary Operator

Around 44.38% of correct code was classified as correct by our model (TP) and 40.02% of buggy code was classified as buggy by our model (TN). Around 5.62% code was correct but these models were classified as incorrect (FP) and 9.98% code was incorrect but these project's model classified it as correct (FN).

6.2.9 True Positive Prediction Analysis

As seen in the confusion matrix (Figure 6-20), the model has made 40.02% of the time true positive prediction which is quite good in a balanced dataset. To analyze what type of code snippets have been categorized as true positive by the Transformer model, an example is presented below:

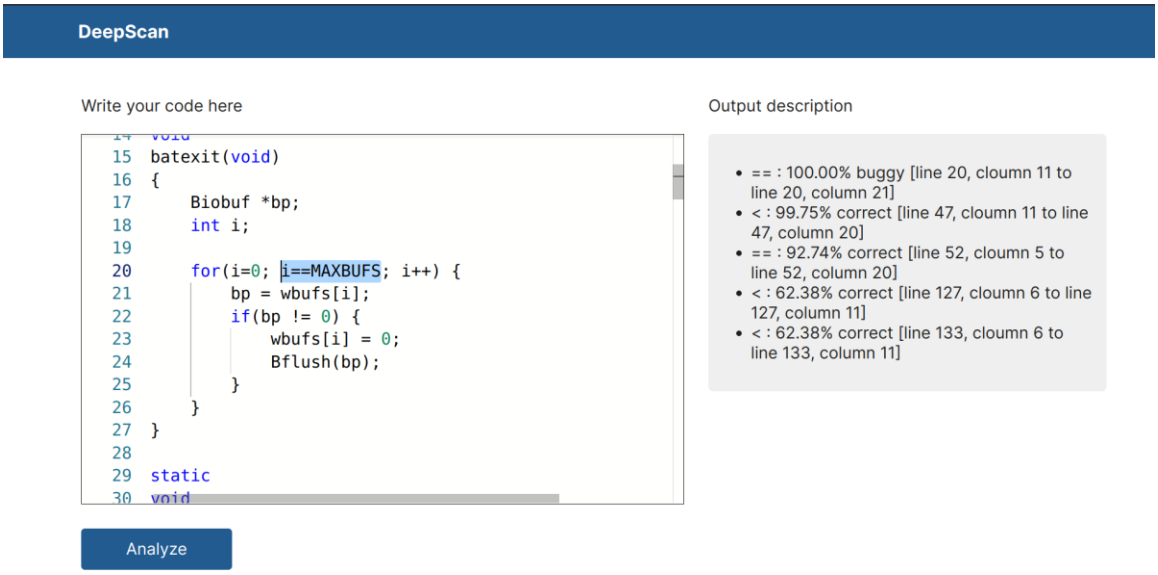

Figure 6-21: True Positive Code Snippet

The dataset extracted from code snippet in the above figure which resulted in true positive prediction is: X_{input} = (i, ==, MAXBUFS, int, int, FOR_STMT, COMPOUND_STMT),), Y_{actual} = 1 (i.e. buggy) and $Y_{predicted}$ = 1 (i.e. buggy) where X_{input} is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This true positive prediction is due to the presence of "==" operator in the condition statement of for loop which is unusual between the operands named "i" and "MAXBUFS".

6.2.10 True Negative Prediction Analysis

As seen in the confusion matrix (Figure 6-20), the model has made 44.38% of the time true negative prediction which is quite good in a balanced dataset. To analyze what type of code snippets have been categorized as true positive by the Transformer model, an example is presented below:

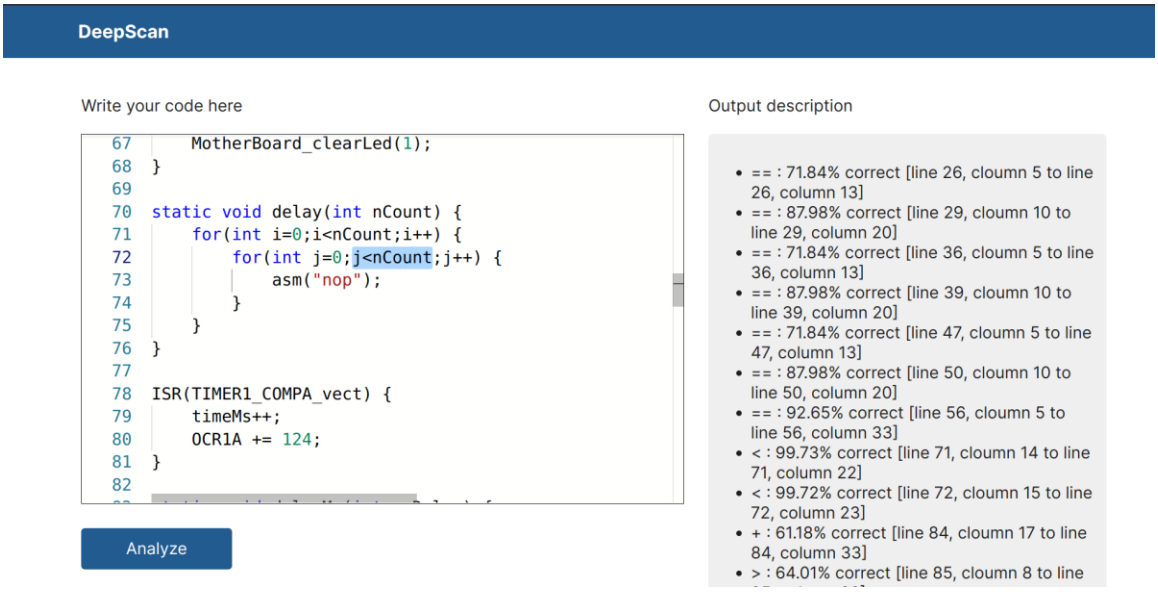

Figure 6-22: True Negative Code Snippet

The dataset extracted from code snippet in the above figure which resulted in true negative prediction is: $X_{input} = (j, <, nCount, int, int, FOR\_STMT, COMPOUND\_STMT)$, $Y_{actual} = 0$ (i.e. correct) and $Y_{predicted} = 0$ (i.e. correct) where X_{input} is the input to the model, Yactual is the actual output value that the model should predict and $Y_{predicted} =$ is the predicted output value.

This true negative prediction is due to the presence of names relating semantically with each other i.e. "<" operator between operands "j" and "nCount" which is a common and correct statement and also due to the presence of "FOR_STMT" (i.e. for statement) as a parent which gives it more context.

6.2.11 False Negative Prediction Analysis

As seen in the confusion matrix (Figure 6-20), the model has made 9.98% of the times false negativee prediction which is not a bad number as it's significantly less as compared to true positive and true negative values. To analyze what type of code snippets have been categorized as false negative by the Transformer model, an example is presented below:

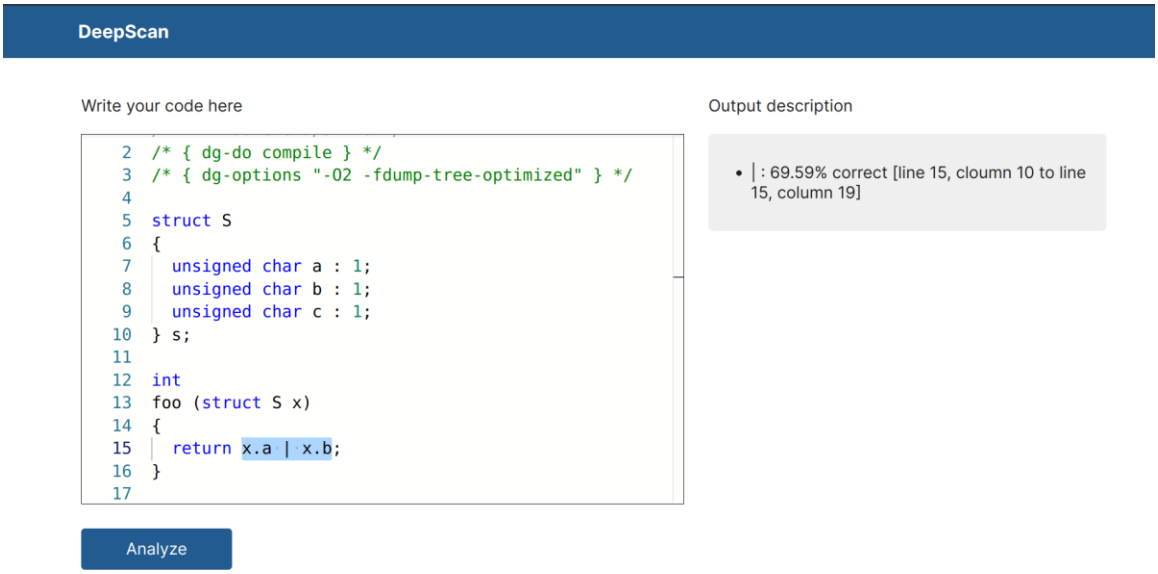

Figure 6-23: False Negative Code Snippet

The dataset extracted from code snippet in the above figure which resulted in false negative prediction is: $X_{pos} = (j, x.a, |, x.b, int, int, BINARY\_OPERATOR, RETURN\_STMT)$, $Y_{actual} = 1$ (i.e. buggy) and $Y_{predicted} = 0$ (i.e. correct) where X_{pos} is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This false negative prediction can be due to the lack of semantic meaning of operands "x.a" and "x.b" between the operator "|". As the ML model for detection of wrong binary operand highly depends on the names of variables, this input was predicted as correct even if it was buggy.

6.2.12 False Positive Prediction Analysis

As seen in the confusion matrix (Figure 6-20), the model has made 5.62% of the times false positive prediction which is not a bad number as it's significantly less as compared to true positive and true negative values. To analyze what type of code snippets have been categorized as false positive by the Transformer model, an example is presented below:

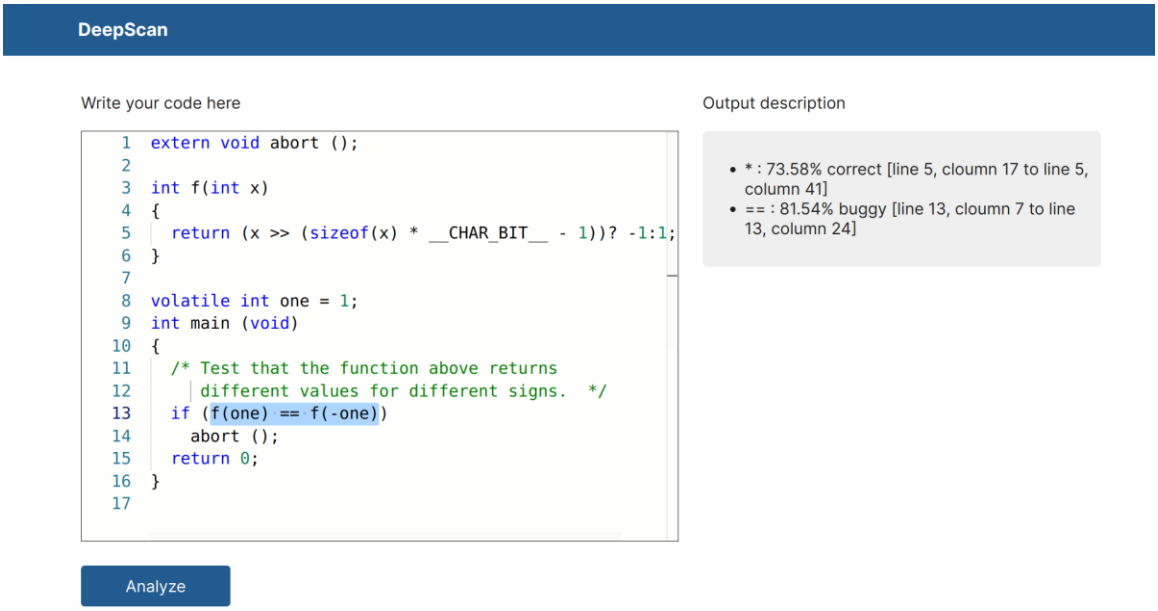

Figure 6-24: False Positive Code Snippet

The dataset extracted from code snippet in the above figure which resulted in false positive prediction is: $X_{pos} = (f(one), ==, f(-one), int, int, IF\_STMT, COMPOUND\_STMT)$, Y_{actual}= 0 (i.e. correct) and $Y_{predicted}$= 1 (i.e. buggy) where X_{pos} is the input to the model, Y_{actual} is the actual output value that the model should predict and $Y_{predicted}$ is the predicted output value.

This false positive prediction can be due to the lack of semantic meaning of operands "f(one)" and "f(-one)" between the operator "==". As the ML model for detection of wrong binary operand highly depends on the names of variables, this input was predicted as buggy even if it was correct.

6.3 Detection of Mixed Types of Bugs

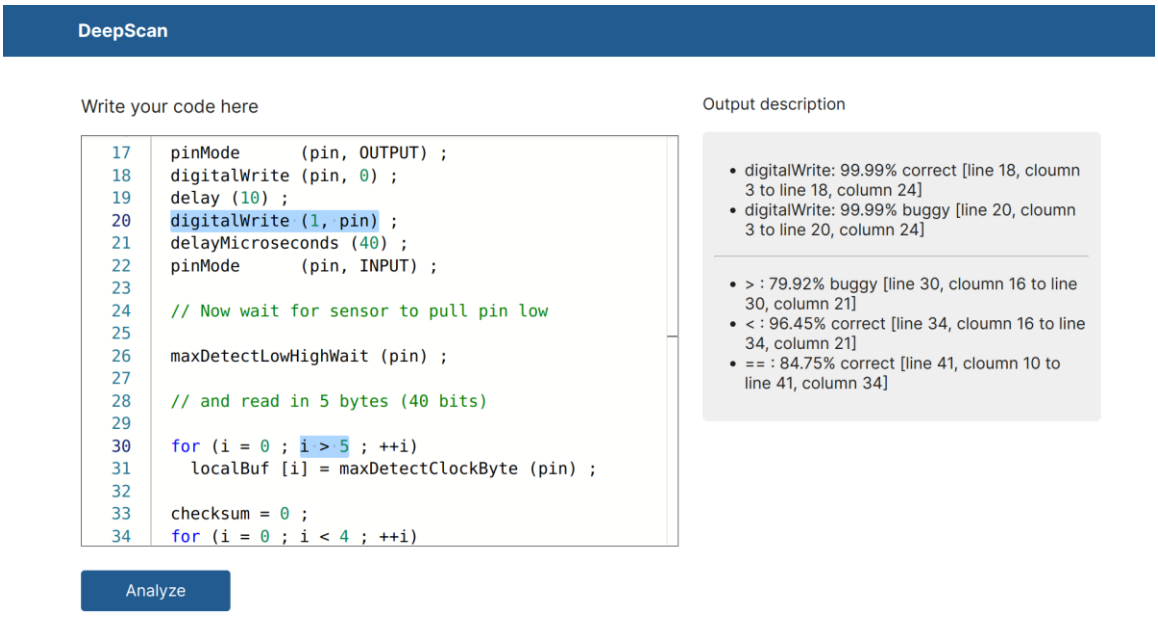

Figure 6-25: Example of Both Types of Bugs

The project can also detect multiple types of bugs in the user's source code at the same time. As shown in the above figure, bug related to swapped function arguments (line 20, column 3 to line 20, column 24) and wrong binary operator (line 30, column 16 to line 30, column 21) have been evaluated and shown as at the same time in the output. In the backend of web application, the results from the two models are concatenated and sent to the client-side web application making it possible to analyze multiple types of bugs.

6.4 Comparison of Performance with Similar Projects

Table 6-2: Comparision of Accuracy with Similar Projects

Type of Bug	DeepBugs		This Project
	Random Embedding	Learned Embedding	Learned Embedding
Function Arguments Swap	93.88%	94.70%	91.89%
Wrong Binary Operator	89.15%	92.21%	84.39%

In the above table, the validation accuracy of DeepBugs [6] and this project is being compared. DeepBugs project consists the validation accuracy of random and learned

embedding where random embedding has low accuracy as compared to learned embedding. DeepBugs had done name-based bug detection in JavaScript which is a dynamically typed language but this project is based on detection of name-based bugs in C language which is a statically typed language. As these two projects have significant difference, the comparision of metrices may not give the exact difference. In comparision with DeepBugs, this project has 2.81% and 7.82% less accuracy in swapped function arguments model and wrong binary operator model respectively.

7. FUTURE ENHANCEMENT

7.1 Exploring Other ML Models

In this project, DistilBERT model for only two types of named based bugs (i.e. function arguments swapped bug and wrong binary operator bug) have been implemented. There are still some areas where improvement is needed. The F1 score of the two models are 91.89% and 83.60% respectively on validation dataset and 88.12% and 83.69% respectively on test dataset which can be improved in the future. Several other architectures such as BERT, RoBERTa, etc. can be explored to see if there is an enhancement in the performance and the changes can be addressed subsequently.

7.2 Introduction of Other Types of Name Based Bugs

Transformer based models can also be developed for other types of name based bugs which can be done following the similar steps taken in developing current models. Some other name based bugs can be wrong operator precedence, wrong binary operands, etc. After acquiring profound insights on such types of bugs, next strides can be taken to detect these types of name based bugs.

7.3 Detection of Bugs in Multiple Programming Languages

This project can be further extended to include name based bugs detection in programming languages other than C. It will be relatively easier to do so in other programming languages as the major concepts and steps of name based bug detection have been already implemented in this project. The core concept of AST can be applied across the multiple programming languages. Thus, a smooth transitioning of incorporating name based bugs into multiple programming languages is highly feasible.

8. CONCLUSION

A suitable training data consisting of correct and buggy code was successfully created. Around 720,000 code snippets of C language were taken from C Code Corpus [13] dataset. The dataset was raw and required plenty of operations prior to training the data. The dataset had to be dealt with outliers as it was superfluous to generate Abstract Syntax Trees (ASTs) for the code snippets. Duplicated and missing data were handled. In this way, positive sample was created. AST's were generated for each correct code and the negative sample was generated respectively. The buggy code was different for both of the Function arguments swapped bug and Wrong binary operator bug as both had distinct ways to generate negative sample from the positive sample i.e. correct code. Eventually, the data was ready for training and further proceedings.

DistilBERT model was trained seperately for the detection of swapped function arguments bug and wrong binary operators bug incorporating a fine-tuned CodeT5 tokenizer. The CodeT5 tokenizer was fine-tuned on C Code Corpus [13] dataset which made it more efficient on C language codes. The models were trained using Nvidia T4 GPU from Kaggle [14] and F1 score of the two models are 91.89% and 83.60% respectively on validation dataset and 88.12% and 83.69% respectively on test dataset. The model is capable of detecting two types of name-related bugs during static code analysis. The steps and techniques used in this project can be utilized to easily develop Machine Learning models for other types of name-based bugs.

9. APPENDICES

9.1 APPENDIX A: PROJECT SCHEDULE

Table 9-1: Gantt Chart with Project Activities and Timeline

Project Start Date	7-Dec-22											Project End Date	7-Mar-23
		Week											
Task	Progress(%)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Literature Review	100														
Dataset Preparation	100														
- Function Arguments Swap															
- Wrong Binary Operators															
Training Model	100														
- Function Arguments Swap															
- Wrong Binary Operators															
Model Implementation	100														
- Function Arguments Swap															
- Wrong Binary Operators															
Website Development	100														
Testing & Debugging	100														
System Implementation	100														
Documentation	100														
Task Completion:	100														

9.2 APPENDIX B: CODE SNIPPETS

```
def get_function_params(root, function_name, result):
    """
    A function to get details of function parameter from Abstract Syntax Tree

    Parameters
    ----------
    root : clang.cindex.Cursor
        root of Abstract Syntax Tree
    function_name : str
        name of function whose parameters' detail is needed
    result : list
        list to store the parameter details

    Return
    ------
    void
    """

    for node in root.walk_preorder():
        try:
            """
            checking if AST node belongs to function declaration
            and its name is same as passed function_name
            """
            if node.kind == CursorKind.FUNCTION_DECL \
            and node.spelling == function_name:
                # loop through its children and only append details of parameter node
                for c in node.get_children():
                    if c.kind == CursorKind.PARM_DECL:
                        result.append({"name": c.spelling,
                                       "data_type": c.type.spelling})
                return
        except ValueError as e:
            print("Error:", e)
```

Figure 9-1: get_function_params Function for Obtaining Parameters

```
class InputItem(BaseModel):
    code: str = None

# POST request endpoint for "/analyze" path
@app.post("/analyze")
async def analyze(inputItem: InputItem):
    code = inputItem.code

    # get cursor of root node of AST
    ast_root_cursor = generate_ast(code)
    # get prediction for function args swap bug
    fasb_prediction = generate_fasb_prediction(ast_root_cursor)
    # get prediction for wrong binary operator bug
    wbob_prediction = generate_wbob_prediction(ast_root_cursor)

    # converting output to python dictionary for JSON conversion
    output = {"analysis": {
        "function_args_swap_bug": fasb_prediction,
        "wrong_binary_operator_bug": wbob_prediction
    }}

    # sending JSON response to client-side app
    return JSONResponse(content=str(output))
```

Figure 9-2: RESTful API Endpoint Function for Receiving Post Request

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

static bool test(int dividend, int divisor) {
        div_t result;

        result = div(divisor, dividend);
        printf("%+d/%+d= %+d, %+d%%%+d= %+d, div()= %+d, %+d\n",
                dividend, divisor, dividend / divisor,
                dividend, divisor, dividend % divisor,
                result.quot, result.rem);
        return result.quot * divisor + result.rem != dividend;
        }

int main(void){
        bool t;

        printf("\nTest of division and modulus operations:\n\n");
        t =     test(+40, +3) ||
                test(+40, -3) ||
                test(-40, +3) ||
                test(-40, -3);
        if (t)
                printf("\nThe div() function made a wrong result!\n");

        printf("\nTap a key, to exit. ");
        getchar();
        return (int)t;
        }
```

Figure 9-3: True Positive Code Snippet of Function Args Swap Bug

```c
#include <stdio.h>
#include <stdlib.h>

int Ack(int M, int N) {
  if (M==0) return N +1;
  else if(N==0) return Ack(M-1,1);
  else return Ack(M-1, Ack(M,N-1));
}

int main(int argc, char *argv[]) {
    int n = 13;
    int v = Ack(n, 3);
    printf("Ack(3,%d): %d\n", n, v);
    return 0;
}
```

Figure 9-4: False Negative Example of Function Args Swap Bug

References

[1] M. Allamanis et al., “Self-Supervised Bug Detection and Repair.”, 2021. *Arxiv*, https://arxiv.org/abs/2105.12787. Accessed 3 Jan 2023.

[2] S. Chakraborty et al., “Deep Learning-based Vulnerability Detection: Are We There Yet?”, 2021. *Arxiv*, https://arxiv.org/abs/2009.07235. Accessed 5 Jan 2023.

[3] Jannik Pewny and Thorsten Holz, “EvilCoder: Automated Bug Insertion.”, 2020. *Arxiv*, https://arxiv.org/abs/2007.02326. Accessed 29 December 2022.

[4] R. Karampatsis *et al.*, “Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code.”, 2020. https://arxiv.org/abs/2003.07914. Accessed 25 December 2022.

[5] Y. Li *et al.*, “Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks.”, 2019. *ACM Digital Library*, https://dl.acm.org/doi/10.1145/3360588. Accessed 30 December 2022.

[6] Michael Pradel and Koushik Sen, “DeepBugs: A Learning Approach to Name-based Bug Detection.”, 2018, https://arxiv.org/abs/1805.11683. Accessed 30 December 2022.

[7] R. Gupta *et al.*, “Deep Fix: Fixing Common C Language Errors by Deep Learning.”, 2017. *Research Gate*, https://www.researchgate.net/publication/361536260_DeepFix_Fixing_Common_C_Language_Errors_by_Deep_Learning. Accessed 11 January 2023.

[8] J. Devlin *et al.*, “Semantic Code Repair using Neuro-Symbolic Transformation Networks.”, 2017. *Arxiv*, https://arxiv.org/abs/1710.11054. Accessed 10 January 2023.

[9] Y. Wang *et al.*, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.” 2021. *Arxiv*, https://arxiv.org/abs/2109.00859. Accessed 27 January 2023.

[10] A. Vaswani *et al.*, “Attention Is All You Need” 2017. *Arxiv*, https://arxiv.org/abs/1706.03762. Accessed 22 December 2022.

[11] Dan Hendrycks and Kevin Gimpel, “Gaussian Error Linear Units (GELUs)”, 2016. *Arxiv*, https://arxiv.org/abs/1606.08415. Accessed 15 January 2023

[12] Diederik P. Kingma, Jimmy Ba, “Adam: A Method for Stochastic Optimization”, 2014. *Arxiv*, https://arxiv.org/abs/1412.6980. Accessed 12 January 2023

[13] Rafael - Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Suttonand Andrea Janes, “Raw C Code Corpus”. Zenodo, Jan. 27, 2020. doi: 10.5281/zenodo.3628775

[14] "Kaggle: Your Machine Learning and Data Science Community", https://www.kaggle.com. Accessed 21 December, 2023.

[15] C. Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”, 2019. *Arxiv*, https://arxiv.org/abs/1910.10683. Accessed 3 January 2023.