

# **‘Identification of Redundant Code using AI’**

A major project report submitted in partial fulfilment of the requirement  
for the award of degree of

**Bachelor of Technology**

in

**Computer Science & Engineering / Information Technology**

*Submitted by*

**Piyush Joshi(211397), Samriti Thakur(211150),**

**Prakhar Varshney(211327).**

*Under the guidance & supervision of*

**Dr. Ramesh Narwal**

**Ms. Seema Rani**

**Assistant Professor CSE&IT**

**Assistant Professor CSE&IT**



**Department of Computer Science & Engineering and  
Information Technology**


**Jaypee University of Information Technology, Waknaghat,  
Solan - 173234 (India)**

**May 2025**

## Supervisor's Certificate

This is to certify that the major project report entitled '**Identification of Redundant Code using AI**', submitted in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science & Engineering**, in the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, is a bona fide project work carried out under my supervision during the period from July 2024 to May 2025.

I have personally supervised the research work and confirm that it meets the standards required for submission. The project work has been conducted in accordance with ethical guidelines, and the matter embodied in the report has not been submitted elsewhere for the award of any other degree or diploma.



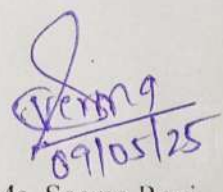
Dr. Ramesh Narwal

Assistant Professor

CSE&IT

Date: 09/05/2025

Place: JUIT



Ms. Seema Rani

Assistant Professor

CSE&IT

Date: 09/05/2025

Place: JUIT

## Candidate's Declaration

We hereby declare that the work presented in this major project report entitled '**Identification of Redundant Code using AI**', submitted in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science & Engineering**, in the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, is an authentic record of our own work carried out during the period from July 2024 to May 2025 under the supervision of **Dr. Ramesh Narwal and Ms. Seema Rani**.

We further declare that the matter embodied in this report has not been submitted for the award of any other degree or diploma at any other university or institution.

Piyush Joshi

Name: Piyush Joshi

Roll No.: 211397

Date: 09/05/2025

Samriti

Name: Samriti Thakur

Roll No.: 211150

Date: 09/05/2025

Prakhar Varshney

Name: Prakhar Varshney

Roll No.: 211327

Date: 09/05/2025

This is to certify that the above statement made by the candidates is true to the best of my knowledge.

Dr. Ramesh Narwal  
09/05/2025

Assistant Professor

CSE&IT

Date: 09/05/2025

Place: JUIT

Seema Rani  
09/05/25

Ms. Seema Rani

Assistant Professor

CSE&IT

Date: 09/05/2025

Place: JUIT

## ACKNOWLEDGEMENT

Firstly, I express my heartiest thanks and gratefulness to almighty God for His divine blessing makes it possible for us to complete the project work successfully.

We are really grateful and wish our profound indebtedness to our project supervisors **Dr. Ramesh Narwal** and **Ms. Seema Rani**, Department of CSE Jaypee University Of Information Technology, Waknaghat. Deep Knowledge & keen interest of our supervisors guided us all along in our project work titled '**Identification of Redundant Code using AI**'. Their endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism and valuable advice along with reading many inferior drafts and correcting them at all stages have made it possible for us to complete this project.

We would like to express our heartiest gratitude to **Dr. Ramesh Narwal** and **Ms. Seema Rani**, Department of CSE, for their kind help in our project.

We would also generously welcome each one of those individuals who have helped us straight forwardly or in a roundabout way in making this project a win. In this unique situation, we might want to thank the various staff individuals, both educating and non-instructing, who have developed their convenient help and facilitated our undertaking.

Finally, we must acknowledge with due respect the constant support and patience of our parents.

Name: Piyush Joshi

Roll No.: 211397

Name: Samriti Thakur

Roll No.: 211150

Name: Prakhar Varshney

Roll No.: 211327

# TABLE OF CONTENTS

<b>LIST OF FIGURES</b>	<b>VI</b>
<b>LIST OF TABLES</b>	<b>VII</b>
<b>LIST OF ABBREVIATIONS</b>	<b>VIII</b>
<b>ABSTRACT</b>	<b>IX</b>

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
1.1	Introduction .....	1
1.2	Problem Statement .....	2
1.3	Objectives.....	4
1.4	Significance and motivation of the project work .....	6
	1.4.1 Significance .....	6
	1.4.2 Motivation .....	7
1.5	Organization of project report .....	9
<b>2</b>	<b>LITERATURE SURVEY .....</b>	<b>12</b>
2.1	Overview of relevant literature .....	12
2.2	Key gaps in the literature .....	13
2.3	A summary of the relevant papers .....	15
<b>3</b>	<b>SYSTEM DEVELOPMENT .....</b>	<b>20</b>
3.1	Requirements and Analysis .....	20
	3.1.1 Functional Requirements .....	20
	3.1.2 Non-Functional Requirements .....	21
	3.1.3 Hardware Requirements .....	22
	3.1.4 Software Requirements .....	22
	3.1.5 Problem Analysis .....	23
	3.1.6 Approach Analysis .....	23
3.2	Project Design and Architecture.....	25
	3.2.1 System Overview .....	25
	3.2.2 Architecture Diagram.....	26

3.3	Data Preparation .....	30
3.3.1	Dataset Description.....	30
3.3.2	Data Collection and source.....	31
3.3.3	Data Cleaning and Processing.....	32
3.3.4	Data Splitting.....	33
3.4	Implementation .....	33
3.4.1	Tools and Techniques.....	34
3.4.2	Algorithm Overview.....	36
3.4.3	Sample Code Snippets.....	38
3.4.4	Model Output and Visualization.....	39
3.5	Key Challenges .....	41
<b>4</b>	<b>TESTING .....</b>	<b>43</b>
4.1	Testing Strategy .....	43
4.2	Test Cases and Outcomes .....	45
<b>5</b>	<b>RESULTS AND EVALUATION .....</b>	<b>46</b>
5.1	Results .....	46
5.7	Comparison with Existing Solution.....	52
<b>6</b>	<b>CONCLUSIONS AND FUTURE SCOPE .....</b>	<b>53</b>
6.1	Conclusion .....	53
6.2	Future Scope .....	54
	<b>REFERENCES .....</b>	<b>56</b>

# LIST OF FIGURES

<b>3.1</b>	Data flow diagram .....	26
<b>3.2</b>	Flow chart of LSTM model .....	28
<b>3.3</b>	Flow chart of model loading .....	29
<b>3.4</b>	Code pairs in dataset.....	31
<b>3.5</b>	Pair generation logic.....	38
<b>3.6</b>	Model architecture code.....	38
<b>3.7</b>	Feature Comparison and Fusion Logic.....	38
<b>3.8</b>	Prediction and threshold logic.....	39
<b>3.9</b>	Interactive CLI.....	39
<b>4.1</b>	Manual input of code.....	44
<b>4.2</b>	Invalid file path input.....	45
<b>5.1</b>	Classification report of model .....	46
<b>5.2</b>	Confusion Matrix .....	47
<b>5.3</b>	Distribution of Redundancy Scores .....	48
<b>5.4</b>	Precision - Recall Curve .....	49
<b>5.5</b>	ROC Curve .....	50
<b>5.6</b>	CLI Based Output .....	51

# LIST OF TABLES

<b>2.1</b>	Summary of the Relevant Literature.....	15
<b>3.1</b>	Hardware Requirements.....	22
<b>3.2</b>	Software Requirements.....	22
<b>4.1</b>	Test Case and Outcomes.....	45



# LIST OF ABBREVIATIONS

Abbreviation	Meaning
AI	Artificial Intelligence
SE	Software engineering
SWD	Software development
ML	Machine Learning
DL	Deep Learning
LSTM	Long Short Term Memory
CNN	Convolutional Neural Networks
AUC-ROC	Area under the Receiver Operating Characteristic Curve
GRU	Gated Recurrent Unit
JSON	JavaScript Object Notation
CLI	Command Line Interface
IDE	Integrated Development Environment
BERT	Bidirectional Encoder Representations from Transformers
NLP	Natural Language processing

# ABSTRACT

Current code duplication is a critical issue in modern software development since increased complexity, even lower quality, and inconsistencies, may be introduced during the maintenance process. This paper presents a new approach to detecting and fixing code clones using artificial intelligence that can be applied across several repositories of software. The idea aims at figuring out both strict and similar code clones using innovative machine learning and NLP tools. The purpose of the present work is to develop an efficient Analytical Model to detect patterns of repetitious code through machine learning of massive code sets. The purpose of the system is important to refactor the code and offer useful information for extracting and analyzing the features of the code. It outlines several stages of cloning including feature extraction, data collection and training of the machine learning algorithms.

The way that can be achieved through minimizing the specific technical debt and enhancing the maintainability of code is to build an effective, all-inclusive tool for enhancing SDEs.

# CHAPTER – 1 INTRODUCTION

## 1.1 INTRODUCTION

Code duplication has remained a common challenge in SWD which complicates software systems as well as their handling and administration. When developers use copy and paste knowingly or unknowingly due to lack of time or poor communication between the development team members, they put the technical health of software on the line as this is considered one of the biggest issues which affect the quality and scalability of software.

Through the help of the advanced approach known as machine learning and natural language processing, the project “Identification of Redundant Code Using AI” will try to solve this problem by identifying code duplications. As a prerequisite for changing the existing approach to code maintenance and optimization, the project should aim at designing the intelligent system that can detect both k-coincidence and semantically related code clones.

Such approach presupposes an increased level of a developed machine learning algorithm that utilizes several sets of basic code records. The first application will be to use the AI system based on the source code repositories and other aspects of the code base besides the string comparison. This is because it encompasses context, meaning, machines and formulas, and regularities that the conventional code scanning methodologies fail to consider.

There are several important methodological factors in the approach:

- 1. Data Collection:** To make the collected code fragments as reliable as possible, select them from different sources and written in different programming languages.
- 2. Feature extraction:** As a result they are trying to develop better representations of code segments with regard to structure and function, described in the following sections.
- 3. Machine Learning Model Training:** This type of technique deployed the most enhanced deep learning technique to identify other and more complex code clone patterns.
- 4. Semantic analysis:** To improve solutions, stop relying on the key-phrases matching and Oil switch the focus to learning about the code’s context and what the programmer intended.

Due to the fact that it is a theoretical work, this research has a great potential of making a big impact.

This is how developers can do the following:

- Reduced complexity of the codebase.
- Less costs of maintenance.
- Faster running of the programs.
- Ease in code review and subsequent modifications.
- Faster rates of refactoring processes.

In addition, it helps to scale the approach up significantly more than without the use of AI. Software engineering teams can benefit from this system as it can provide fast analysis of large code-bases and reviews of the code are time-consuming and prone to mistakes. Concerning the value of the project to software engineering, the proposed system exhibits a straightforward application of artificial intelligence to enhance current practices, which is significant due to its speed and scale. When it comes to the code maintenance, the process grows into an automatic smart procedure rather than a boring and time-consuming process. In such cases these technologies are necessary to deliver best quality, easily scalable, and optima code is needed more frequently every time as software complexity increases than in modern and wider ranging technological requirements.

## **1.2 PROBLEM STATEMENT**

Code duplication is also known as “code clones” and it is a code that exists in two or more places in a program: programmers accidentally duplicate the code by inadvertently typing the same line of code again and again, or they could also copy and paste the code they have previously written. Even when only one cloning is altered while others remain untouched, this merely increases the compile-time duplicated code that needs to be controlled and updated, possibly further worsening inconsistency.

Pre-existing approaches that may for example use string comparison to compare two pieces of code or methods that requires someone to compare code manually to detect clones, are often likely to miss close clones or semi clones, which are clones that are almost the same in syntax but slightly different, semantic clones which are clones of code segments that execute the same instruction but are typed differently.

This is why code search and source code transformation to search for copied code and transform it, leads to low coding frequency, and as code bases grow so does the amount of technical debt.

This has led to the support of an automated, intelligent method that can be used to identify most accurate and semantic code clones, to make the code base cleaner, efficient and sustainable for developers and maintainers.

This is a phenomenon of repetition and replications of the same or similar codes in the same or different parts of a program resulting from the cut-copy-paste mode of programming. These redundancies also lead to a lot of repeated code, which can make maintenance a nightmare and take much longer than otherwise. changes applied to one version of the clone might not reflect in the other versions of the clones, which would give the software a bug or quirk.

Code clone detection that uses only text mining detection methods and manual code reviews are inadequate techniques that could be used to detect clones. The authors reported that the identification of near-clones, or what some researchers refer to as similar copy-clone, copy-cat clone, near-miss clone, clones that are syntactically different but semantically similar, poses some difficulty to them. One cannot guarantee the overall coverage because most of the mentioned imperfections originate from Perl language nuances typical for such types of projects implemented on a large scale.

Every developer will likely work in environments where the systems are large and intricate and where searching for every type of duplicate code by hand is almost out of the question. This makes the code more complex which leads to repetitive and complicated code, higher maintenance costs and buildup of technical debts which affect the product's capacity and its reliability quantitatively.

It means that to overcome those obstacles it is crucial to develop an automatic system that would be able to identify exact and semantic clones. It would give developers a possibility to keep their projects clean and efficient when it comes to code and minimize numerous risks connected with accidental and repetitive coding work.

## **1.3 OBJECTIVES**

The proposed solution is aimed at reinventing the code maintenance and quality assurance processes through the help of the new advanced AI system for the detection of clones. To change the process through which applications are constructed and maintained, imperative issues that are as crucial to software development as code duplication are solved.

Automated Methodology for Detection: First, it aims at delivering a highly advanced machine learning system which enables for the semantic detection of code clones in large code repositories. This task employs machine learning techniques and natural language processing which make it more of a semantic comparison to the number of forthcoming static analysis tools that only compare the code at the lexical level by strings.

### **Key Technical Objectives:**

#### **1.3.1. Comprehensive Clone Identification:**

- They are capable of distinguishing several types of code clones, identifying elements from one code to the other from several programming language codes and each has a distinctive way of addressing a given set of problems.
- This helps them to give more shades of matching that they cannot follow by simply using the words that are in their dictionary and thesaurus.

#### **1.3.2. Codebase Quality Enhancement:**

- There is the need to systematically identify these pairs of redundant code patterns. Specific recommendations should be made. Technical debt should be reduced. Algorithm refers to the specific way that the code should be analysed.
- Optimizing the total architecture within the software to enhance the robustness with respect to scalability.

### **1.3.3. Developer Productivity Tools:**

There was no doubt that designing the user interface for the system must be made an important priority. It should be able to generate tangible and comprehensible reports for areas that are duplicated in order to reduce the time it took in identifying and eliminating gaming code duplications in order to serve the greening of the coding process.

**Approach to Technical Implementation:** The system also will incorporate right machine learning models that can be developed from different codes. The data will then be pre-processed with such factors as features and semantics being predicted through the computation of complex algorithms. Therefore, the tool will offer such differences for code redundancy that have not been reached before.

#### **The AI-driven approach ensures:**

- Examining extensive software projects.
- Overview of clones with relatively high levels of confidence.
- Ability to effectively explain similarities in patterns.
- Ideas for additional code enhancements.

#### **Expected Outcomes:**

- Improvement in the extensiveness of software system maintenance.
- Incremental code maintainability.
- Derived from the concept of code productivity.
- Decreased technical debt.
- Faster life cycle of developing the software.

As a result, the collaboration of AI and SE is a project that takes SE a step further in the enhancement of intelligence, sustenance, and efficiency within the development of AI solutions.

## **1.4 SIGNIFICANCE AND MOTIVATION OF THE PROJECT WORK**

### **1.4.1 SIGNIFICANCE:**

Primarily because the concept ‘no copying and pasting’ also factor the work-load, code-size and speed aspect in software systems then optimising the source codes to remove the codes which have redundant code copied that needs the aid of computation processes would also enhance the systems capacity and the elapsed time in relation to hosted application specifications as well as user density.

#### **1.4.1.1. Facilitating Code Reuse and Modularization:**

The selection of segments that can be used in many application designs makes it easier to apply the principle of modularity on the project. However, in a few cases, a programmer’s work decreases by factors of two or three when a code is divided into small parts that depend on each other. This means longer-lasting packages with fewer possibilities of complications during maintenance; the alteration of one of these packages entails alterations in the other one.

#### **1.4.1.2. Advancing Open Source and Collaborative Development:**

This means that one's source projects are developed collaboratively and thus coherent where there is less code repetition. By using code analysis, the chances of merge conflicts in a startup project will be minimized since appealing to the artificial intelligence will provide practical means for code review.

#### **1.4.1.3. Ensuring Consistency and Standardization:**

Duplication leads to having similar codes implemented in the project through a variety of forms and approaches. By ensuring that code projects are properly coded, the AI system can, in one way or another, format all of the code so that another programmer can easily have an idea on how and where the code can be added to the system.



#### **1.4.1.4. Improving Code Security:**

In cases of other redundancies, there might be actual security loopholes when there is code duplication as the vulnerabilities may remain hidden. At this stage, the same AI system can also look for the kind of code section that might have differences in its age or complexity that has led to presence of security flaws, perhaps through code repetition. It also assists with software security.

The “**Identification of Redundant Code using AI**” project has recognized these core areas of the software development and started providing a solution which not only increases the efficiency and superiority of the software but also is setting the base for the further enhancements in software engineering technologies.

### **1.4.2 MOTIVATION:**

#### **1.4.2.1. Consequently, the work requires precautions to guarantee code scalability and adaptability.**

In the complex and rapidly developing world, the software systems have to be activated and modified with comparable ease. While flexibility might be something that can help you, this flexibility may in fact turn on you, and lead to a system with a rigid and bloated code set that only repeats the code. However, maintaining the code more flexible and compact to add more functionality in order to adapt to future challenges would foster from having an AI system to check for this kind of redundancy and rectify it on its own.

#### **1.4.2.2. Supporting Collaboration Internet Development Systems**

There are several categories of people who make contributions at divergent periods or various time zones and/or different geographical regions in the contemporary approach to developing application software. The integration of AI in this case will improve this procedure of code review as well as quality assurance because it will reduce the amount of efforts made by human beings to look for similar code. It must be challenging to persuade new developers to join the team and contribute to the process actively.

#### **1.4.2.3. Increasing the adoption of a Positive Environment and Reuse of Code**

Therefore, AI can assist the developers themselves in adhering to code-related restrictions and execute design problems in the locations where the code is excessive. This is usually done to curb the amount of work they have to do and most commonly, code duplication does this using libraries, functions, or templates. In the long run, it may lead to the creation of even more long-lasting and versatile construction materials applicable for different construction projects.

#### **1.4.2.4. Countermeasures in Preventing Security Flaws**

Other risks associated with code duplication consist of handling errors often, reviewing it often and adding features which may possess different defects. Such weak signals must be distinguished using a practical artificial intelligence system that can recognize redundancy to enhance the order of security in software development.

#### **1.4.2.5. Some general relevant research interests include:**

Also, like people, their software becomes old but remains under constant process of updating, uninstalling and reinstating. Maybe knowing who within the employees should be let go of will help in this continuous lessening. This is an effective means of preserving the system's flexibility for adapting to first-order alterations and preventing the introduction of new problems or code debts.

Code duplication within software applications should be avoided by using artificial intelligence so that development and maintenance of the software can be brought forward.

## 1.5 ORGANIZATION OF PROJECT REPORT

### **Chapter 1 – Introduction:**

In this chapter the reader will come to know why a software project faces the present and future trouble of code replication. It will help develop an understanding of ideas such as technical debt, code clones, and AI code analysis. This will be followed by the problem statement and a brief illustration of the large software ecosystems that are created due to multiple occurrences of alike code snippets. In this chapter, the goals of the project will be described, as well as the detection of exact and similar code clones and the use of machine learning and artificial intelligence to solve the remaining problem in software engineering.

### **Chapter 2 - Literature Survey:**

Conduct a literature review that has discussed earlier studies done in the areas of code clone detection, software engineering machine learning, and current ways of dealing with code clones. To check for code duplication, this chapter will explore previous studies, approaches, and materials that have been done before. The current problem of limitations in the tools, the progressive development and diversification of the code analysis approach, and how artificial intelligence is taking back code maintenance will also be covered. Also, some details concerning the natural language processing paradigm in source code analytics as well as recent developments in this sphere will be provided.

### **Chapter 3 - System Development:**

This chapter focuses on the development of the system as has been explained in the earlier chapters. It begins by formulating the proposed model and continues through a methodological approach on the system, its construction and evaluation. Here is the layout of the chapter:

1. **Dataset Preparation:** This chapter focuses on the definition of the dataset used, its main characteristics, and any changes made before incorporating it into the project. It begins with the proposed model and looks at the steps taken when going through the process of designing, building and evaluating the system. This is how the chapter is organized.
2. **Model Design:** A detailed discussion of the structure of the model and the reasons behind the dense layers being used for predicting redundancy, embedding layers and the LSTM modules. Further discussed are why this architecture was chosen and whether there are advantages to this method than more traditional approaches.

3. **Training and Validation:** In this section, the training process is explained, including the procedure in handling the training and testing data, optimum selection of the hyperparameters, and methods to avoid overfitting. Also, the measures for making the overall assessment of the model are outlined.
4. **Implementation Details:** A detailed explanation of the frameworks and libraries that have been employed for implementation such as TensorFlow, scikit-learn, pandas, etc. The hardware and software requirements that are required to implement the system are outlined in this section.
5. **Experimental Results:** In this subsection, another evaluation of the model is shown where the visualisations accompanied are confusion matrices, metrics on accuracy and a classification report.

## **Chapter 4 - Testing:**

This chapter aims to present an explanation on the various tests performed during the testing phase, the methodology used in developing the test strategies, and the outcomes that ensured that the developed system was accurate and reliable. The following are some of the key topics that have been captured:

### **1. Testing Methodology:**

In this section the details of the system testing such as unit testing, integration testing and system testing are outlined. In defining each type of test, the project context is utilised to establish the objectives and the level of testing at each phase of the project.

### **2. Test Data and Environment:**

It also defines the kind of hardware and software that has been installed or deployed in implementing a test environment. Furthermore, details about the test dataset which is distinct from the training data, or else details about data preprocessing for making a situation realistic are provided.

### **3. Validation Metrics:**

A discussion on the metrics used to evaluate the model's performance during testing, such as accuracy, precision, recall, F1-score, and AUC-ROC. Each metric is explained in relation to its relevance for code redundancy detection.

#### **4. Testing Results:**

The usefulness of the used testing phase is based on presenting the effectiveness of the system in discriminating between the redundant and non-redundant code. It is worth mentioning the set of works that use confusion matrices, ROC curves, and performance graphs as supplements to the results.

#### **5. Error Analysis:**

This section also defines the mistakes and errors that were identified during testing in terms of classification. In this paper, effects of these errors to the overall system are discussed, in addition to the possible causes.

### **Chapter 5 - Results and Evaluation:**

This chapter presents the results of the deployed code redundancy detection system in addition to an assessment of its success in achieving the goals of the project. Thus, concentration on such key parameters as accuracy, precision, the level of recall, F1-score, and AUC-ROC allows providing more comprehensive conclusions about the model performance. There are two types of plots used to present the system's outputs: confusion matrix and performance curves. The accuracy of the given model was impressive to show that the model was far effective in identifying the redundant and non-redundant code. The measures of precision and of recall complemented to the assessment of the capability for avoiding false positive and false negative cases. The F1-score as a balanced measure of both precision and recall also gave the proof to the efficacy of the model. The AUC-ROC demonstrated how effectively the model coped with the bend of the TP and FP rates.

### **Chapter 6 - Conclusions and Future Scope:**

This chapter provides the summary of the finding of the project as well as the recommendation for further research. The goal of the project was to apply syntactic and semantic methods to develop an AI-based system for identifying bad code duplications. The arrived model introduced the capacity of being generalized into various codes belonging to distinct programming languages and placed high accuracy through tokenization and an LSTM-based neural network for analyzing code pairs. This problem is important in software product development and management as it verifies the possibility of detecting code duplicates by applying machine learning approaches.

# CHAPTER – 2 LITERATURE SURVEY

## 2.1 Overview of relevant Literature:

Due to its importance to software maintenance, bug detection and optimisation the issue of finding duplicate or redundant code segments has received a lot of attention in recent years. A very important part of ensuring software quality is identifying the level of code redundancy that is present to help coordinate code reuse and making improvements in overall maintainability.

**Research on Code Clone Detection:** Significant amount of work has been done focused on the identification of clone codes which are similar or same code segments. Some of the approaches included tree based methods, token comparison and text comparison methods. There is the Clone Detection Tool that is used to identify the clones through its token-based techniques. Although they can be rather successful these techniques often lack the ability to identify code differences in structure or semantics.

**Advancements in Machine Learning for Code Analysis:** Intelligent approaches, such as ML and DL, have been increasingly used in code analysis during the last 5 years. CNN and LSTM models for example have been used to identify similarities in the code context. These methods illustrate the evolution from the syntactic approaches to the methods based on semantics.

**Transformer Models and Pretrained Architectures:** In this work, two state-of-the-art models, CodeBERT and GraphCodeBERT, paved the way in using transfer learning for code analysis. These models have been trained on natural language and code corpora, and hence, are capable of identifying syntactic and semantic relationships. For instance, CodeBERT has proved useful in some applications like code summarization, code clones detection, and code defect prediction. This has helped in enhancing the effectiveness and applicability of fault tolerance systems in redundancy detection.

**Datasets for Code Analysis:** For instance, There are data sets such as CodeNet and BigCloneBench that can be used in testing redundancy detection systems. These datasets are useful so that researchers can apply their approaches on annotated examples of code clones. Scientists are required to generate their datasets because although helpful, these datasets are typically designed for a specific number of languages and coding paradigms.

## 2.2 Key Gaps in the Literature

Despite the progress made in code redundancy detection, several challenges and limitations still exist:

1. **Limited Language Coverage:** Given that most models and tools currently used are designed for a particular suite of languages, this means that other programming languages are underserved. Such a constraint has an impact on the generalizability of these techniques in other development settings.
2. **Handling Semantic Variations:** Traditional and ML-based approaches are prone to fail in the detection of code with similar semantic character while being syntactically different. Understanding the subtle intent of code snippets is still a difficult task.
3. **Dataset Limitations:** Although datasets such as BigCloneBench create a starting point, their scope is not usually wide enough. These datasets do not have modern coding paradigms, domain-specific languages, and actual-world codebases with their default noise.
4. **Computational Costs:** Transformers-based architectures are computationally intensive for deep learning models and training & inference. This presents a hurdle to entities or individuals who do not have access to high-performance computing.
5. **Practical Deployment Issues:** Moving from research prototypes to practical, real-world tools can reveal the challenges of how a tool integrates into day-to-day workflows, whether a preceding function can be accommodated, and user-friendliness of a system.

The code duplication detection has been significantly advanced, but there still is a lot to be done in terms of significant issues. First of all, the existing tools and models usually are applicable for particular programming languages such as Java or C++, therefore, they often neglect the development of new and domain-specific languages (e.g. Rust and Solidity). This limits their use in modern polyglot development environments that are characterized by multilingual projects. Second, current techniques perform fairly well in finding the syntactic repetitions, but fail with semantic variety, that is, codes that are functionally equivalent, but structurally different. For example, different syntactic structures can enable the avoidance of detection when discussing a recursive and an iterative representation of the same algorithm assuming that each of these algorithm models is functionally the same. This gap is a testament to the need of more sophisticated approaches that should catch beneath the surface code semantics.

The second major limitation is the datasets that are used for training and evaluating. Although widely used at that time, prominent benchmarks such as BigCloneBench presuppose that the original language is used and offer false examples or simply the original ones. The lack of real world codebase in the model contributes to models that do well in controlled settings but generally fail in reality due to built in noise, debugging artifacts and unconventional patterns. The computational requirements of state of the art techniques, especially transformer based models, make them less desirable. Smaller teams or individual developers who do not have high-performance computing facilities, will not be able to use these tools due to the high resource requirements for inference as well as training.

Finally, the transition of research prototypes to practical tools is still problematic. There were numerous redundancy detection approaches developed in academic environments which are not suitably applicable to the development process. High false-positive rates, insufficient helpful refactoring advice and poor compatibility with currently existing development tools (IDE and CI/CD pipeline) are among the flaws which prevent their practical application. Besides, these gaps can only be eliminated if given the best priority in terms of accessibility, scalability as well as cross-language generalizability. Using developer-centered tools, hybrid models, and appropriately chosen datasets, this project attempts to advance these areas. Another major limitation is the datasets utilized during evaluation and training. Although valuable, famous benchmarks such as BigCloneBench apply only to a certain language, and often consist of dumbed-down or false examples. Ger Minnesota, artificial neural networks and decision trees struggle in the real world despite their good performance in the controlled settings because they are characterised by undesirable noise, debugging artifacts, and aberrant patterns. The use of intelligent methodologies like those involving transformers is compounded by computing needs which in turn limits adoption. Owing to the high resource demand for the inference and the training, this kind of tools is non-practical for weaker teams or for individual developers without a high performance computing equipment.



## 2.3 A summary of the relevant papers:

**Table 2.1: Summary of the Relevant Literature**

S. No.	Author & Paper Title [Citation]	Journal/Conference (Year)	Tools/Techniques/Dataset	Key Findings/Results	Limitations/Gaps Identified
1.	Zhenyu Xu, Victor S. Sheng. "Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models"[20]	2024	Perplexity of Large Language Models, targeted perturbation, CodeBERT for mask- filling, unified scoring scheme.	The approach outperformed current detectors, raising the average AUC from 0.56 (GPTZero) to 0.87.	The method has not been tested across a diverse set of code generation models, which could introduce biases.
2.	Swaraj, A. and Kumar, S., 2023. Programming Language Identification in Stack Overflow Post Snippets with Regex Based Tf-Idf Vectorization over ANN.[4]	2023	The corpus consisted of a total 232,727 posts varying across 21 different programming languages	Random Forest outperformed other ML classifiers in grid search, NN even performed better than RF.	Prediction of programming languages in posts is relatively much challenging task.
3.	Morteza Zakeri - Nasrabadi, Saeed Parsa , Mohammad Ramezani, Chanchal Roy:"A systematic literature review on source code similarity measurement and clone detection.[13]	2023	Utilize methods like token-based, text-based, AST, and PDG for clone detection	Token-based methods efficiently detect Type-1 and Type-2 clones, but struggle with semantic clones (Type-4).	Token-based approaches are less effective for detecting Type-4 (semantic) clones.
4.	Tong Zhou, Ruiqin Tian, Rizwan A. Ashraf, Roberto Gioiosa, Vivek Sarkar. "ReACT: Redundancy- Aware Code Generation for Tensor Expressions[19]	2022	Redundancy-aware code generation, fusion algorithms, memory optimizations.	The ReACT system significantly reduces redundancies in code generation, improving performance	The approach may introduce more memory accesses under specific input conditions, particularly when loop iterations are very small.

S. No.	Author & Paper Title [Citation]	Journal/ Conference (Year)	Tools/ Techniques/ Dataset	Key Findings/ Results	Limitations/ Gaps Identified
5.	Ajad Kumar, Rashmi Yadav and Kuldeep Kumar : “ A Systematic Review of Semantic Clone Detection Techniques in Software Systems “. [12]	2021	Program Dependency Graph (PDG): Used for detecting semantic clones by analyzing functional and data flow.	Program Dependency Graphs (PDGs) are highly effective for detecting semantic clones.	PDG-based approaches are time-consuming due to graph generation complexity.
6.	Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." [1]	2020	CodeBERT is trained from Github code repositories in 6 programming languages.	CodeBERT performs better than previous pre-trained models on NL-PL probing.	The CodeBERT itself could be further improved by generation-related learning objectives.
7.	Dong Kwan Kim: “Enhancing code clone detection using control flow graphs “[11]	2019	Control Flow Graphs (CFGs) and deep learning for clone detection.	CFGs help capture both syntactic and semantic clones, making them more effective for deep learning-based clone detection.	The framework's performance can degrade with larger thresholds, especially for more semantically distant clones
8.	Kim, D.K., 2019. Enhancing code clone detection using control flow graphs. International Journal of Electrical & Computer Engineering.[3]	2019	BigCloneBench is used as the training dataset.	Deep learning algorithms can be considered to improve the weakness of the clone classifier using supervised learning.	In the case of semantic clone types such as MT3 and WT3/4 clones, the detection performance still needs to be improved

S. No.	Author & Paper Title [Citation]	Journal/ Conference (Year)	Tools/ Techniques/ Dataset	Key Findings/ Results	Limitations/ Gaps Identified
9.	Ain, Q.U., Butt, W.H., Anwar, M.W., Azam, F. and Maqbool, B., 2019. A systematic review on code clone detection [6]	2019	Detection techniques are categorized into four classes. The textual, lexical, syntactic and semantic classes are discussed.	Detection techniques are categorized into four classes. The textual, lexical, syntactic and semantic classes are discussed. Software clones occur due to reasons such as code reuse by copying pre-existing fragments.	There is no clone detection technique which is perfect in terms of precision, recall, scalability, portability and robustness.
10.	Neha Saini, Sukhdip Singh, Suman: "Code Clones: Detection and Management "[10]	2018	Tokenization : Converts code to token sequences.	Code cloning helps reduce development time and costs. Clones can introduce bugs through propagation.	Hard to identify clones with minor modifications.
11.	Ragkhitwetsagul, C., Krinke, J. and Clark, D., 2018. A comparison of code similarity analysers. Empirical Software Engineering, 23, pp.2464-2519. [5]	2018	The generated data set with pervasive modifications used in this study has been created to be challenging for code similarity analysers.	Highly specialised source code similarity detection techniques and tools can perform better than more general textual similarity measures.	Highly specialised source code similarity detection techniques and tools can Similarity detection techniques and tools are very sensitive to their parameter settings.
12.	Yi Wang, Qixin Chen, Chongqing Kang, Qing Xia, and Min Luo. Sparse and Redundant Representation [14]	2017	Comparative analysis with k-means clustering, discrete wavelet transform (DWT).	The method effectively reduces data storage and communication costs while preserving high-resolution load information.	The sparsity constraint is uniform across all load profiles, and better results might be achieved with variable sparsity.

S. No.	Author & Paper Title [Citation]	Journal/Conference (Year)	Tools/Techniques/Dataset	Key Findings/Results	Limitations/Gaps Identified
13.	Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk : “Deep Learning Code Fragments for Code Clone Detection”[9]	2015	Uses deep learning models: Recurrent Neural Networks (RtNN) and Recursive Neural Networks (RvNN).	Detected all clone types (I-IV). Achieved 93% precision. Feasible for real-world Java systems.	High computational costs. Large files slowed training.
14.	Svajlenko, J. and Roy, C.K., 2015, September. Evaluating clone detection tools with bigclonebench.[2]	2015	BigCloneBench contains both intra-project and inter-project clones of the four primary clone types.	The tools have strong recall for Type-1 and Type-2 clones, as well as Type-3 clones with high syntactical similarity.	The tools have weaker detection of clones with lower syntactic similarity.
15.	M. Zhang, "Detecting Redundant Operations with LLVM" GSoC 2015 Proposal[17]	2015	LLVM for static and dynamic instrumentation	Prototype implementation successfully detected performance bugs in known applications.	Large trace file sizes generated by the current method, requiring optimization
16.	Randy Smith and Susan Horwitz:”Detecting and Measuring Similarity in Code Clones”[8]	2009	Uses fingerprinting algorithms at the statement level.	Detects non-identical but similar code blocks Effective fingerprinting balances efficiency and accuracy for code similarity detection	O(n <sup>2</sup> ) complexity when computing similarity for large datasets.

S. No.	Author & Paper Title [Citation]	Journal/Conference (Year)	Tools/Techniques/Dataset	Key Findings/Results	Limitations/Gaps Identified
17.	Z. A. Alzamil, "Application of Redundant Computation in Program Debugging"[16]	2008	Execution traces of sample programs for analysis	The detection of redundant computations aids in narrowing down the space of potential program defects and improves debugging efficiency.	Further optimization needed for detecting more types of memory leaks by refining dynamic memory dependency analysis.
18.	Van Rysselberghe, F. and Demeyer, S., 2003, September. Evaluating clone detection techniques.[7]	2003	The detection of code clones is a two phase process which consists of a transformation and a comparison phase.	No false matches are reported by both simple line matching and parameterized matching using suffix trees.	Future experiments should incorporate large and very-large programs into the set of cases.
19.	Andrian Marcus, Jonathan I. Maletic. "Identification of High-Level Concept Clones in Source Code"[18]	2001	Latent Semantic Indexing (LSI) for semantic similarity analysis, combined with clustering algorithms for clone detection.	The proposed method identifies high-level concept clones based on semantic similarities in source code documents.	Integration with structural-based clone detection methods could improve precision
20.	K. A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection"[15]	1996	Dynamic programming algorithms	The dynamic programming method provided higher accuracy compared to direct metric comparison.	Limited ability to handle inexact matches and scale to very large systems.

# Chapter - 3 System Development

## 3.1 Requirements and Analysis

The development of an AI-based code redundancy detection system requires an in-depth understanding of the systems needed to be used, the performance standards of the systems and the performances in the intended functions. This section provides an overview of the functional and non-functional requirements for the system, required hardware and software as well as the issue analysis and the chosen solution.

### 3.1.1 Functional Requirements

The functional requirements present the necessary attributes and actions, which the system ought to possess, in order to be considered complete. The fundamental functional requirements for the proposed system are listed as below:

1. **Code Pair Input Handling:**

Users should be able to choose two code files or put two code snippets for comparison.

2. **Tokenization and Preprocessing:**

To standardise the code inputs to a predictable form suitable for the deep learning model, tokenizing, sequencing, and padding of code inputs should be executed automatically by the system.

3. **Prediction Generation:**

A redundancy score, ranging from 0 to 1 must be predicted by the system that denotes the probability of the code snippets being redundant.

4. **Classification Decision:**

The pair should be classified by the system as “**Redundant**” or “**Non-Redundant**” according to the redundancy score, and a fixed threshold (e.g., 0.6).

5. **Batch Prediction Support:**

Large-scale redundancy detection calls for the ability of the system to assess multiple pairs of code in one batch.

6. **Result Display and Export:**

Results of both numeric and textual nature must be presented in an understandable way, exporting the results for logging or documentation is an optional feature.

## 7. **Model Training and Evaluation:**

It should be possible to train the deep learning model on labelled data and measure its performance using, for example, the accuracy, precision, recall, F 1 – score, and AUC metrics.

### 3.1.2 Non-Functional Requirements

Non-functional criteria ensure reliable and efficient functioning of the system and that it works in a user-friendly manner. Non-functional requirements for this system include:

#### 1. **Performance:**

The system should have only minimal latency for single predictions and be capable of returning redundancy scores in real-time/ near real-time.

#### 2. **Accuracy:**

The results of AUC score and confusion matrix indicate that the system should continue to correctly discriminate between redundant and non-redundant code pairs with a high accuracy (>90%).

#### 3. **Scalability:**

No crashes or lags should allow the system to work with big data with several thousand code pairs.

#### 4. **Usability:**

It should be simple enough to use that little technical know-how is required to interface (CLI or GUI).

#### 5. **Portability:**

If the line of software requirements is met then the system should be easy to transfer from one computer to another.

#### 6. **Maintainability:**

In order to make updates, modifications, or integrating with the new models or features a convenient process, the codebase should be documented and modular.

### 3.1.3 Hardware Requirements

The following hardware configuration is recommended for optimal performance during training and deployment:

**Table 3.1 Hardware Requirements**

Component	Minimum requirement
Processor	Intel i5 or AMD Ryzen 5 or above
RAM	8 GB (16 GB recommended)
Storage	100 GB Free Space
GPU (Optional)	NVIDIA GPU with CUDA support
Display	1080p Monitor

### 3.1.4 Software Requirements

The following software stack was used and is required to run or modify the system:

**Table 3.2 Software Requirements**

Component	Version/Details
Operating System	Windows 10 / 11 or Linux-based system
Programming Language	Python 3.7+
IDE / Editor	VS Code / PyCharm / Jupyter Notebook
Deep Learning Library	TensorFlow 2.11+
Data Libraries	Pandas, NumPy
NLP Utilities	Keras Tokenizer
Visualization Tools	Matplotlib, Seaborn
Additional Tools	Scikit-learn (for metrics, splitting)



### 3.1.5 Problem Analysis

It is common practice to repeat existing programming code on a large scale in contemporary software. It is necessary to search for redundant code during projects or submissions at such instances as is, primarily for educational platforms, version control systems, and plagiarism detection software. When the logic but not the syntax of code is retained and changed, traditional code comparison technique (Example: string matching and syntactic analysis) fails.

The core problem addressed by this project is:

**“How can we accurately identify logically redundant code snippets that may differ syntactically but are functionally identical or near-identical?”**

Challenges include:

- Diversity in naming conventions, formatting, and structure of the code.
- Using the token based techniques to capture meaning.
- To train a successful model, data sets that are huge and balanced are necessary.
- For generalisability across different areas of problems.

Consequently, this challenge was addressed as a **binary classification** problem where the task of the deep learning model is to predict whether or not a given two pieces of code are redundant.

### 3.1.6 Approach Analysis

The solution approach applied in this project combines deep learning architectures that were developed to perform sequence comparisons with approaches borrowed from natural language processing. Here's a detailed explanation:

#### 1. Dataset Construction:

Depending on what type of an issue the team had resolved, raw code files were organized into folders. 50 000 tagged code pairs (positive, negative) were generated using Python scripts.

## 2. **Text Preprocessing:**

An input of the same length was facilitated through padding, sequencing and tokenization. The tokenizers in training for both the code 1 and code 2 input branches were different.

## 3. **Model Design:**

The last used architecture was a bidirectional LSTM model having shared weights and dense layers. It computed element-wise products and absolute differences of the two branches enhancing its ability to pick semantic similarity. The uniform input length resulted from padding, sequencing, and tokenization. Different tokenizers were trained for the code 1 and code 2 input branches respectively.

## 4. **Training Strategy:**

Metrics such as accuracy, AUC, precision, and recall were also applied to evaluate the model after training it with binary\_cross entropy loss. Most optimal model retention was implemented through early stopping and checkpointing.

## 5. **Evaluation and Visualization:**

The model achieved nearly 90% accuracy as evidenced by its AUC score of 0.967, implying the model can recognize redundancy in codes appropriately. Redundancy score distribution, confusion matrices, ROC and precision-recall curves were some of the evaluation tools.

## 6. **Deployment:**

The model was loaded and accepted user input (text or files), which returned real-time predictions with the help of another python script `code_redundancy_predictor.py`.

As for the redundancy detection in the code with respect to the different programming styles, this combination of deep learning and conventional preprocessing has provided a scalable and efficient solution.

## 3.2 Project Design and Architecture

The planning of this project's architecture and design is done such that one can ensure that there is accurate code redundancy detection, a robust training of a model, and proper data handling. The architecture with its multiple interconnected modules makes data pre-processing, model training and final evaluation much simpler. In order to obtain a high accuracy and scalability solution, all modules are necessary.

### 3.2.1 System Overview

Recognition of logical redundancy between two code snips is the primary aim of the system. To achieve this, the project is divided into distinct phases, each of them controlled by an identifiable module. The system is capable of working in a training and inference mode.

#### Main Modules:

##### 1. Dataset Preparation Module

Reading raw code files from categorised folders, this module is able to produce positive and negative code pairs, and stores them as structured JSON. Therefore scalability and reproducibility are guaranteed.

##### 2. Preprocessing Module

Is responsible for code snippet padding, tokenization, sequence transformation and text normalisation. To facilitate more effective representation of their respective distributions, separate tokenizers are maintained for each input.

##### 3. Model Module

Analyses the links between pairs of code using mainly Bidirectional LSTM deep learning architecture. Even where there is a variation in syntax, syntactic predictions are made based on learnt semantic similarity.

##### 4. Training and Evaluation Module

Trained on the preprocessed data and then evaluated with help of common classification measures. ROC curves, confusion matrices & distribution plots are used in this module for visualisation of the results.

## 5. Prediction Module

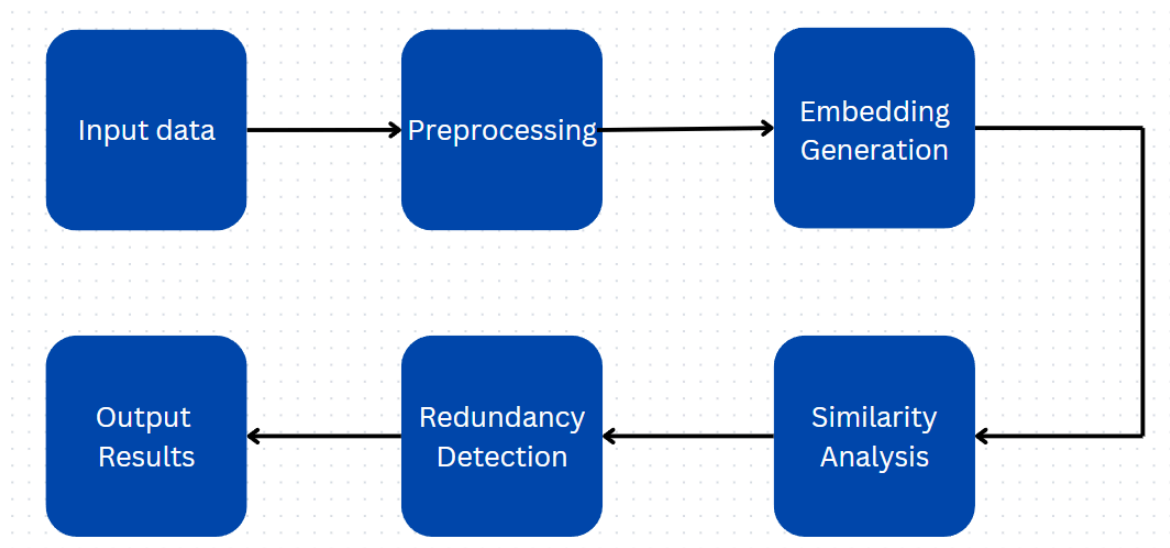
Allows interaction between users and trained models. It returns classification and redundancy score after preprocessing user input (text or file paths). Further, it enables batch inference on giant datasets.

## 6. Model Persistence Module

Will take over loading and storing trained tokenizers and models in order to make portability and reuse possible without retraining.

### 3.2.2 Architecture Diagram

The design and flow of the system are shown in the diagram below. Flow of data by data from input to preprocessing, model training, and evaluation, to the final output is in this figure. The architecture shows how things are connected and cooperating to find code redundancy.



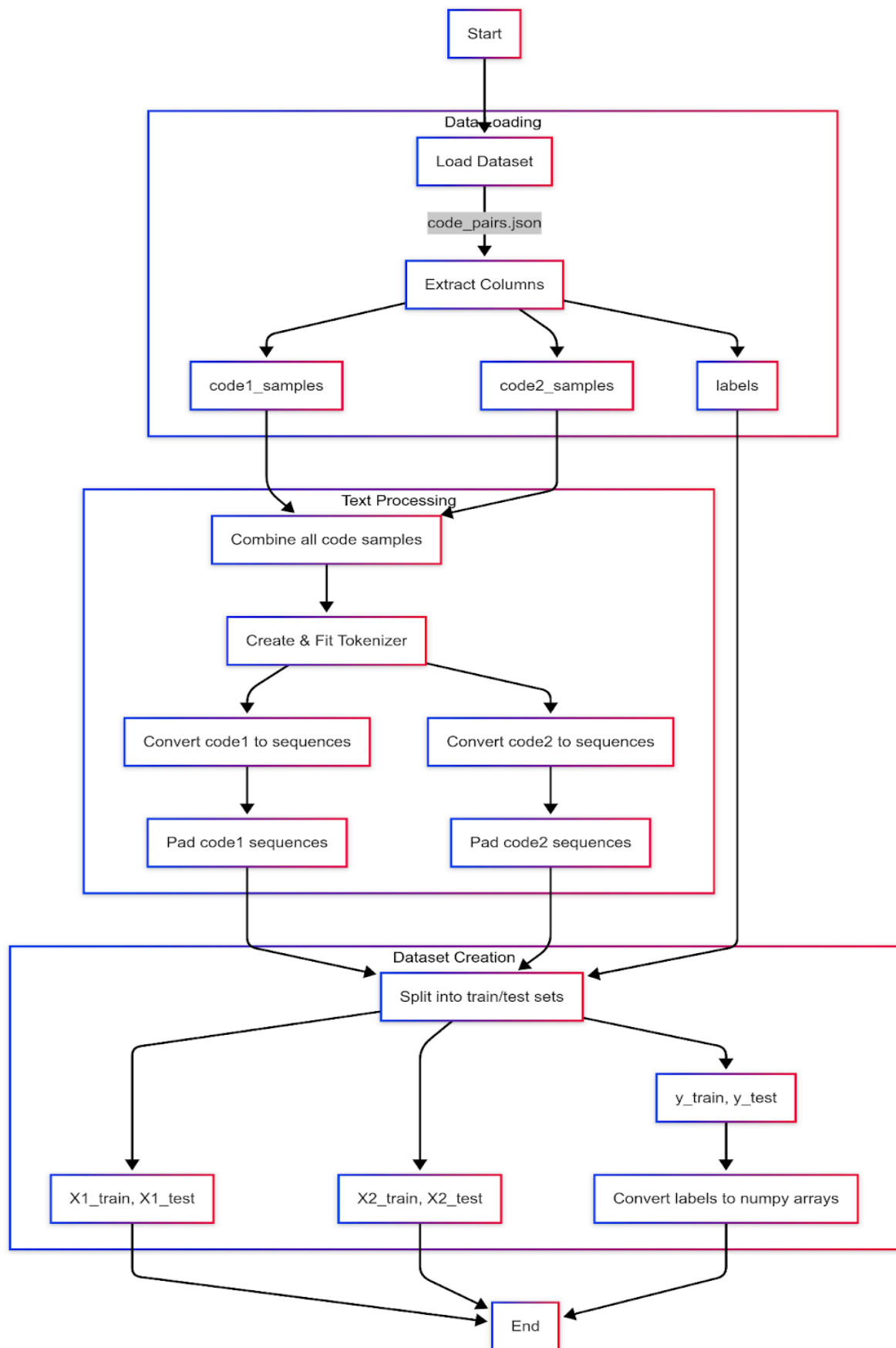
**Figure: 3.1 Data flow diagram**

The whole training and assessment pipeline of the deep learning-based redundancy detection model is controlled by the code file (`LSTM_Model.py`). First, this script loads a structured dataset of JSON format with labelled pairs of code snippets. The data is preprocessed by transforming both pieces of the code input/`code1` and `code2` into padded sequences to have uniformity and tokenize each separately. A variety of forms such as a basic LSTM, a bidirectional LSTM (employed in the last model), GRU plus an attention mechanism, as well as even Transformer-based setup, are implemented by the dynamically constructed model architecture. The inputs of both codes are computed in shared layers and relations are learned by fusion of their outputs via operations such as element-wise product and absolute difference. In order to reduce overfitting, these are then worked with dense layers and batch normalisation, and the dropout option. Also the script provides information on how to assess the model based on critical performance indexes such as accuracy, precision, recall and AUC score and different visualisations such as ROC curve, confusion matrix and redundancy score distribution. Finally, tokenizers and trained model are saved for use later.

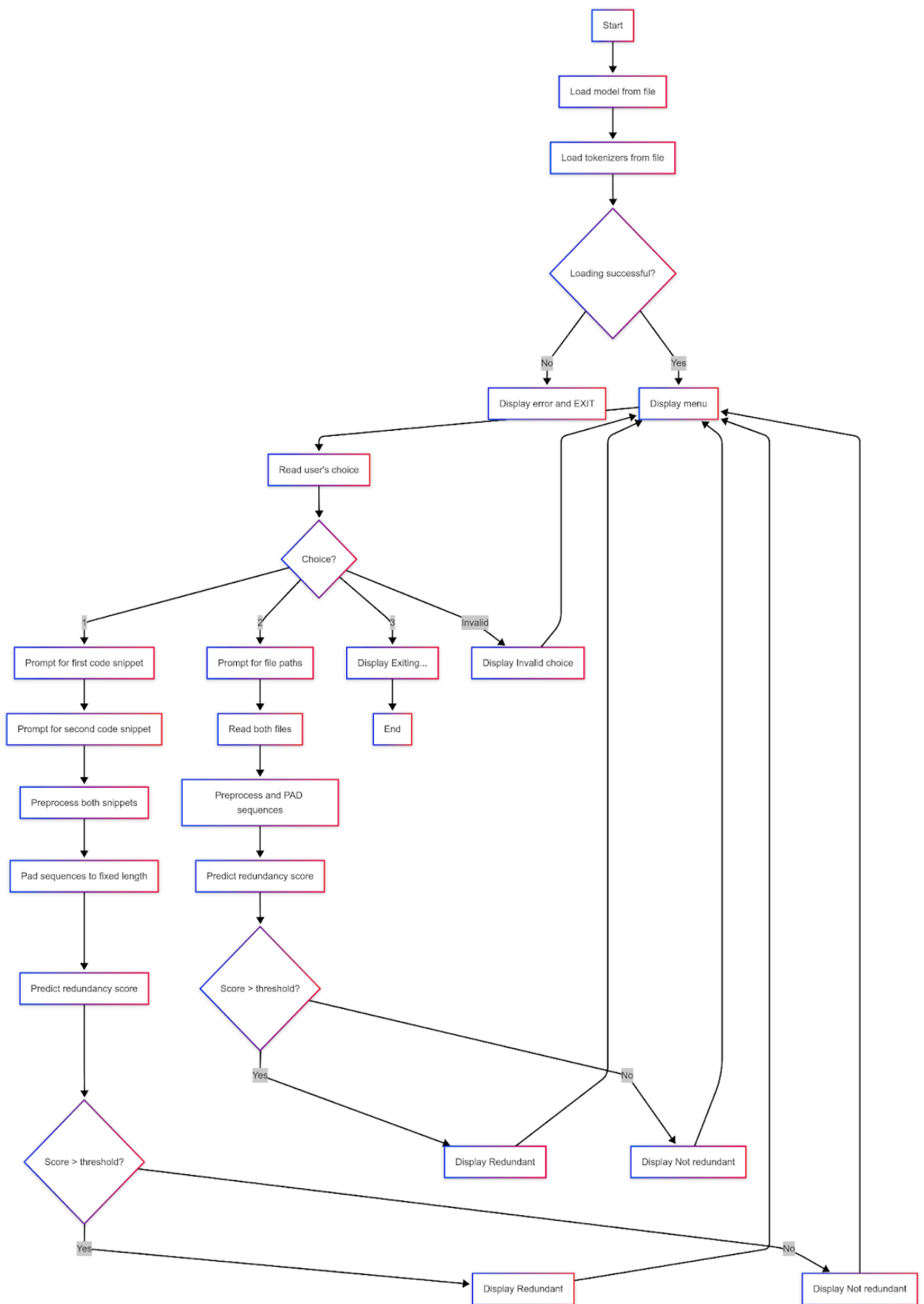
(See Fig. 3.2)

The code file (`code_redundancy_predictor.py`) conforms as an interface for deployment and inference of trained model. It begins with the unregulated loading of saved model and tokenizer files for the sake of compatibility, with custom objects such as attention mechanisms and lambda layers in the original architecture during model loading getting their proper registration. The script has a CLI that takes two code snippets to be typed in manually or that allows the users to select two code files from the system to be compared and which pre-process the inputs using the same tokenisation logic as that of training and then predicts a redundancy score between 0 and 1 and marks the code pair as “Redundant” if the score is more than a threshold (normally 0.6). It also supports batch predictions, which is evaluated for more than one pair simultaneously.. Not only is it extendable and modular, but this script is perfect for adding to larger programs or future graphical user interfaces.

(See Fig. 3.3)



**Figure: 3.2 Flow Chart of the LSTM Model**



**Figure: 3.3 Flow Chart of the script that loads the Model**

### 3.3 Data Preparation

Data preparation is critical to figuring out how effective and accurate the model is going to be in machine learning projects especially text or code based ones. For this project which applies deep learning techniques to detect code redundancy, data preparation entailed collecting various programming solutions, cleaning and structuring the plain input and ensuring it was divided correctly for training, validation and testing.

#### 3.3.1 Dataset Description

The dataset present in this project include programming code snippets stored as .txt files. The data has been distributed to **50** folders and has a single folder containing a unique programming problem or task. Each folder has all solutions to the problem (**500**) written by various authors. These solutions widely differ in names used, indentation, formatting, logic structuring, coding style making them an ideal way of training a robust model, able to understand logic level redundancy and not superficial syntax.

The final structured dataset was not a direct extraction of these files, but rather a generated collection of **code pairs** labeled as either redundant or non-redundant:

- **Redundant Pairs (Label = 1):** Formed by selecting two different code files from the **same folder** (i.e., solving the same problem).
- **Non-Redundant Pairs (Label = 0):** Formed by randomly selecting one code snippet each from **two different folders** (i.e., solving different problems).

The dataset was transformed using a Python script into a JSON file with 50 thousand code pairs overall: 25 thousand redundant and 25 thousand non-redundant. Each JSON entry contains a binary label (1 or 0); and two code strings (code1 and code2).



```

Sem 8 > {} Code_Similarity_Dataset.json
1  [
2  {"code1": "//*****
3  {"code1": "/*1090 ???*/ \n\n\n\nint num;\n\nvoid divide(int a,int k)\n{\n\tint
4  {"code1": "int sum=1;\nvoid f(int i,int j)\n{\n\tint k;\n\tif(i==1)\n\t\tsum
5  {"code1": "int n;\nvoid count(int x,int y)\n{\n\tint i;\n\tif(x==1) n=n+1;\n\t
6  {"code1": "int main()\n{\n\tint f(int x, int y);\n\tint n,m,i,z;\n\n\tscanf(\
7  {"code1": "/*\n *   Created on: 2011-11-18\n *           Author: 1100012870\n */\n\n
8  {"code1": "void qut(int a,int b);                                //????
9  {"code1": "int all=1;  //????????????????????????1\nvoid divd(int,int);  //????
10 {"code1": "//*****?????????*****\n//*****?????????*****
11 {"code1": "int f2(int i,int a)\n{\n\tint c,s=0;\n\tfor(i;i<=a/2;i++)\n\t{\n\t
12 {"code1": "int d(int,int);\nint main()\n{\n\tint i,n,x;\n\tcin>>n;\n\tfor(i=1;
13 {"code1": "int col(int a,int b);\nint main()\n{\n\tint n,i,m,l;\nscanf(\"%d\",&n);
14 {"code1": "int sum(int a,int i)\n{\n\tint ret=1;\n\tfor(i<=sqrt((float)a);i+
15 {"code1": "void f(int,int);\nint sum=0,res=1;\nint main()\n{\n\tint n,a,i,j;\n

```

**Figure: 3.4 Image showing code pairs in the dataset**

### 3.3.2 Data Collection and Sources

Due to the fact the code samples in this project were collected from various open source repositories and coding systems, there was tremendous divergence in coding styles, logic structures and implementation approach. Public GitHub repositories that hold competitive programming problem banks and online coding platforms including Codeforces, LeetCode and CodeChef – from which a large proportion of the user submitted solutions were retrieved – were among the sources.

Further by selecting data from different contributors and platforms, the dataset is naturally rich in a diversity of programming conventions needed to train a model that recognizes redundancy at the semantic and logical level, not just syntactic match. A vast spectrum of sources made it possible to develop a dataset able to reflect the differences in how various programmers solve the same problem.

The accumulated files were then organized manually or via scripts to folders based structure where 500 routines for solving particular programming activity were kept on each folder. Using this clear separation it was possible to generate positive (redundant) and negative (non-redundant) pairs in an orderly manner and in an automated form.

### **3.3.3 Data Cleaning and Preprocessing**

Once the raw dataset was compiled, it then had the rigors of preprocessing and data cleaning done on it for the sake of it being usable by deep learning models. Such operations as follows took place at this stage:

#### **1. Encoding Handling:**

Because the text files were produced from multiple systems and leveraging, the applied encoding formats were not homogenous. It is done by using a sensible exception handling of a Python script to open files with the UTF-8 encoding by default and to flip to Windows-1252 or ISO-8859-1 on demand that was how the correction was made. This ensured that in no way was encoding errors done.

#### **2. Whitespace and Character Cleanup:**

The code samples were all stripped of the extra white spaces, escape characters and control characters. Taking away noise especially helped the tokenizer focus on meaningful syntax and structure.

#### **3. Tokenization:**

Using keras tokenizer, for each of the pieces of code, certain tokens were generated. Importantly, however, since code1 and code2 inputs are from separate source files and can have differing token distributions then separate tokenizers must be trained for each. This promotion of a better understanding of the model and an intact personality character was created.

#### **4. Sequencing and Padding:**

After tokenization, the outputs were padded to a fixed output limit of 150 tokens and converted into a series of integers. This uniformity is therefore necessary to feed data that has to be in deep learning models with a consistent input dimension.

The raw and unstructured code data was transformed in a neat numerical format through these preprocessing processes and reflected each code snippet semantics at the same time and fitted the input requirements of deep learning.

### 3.3.4 Data Splitting

The cleaned dataset was divided into training, validating and testing subsets to make sure that proper training and objective evaluation went fully.

- **Training Set:** This subset constituted 80% of the data (40,000 pairs) which was used to train the model's parameters.
- **Validation Set:** This data as 20% of the training set was utilized during training to track model performance, make possible early stopping, and avoid over-fitting.
- **Test Set:** Ten thousand pairs, 20% of the whole dataset, were reserved for the last model evaluation. In training and validation, this ensured that model was challenged with unseen data.

When splitting up the dataset, a stratified sampling method was applied in order to balance between redundant and non-redundant classes. This ensured that class imbalance did not skew model metric's performance by keeping the 50:50 ratio of both classes involved in the subsets.

To completely eliminate any structural bias from the original data's folder-based arrangement, shuffling was used to distribute pairs randomly to different orders.

## 3.4 Implementation

In the execution of the project, design concepts are actualized into an executable system that is working. To handle the data, develop the model, train, evaluate, and deploy the model this required writing scalable, and modular Python code. Complementing with powerful data processing methods, implementation of the idea essentially followed machine learning libraries and deep learning frameworks.

### 3.4.1. Tools and Techniques

This project relied upon a diversity of state-of-the-art machine-learning methods, well-structured libraries, and modern programming instruments to ensure successful and productive development. Each tool was selected from the extensive list, based on its performance in similar applications, support from the community, and its compatibility with the aims of the project.

#### 3.4.1.1 Programming Language

- **Python:** Selected to become the primary implementation language due to its simplicity, ease of reading, and powerful machine learning and data science ecosystem, Python made an excellent choice. It was ideal for developing prototypes of deep learning models, training and evaluation due to its large library (such as TensorFlow and scikit-learn) and dynamic developer community.

#### 3.4.1.2 Development Environment

- **Visual Studio Code and Google Collab:** Due to strong debugging facilities, Python development enhancements in addition to smooth integration with a version control engine such as Git, Visual Studio Code (VS Code) was the main IDE. Google Collab allowed for exploration of the power of applying different preprocessing/modelling techniques using exploratory data analysis, iterative model testing and visualisation.

#### 3.4.1.3 Libraries and Frameworks

- **TensorFlow / Keras:** Initiated the setting up of deep learning model construction, training, and optimisation. Trained scalability and GPU accelerations in training were ensured by TensorFlow along with rapid neural network prototyping by Keras's high level API.
- **Scikit-learn:** Which are used as auxiliary in baselines model comparisons, performance metrics ( e.g. precision, recall, F1-score) and data splitting (train/test/validation).

- **Pandas and NumPy:** Working on data requires such vital activities as preprocessing, cleaning and modification . Utilized because it does quick mathematical operations in large sets of numbers, NumPy is useful. On the other hand, Pandas DataFrames simplify the organization of data sets to work with. These are tools that are vital for anyone handling data.
- **Matplotlib and Seaborn:** Such visuals help one explain how good an outstanding model is performing. From the visualizations above one can visualize the errors that the model makes, its discriminative capabilities, and how its predictions are distributed.
- **OS and JSON:** The OS module managed (loading datasets and storing model weights (file system functions)) while the JSON made it easy to store and retrieve the metadata in an organised format ((evaluation results and hyperparameters)).

#### 3.4.1.4 Techniques Employed

- **Tokenization and Sequence Padding:** Raw code snippets were converted into numerical sequences through tokenization (a word level or subword tokenization). For batch processing in neural networks, padding was necessary as by truncating or extending the size to a specific length, there was a guarantee of having uniform dimensions.
- **Siamese Neural Networks:** Pairs of code snippets were compared with a dual-input architecture using a similarity metric to compare them. This method works best for redundancy detection because it prioritises relational features over lone classifications..
- **Bidirectional LSTM:** Processed sequences in both forward and backwards to capture context information in code. This was important in understanding semantic and syntactic patterns especially in larger code blocks.
- **Attention Mechanism (for GRU variant):** Attention mechanisms, which are extensions of the base-model, helped the network focus on effective tokens (function calls and keywords) and deemphasize ineffective tokens (comments and whitespace). With a complex structure of the code, this increased accuracy and interpretability.
- **Model Checkpointing and Early Stopping:** Continuous training decreases data loss due to interruption by the checkpointing because it saves top performing model weight of the training. By halting training when validation performance plateaued, the latter problem was avoided while costs were also contained.

### 3.4.2. Algorithm Overview

The basis of the system is the Siamese Bidirectional LSTM architecture, which is supposed to make a comparison between two input code snippets and determine if they are logically redundant. Below, a detailed description of the training and prediction stages algorithm's operation can be found.

#### **Input Pair:**

The algorithm has two code snippets to work with; code1 and code2. These snippets can be used for batch processing and real-time analysis and can be derived from a preprocessed dataset or direct users submit them.

#### **Tokenization:**

Each code piece is tokenized through use of Keras' Tokenizer which converts unformatted text to a series of integer values. To ensure their exclusive token distributions and fair treatment, different tokenizers remain for codes 1 and 2.

#### **Sequence Padding:**

With the truncation or padding, the tokenized sequences are normalised to a length of 150 tokens. The uniformity of input dimension is maintained at this step which is essential for batch processing and maintaining steady performance of the model.

#### **Embedding Layer:**

For each integer token a shared embedding layer that maps tokenized sequences both, maps each integer token to a dense 100-dimensional vector. This transformation makes the model capable of efficiently decoding code structure and semantics due to the syntactic and semantic relations between tokens captured by it.

#### **Bidirectional LSTM Layer:**

A bidirectional LSTM layer, shared, uses the embedded sequences to analyse the code from both directions. Due to this dual-context analysis, the model is capable of understanding dependencies and patterns which a unidirectional approach may fail to discover.

#### **Feature Comparison:**

A total of three operations are used for comparing the two outputs from the two LSTM branches (which are codes 1 and 2):

- **Absolute Difference:** calls focus on differences between code representations.
- **Element-wise Product:** is able to find similarities between the two code features.
- **Concatenation:** Creates a comprehensive illustration of alignment and divergence by merging the original outputs and the differences and product features.

#### **Dense Layers:**

With the ReLU activation the concatenated vector passes two fully connected layers. To enhance learning stability and relieve overfitting and ensure strong generalisation to unknown data batch normalisation and dropout are employed.

#### **Output Layer:**

The redundancy score of a final dense layer is from 0 to 1 with sigmoid activation. This score measures the probability for which the input fragments are logically redundant.

#### **Thresholding:**

If the score is over 0.6, a specified threshold set to balance recall and precision the model tags the pair as redundant. Below this threshold, code scores are indicative of a non-redundant code.

The pipeline is implemented using TensorFlow/Keras with each step tailored to optimise accuracy and high efficiency. The architecture is quite effective at detecting redundancy in real-world code since it detects not only syntactic similarities, but deeper semantic equivalencies as well.

### 3.4.3. Sample Code Snippet

```
# Generate at most 500 positive pairs
pos_pairs = list(itertools.combinations(codes, 2))
random.shuffle(pos_pairs) # Shuffle to get diverse pairs
pos_pairs = pos_pairs[:PAIRS_PER_FOLDER] # Limit to 500

# Generate exactly 500 negative pairs per folder
other_folders = [f for f in all_folders if f != folder]
neg_pairs = []
while len(neg_pairs) < PAIRS_PER_FOLDER:
    f1, f2 = random.sample(other_folders, 2)
    if folder_codes[f1] and folder_codes[f2]: # Ensure both have codes
        code1 = random.choice(folder_codes[f1])
        code2 = random.choice(folder_codes[f2])
        neg_pairs.append((code1, code2))
```

Figure: 3.5 Pair Generation Logic

```
# Input layers for two code snippets
input1 = tf.keras.layers.Input(shape=(self.max_length,))
input2 = tf.keras.layers.Input(shape=(self.max_length,))

# Embedding layer
embedding = tf.keras.layers.Embedding(self.vocab_size, self.embedding_dim)

# Embedded inputs
embedded1 = embedding(input1)
embedded2 = embedding(input2)

# Choose model architecture
if model_type == "lstm":
    # Basic LSTM architecture
    shared_lstm = tf.keras.layers.LSTM(128)
    lstm1 = shared_lstm(embedded1)
    lstm2 = shared_lstm(embedded2)
```

Figure: 3.6 Model Architecture with Bidirectional LSTM

```
diff = tf.keras.layers.Lambda(abs_diff)([lstm1, lstm2])
prod = tf.keras.layers.Lambda(mul_prod)([lstm1, lstm2])

# Merge layers - concatenate all features
merged = tf.keras.layers.Concatenate()([lstm1, lstm2, diff, prod])
```

Figure: 3.7 Feature Comparison and Fusion Logic



```

# Predict
prediction = self.model.predict([code1_padded, code2_padded], verbose=0)
prediction_value = prediction[0][0]
is_redundant = prediction_value > 0.6

return prediction_value, is_redundant

```

**Figure: 3.8 Prediction and Thresholding Logic**

```

while True:
    print("\nCode Redundancy Detector")
    print("1. Check two code snippets")
    print("2. Check two code files")
    print("3. Exit")
    choice = input("Enter your choice (1-3): ")

    if choice == '1':
        print("\nEnter first code snippet (type 'END' on a new line when finished):")
        code1_lines = []
        while True:
            line = input()
            if line == 'END':
                break
            code1_lines.append(line)
        code1 = '\n'.join(code1_lines)

        print("\nEnter second code snippet (type 'END' on a new line when finished):")
        code2_lines = []
        while True:
            line = input()
            if line == 'END':
                break
            code2_lines.append(line)
        code2 = '\n'.join(code2_lines)

```

**Figure: 3.9 Interactive CLI for Inference**

### 3.4.4 Model Output and Visualization

Ten thousand unseen code pairs in a test dataset were used to test an activated model. Multiple outputs, visual and numerical, were generated from the evaluation in an attempt to assess performance and interpret model behaviours.

#### 3.4.4.1 Performance Metrics:

- **Accuracy:** 90%.
- **Precision & Recall:** 0.90 for both classes.
- **F1-Score:** 0.90 (balanced).
- **AUC Score:** 0.967.

Such metrics serve as an evidence of the powerful discrimination capacity of the model to distinguish redundant and non-redundant code pairs as well as good generalisation to unpaved data.

#### **3.4.4.2 Visualizations Included:**

##### **1. Confusion Matrix:**

Through a clear presentation of true positives, true negatives, false positives and false negatives in a 2x2 format the confusion matrix ensures a lot of detail of classification performance of the model. Based on the relatively low off-diagonal values that imply few misclassifications and high values of the diagonal that imply accurate predictions, this visualisation confirmed that the model had equal performance along redundant and non-redundant classes. The matrix was beneficial in picking out particular cases in which the model performed poorly, this could be targeted for improvement for the classification process.

##### **2. ROC Curve:**

Sensitivity (true positive rate) versus specificity ( $1 - \text{false positive rate}$ ) trade-off at all possible values of the classification threshold is nicely illustrated by the Receiver Operating Characteristic curve. Our model's Area Under Curve (AUC) value was almost 1, verifying an excellent discriminative capacity between redundant and non-redundant code pairs. The steep initial ascent of the curve illustrates that the model can retain high sensitivity at very low false positive rates and thus is well suited for practical use where minimizing false alarms is critical.

##### **3. Precision-Recall Curve:**

This curve provides valuable information regarding the performance of the model for the positive (redundant) class by relating the change in precision-recall ratio to varying decision thresholds. Even if a large proportion of real redundant cases is detected, the model preserves decent precision because the curve remains highly constant among all calls. The optimal cutoff threshold (0.6) that achieved the best trade off for detecting true redundancies and minimizing false positives was, to a great extent, based on this trade off.

##### **4. Redundancy Score Distribution:**

The distributions for actual redundant and non-redundant code pairs are well distinguished in the histogram of predicted redundancy scores. Redundant pairs are largely scored high in

the middle (clustering around 1.0), whereas non-redundant pairs are scored low (clustering around 0.0) having little overlap in the middle. This clean breakup explains the good performance measures observed in other visualisations and proves that the model is able to discriminate between the two classes effectively.

All of these visual outputs were developed using Matplotlib and Seaborn, much attention was given during labelling, colouring and scaling in order to ensure readability and clarity. All taken together, they presented a detailed picture of the effectiveness of the model, in which each of them presented special points of view that significantly helped in performance evaluation, interpretation of results and identification of particular areas where the classification system could be improved.

### **3.5 Key Challenges**

Creating an AI-based code redundancy detection system at various stages presented a series of both practical and technical challenges. These challenges tested the resistance of the chosen approach and affected several design decisions that were made in realization. The majority of difficulties encountered during the process are listed below:

#### **1. Handling Raw and Diverse Code Structures**

Fighting raw.txt documents, where code was written by several people in different styles, was the first and the biggest problem. A lot of noise was introduced by the inconsistencies in formatting, indentation, naming conventions for variables, and the form of comments used. In addition, it was more problematic to read and clean files since some of the files contained non-default characters or other encodings. Many fallback mechanisms and encoding methods were required in the preprocessing pipeline to guarantee that all files could be parsed without losing important syntax.

#### **2. Constructing a Balanced and Meaningful Dataset**

Creating a balanced dataset with regards to class distribution (redundant vs. non-redundant) and enough samples represented another major problem. Positive pairs were easy to generate using code from the same folder while creating meaningful negative pairs without accidental overlap required requisite sampling across folders. There was a possibility of

“false negatives” – code pairs that were similar logically, but come from different issues; this can occur in label noise.

### **3. Capturing Semantic Similarity with Token-Based Input**

Since the syntax of programming code is more rigid and semantics-wise complex compared to natural language, traditional NLP has problems with it. Loop, conditional, and recursion code semantics were not consumed automatically by token-based models that relied on Keras’ Tokenizer. Consequently, the model’s ability to generalise across unknown patterns or logic structures was limited as it was compelled to discover semantic patterns from sequence representation only.

### **4. Designing a Suitable Model Architecture**

One important decision was to choose the correct architecture. A basic LSTM would be able to handle sequence data, but it was incapable of capturing contextual symmetry between code positions and long-term dependencies that are needed for code comparison. Computational complexity, model compatibility, and training stability issues were caused by trying bidirectional LSTMs, GRUs with attention, and even transformer layers. Eventually, a Bidirectional LSTM was selected, as a trade off between interpretability and performance.

### **5. Avoiding Overfitting During Training**

Since code patterns recur within folders, there was a danger of overfitting the high capacity model with a small vocabulary size. This was reduced through utilization of strategies such as use of dropout, batch normalisation and early stopping. However, tweaking these hyperparameters required numerous tries and attention to validation loss.

# Chapter 4: Testing

No machine learning system is perfect and any machine learning system needs to be tested rigorously so as not to discover any flaws during or after its implementation. The multiple testing levels that were applied to this project are functional testing of the end to end prediction pipeline; Unit testing of various components; and model validation. The aim was to ensure the interface is working the same way for a variety of input types and the model is able to identify redundant code pairs.

## 4.1 Testing Strategy

The approach to testing was developed to evaluate the model's ability to predict as well as the correct operation of the deployed system. It belongs to the following general categories:

### 1. Dataset Split Validation

To maintain class balance, stratified splitting of the dataset into training (80%) and testing (20%) subsets was performed. This ensured that the performance of the model in unknown data was not only fairly evaluated. 20% more of the training set was used for validation set during training so as to monitor it for overfitting by early stopping.

### 2. Unit Testing of Functions

The python functions were tested individually including:

- File reading Feature handling and encoding.
- Tokenization and sequence padding.
- Model construction for different architectures.
- Redundancy prediction logic and thresholding.

With the help of unit tests, logical errors and the failure to adhere to the expected input format and incompatibilities of the components (heatmap and the model for example) were identified.

### 3. Model Evaluation Testing

Estimation of the trained Bidirectional LSTM model performance on the test set was done using standard classification metrics:

- Accuracy
- Precision
- Recall
- F1-score
- AUC (Area Under the Curve)

To comprehend the model behaviour and verify that it can successfully differentiate redundant and non-redundant code pairs, visual plots such as confusion matrix, performance ROC curve, and precision-recall curve were used.

#### 4. CLI-Based Functional Testing

The prediction interface was tested using:

- Manual input of code snippets.
- File-based input (with both valid and invalid paths).

This ensured that the interface managed to process invalid, malformed input elegantly, reacted suitably, and turned out equal results with the model's backend.

```
Code Redundancy Detector
1. Check two code snippets
2. Check two code files
3. Exit
Enter your choice (1-3): 1

Enter first code snippet (type 'END' on a new line when finished):
void main()
{
    int n,a[1000],i,j,k,jud=0;
    scanf("%d%d",&n,&k);
    for(i=0;i<n;i++)scanf("%d",&a[i]);
    for(i=0;i<n;i++){
        for(j=i;j<n;j++)if(a[i]+a[j]==k){jud=1;break;}
        if(jud)break;
    }
    if(jud)printf("yes");
    else printf("no");
}
END
```

**Figure: 4.1 Manual input of code snippets**

```

Model loaded and compiled successfully
Loading tokenizers from C:/Users/piyus/Desktop/Major Project/Identification of Redundant Code Using AI/Sem 8/LSTM_Model/lstm_code
Model and tokenizers loaded successfully!

Code Redundancy Detector
1. Check two code snippets
2. Check two code files
3. Exit
Enter your choice (1-3): 2

Enter path to first code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\3\397.txt
Enter path to second code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\3\495.txt
Error reading files: [Errno 2] No such file or directory: 'C:\\Users\\piyus\\Desktop\\Major Project\\ProgramDataset\\3\\397.txt'

```

**Figure: 4.2 Invalid file path Input**

## 4.2 Test Cases and Outcomes

A series of representative test cases were conducted to verify the system’s performance in realistic usage scenarios. Below are selected examples of these test cases and their observed outcomes:

**Table 4.1 Test Cases and outcomes**

Test Case	Description	Input Type	Expected Outcome	Observed Result
TC-01	Identical logic, different syntax	Two snippets using different variable names and formatting.	Classified as <b>Redundant.</b>	Correctly classified
TC-02	Completely unrelated code	Code snippets solving different problems (e.g., sorting vs. recursion).	Classified as <b>Non-Redundant.</b>	Correctly classified
TC-03	Similar structure, different logic	Code with similar loop structure but different goals.	Classified as <b>Non-Redundant.</b>	Correctly classified
TC-04	Minimal input length	Short code with 1–2 lines each.	Handle appropriately.	Correctly classified
TC-05	Invalid file path	Non-existent file provided in CLI.	Appropriate error message.	Error handled without crashing

The testing confirmed that the system performs reliably in various edge cases and input formats, and that the model’s predictions align well with human intuition regarding code redundancy.

## Chapter 5: Results and Evaluation

After training the model on a balanced set of 50,000 code pairs, the system was tested using a separate test set of 10, 000 unseen pairs. The aim was to examine the performance of the model in identifying redundant code against non-redundant code with particular emphasis on effectiveness and reliability. The evaluation was a combination of numerical measures of performance and graphs to verify the system's predictions and make sense of its operation.

### 5.1 Classification Report:

Classification Report:				
	precision	recall	f1-score	support
0	0.90	0.90	0.90	5000
1	0.90	0.90	0.90	5000
accuracy			0.90	10000
macro avg	0.90	0.90	0.90	10000
weighted avg	0.90	0.90	0.90	10000

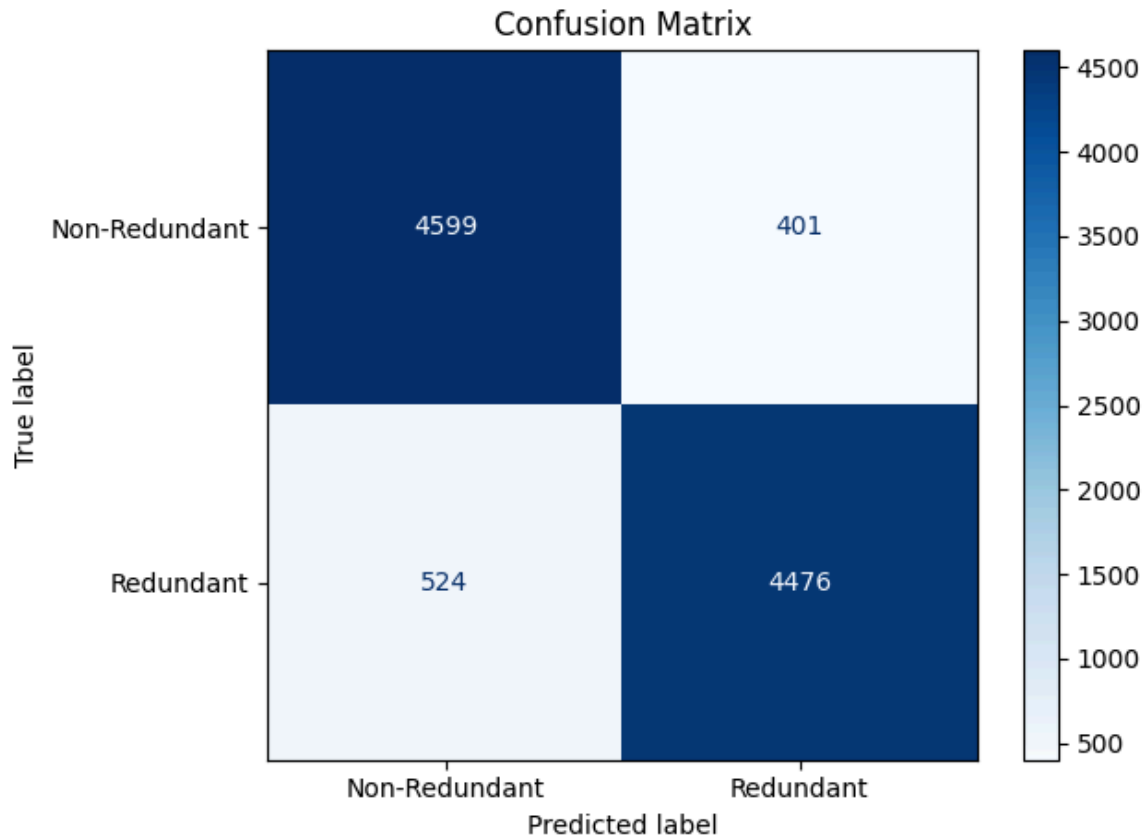
**Figure: 5.1 Classification Report of Model**

Standard classification metrics, including precision, recall, and F1-score are used to summarise the predictive performance of the model for the two classes in the classification report. redundant (class 1) and non-redundant (class 0). The overall accuracy of the model was 90% given 0.90 precision, 0.90 recall and 0.90 F1-score for each class. By the uniformity of these values across the two classes it is confirmed that the model is equally good at mining redundant code and correctly rejecting non-redundant pairs. In addition, there is no class dominance and equal performance, as reflected by equal weighted average and macro average to the per-class metrics.

This strong classification report provides evidence for the models ability to process an assorted range of syntactically different code samples and distinguish between their semantic similarity or difference. Also, it is indicative that the model has performed reasonable generalisations, and is not just a recollection of training data-patterns.



## 5.2 Confusion Matrix:

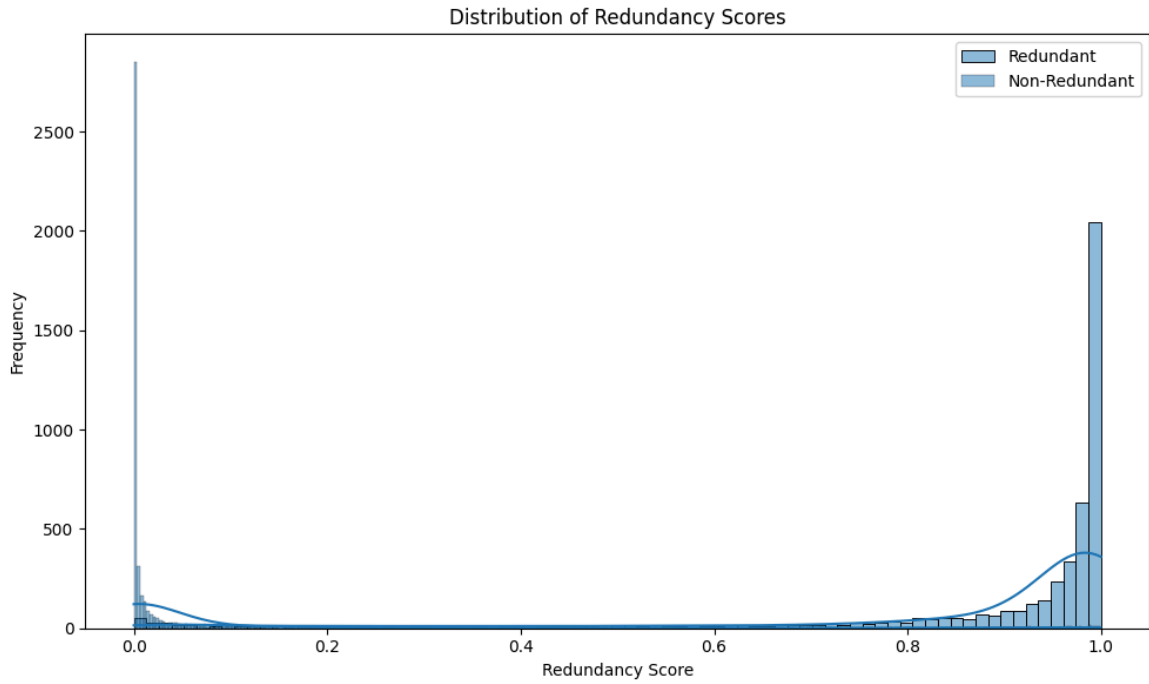


**Figure: 5.2 Confusion Matrix**

Graphical and numerical representation of the model's true against predicted labels is demonstrated in the confusion matrix. From 5,000 real non-redundant pairs that it reports shows that only 401 were wrongly tagged as redundant the other 4,599 were correctly tagged. Similarly 4,476 out of 5,000 real redundant pairs were classified and the other 524 pairs were mis-classified as non-redundant.

Based on this distribution, the model attains a balance between representation in terms of specificity (true negative rate) and representation in terms of sensitivity (true positive rate). The model's capacity to minimize the false positives and false negatives comes to the fore in the relatively low number of misclassifications. This is vital for activities such as detection of redundancy, where errors can mislead one on code similarity.

### 5.3 Redundancy Score Distribution:

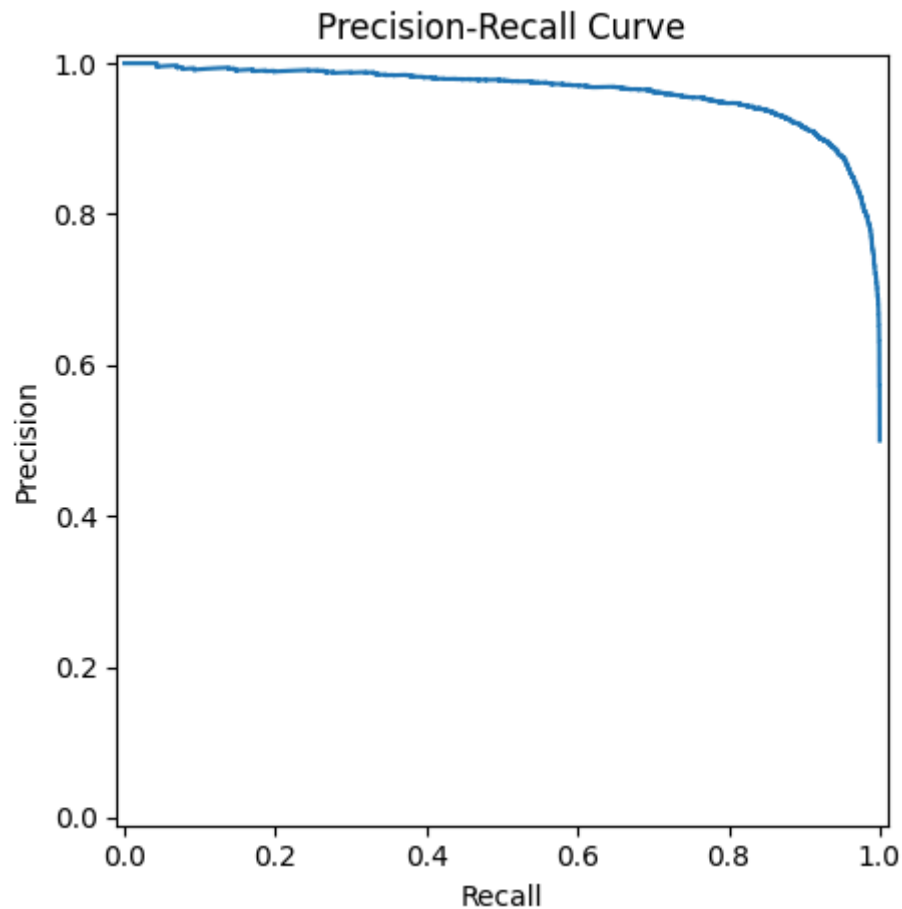


**Figure: 5.3 Distribution of Redundancy Scores**

In the redundancy score distribution plot, the distribution of the spread of the expected probabilities for both classes is shown. Non redundant code pairs center close to 0.0 whereas redundant code pairs pack around 1.0. This extreme separation between the two groups reflects well how the model learns semantic representations.

Also from the plot, it clearly reveals that the model produces predictions with high confidence most of the time, which is suitable for binary classification tasks. The model outputs are more interpretable as a result of better separated score distributions, which also reduces uncertainty in thresholding.

## 5.4 Precision-Recall Curve:

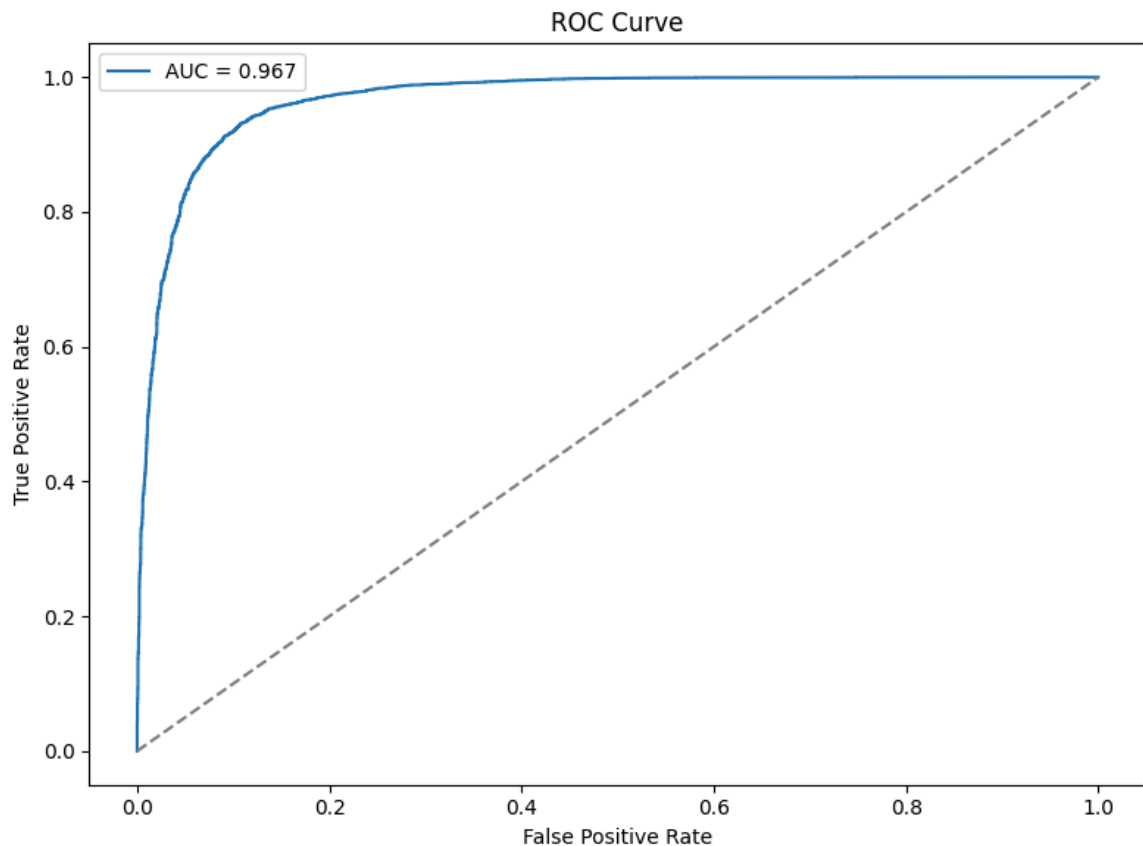


**Figure: 5.4 Precision-Recall Curve**

Precision is shown by precision-recall curve, and given the fact that it is not mandatory that the threshold value be kept constant and that the operating point is adjustable, then it is possible to maintain high precision while increasing default Recall as required. For this model, the thrust of the curve remains high over the plot, indicating that the precision remains high even as the recall increases. For detecting redundancy this balance is crucial, since it helps to detect redundant logic with the minimum number of pseudo-sound alarms.

As is customary and can be predicted, the curve deviates only a little at the highest recall values. The model is reliable over a wide range of thresholds due to its strong precision-recall balance and rather confirms than denies that it does not compromise either metric to boost the other.

## 5.5 ROC Curve:



**Figure: 5.5 ROC Curve**

The Receiver Operating Characteristic (ROC) curve is plotted graphically by showing the true positive rate against the false positive rate taken across varying classification thresholds. With an amazing AUC (Area Under Curve) score of 0.967, the curve presides closely over the plot's upper-left border.

Irrespective of the threshold, a model with an AUC score that is close to 1 has a very high differentiation power between the two classes. This high level of discriminative power is of special importance for cases of deployment where thresholds can be adapted depending on how strict redundancy detection should be.

## 5.6 Final CLI Based Output of redundancy detection Model:

```
Model loaded and compiled successfully
Loading tokenizers from C:/Users/piyus/Desktop/Major Project/Identification of Redundant Code Using AI
Model and tokenizers loaded successfully!

Code Redundancy Detector
1. Check two code snippets
2. Check two code files
3. Exit
Enter your choice (1-3): 2

Enter path to first code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\14\14.txt
Enter path to second code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\14\139.txt

Redundancy Score: 1.0000
Conclusion: The code files are redundant.

Code Redundancy Detector
1. Check two code snippets
2. Check two code files
3. Exit
Enter your choice (1-3): 2

Enter path to first code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\29\26.txt
Enter path to second code file: C:\Users\piyus\Desktop\Major Project\ProgramDataset\46\139.txt

Redundancy Score: 0.0620
Conclusion: The code files are not redundant.

Code Redundancy Detector
1. Check two code snippets
2. Check two code files
3. Exit
Enter your choice (1-3): 3
Exiting...
PS C:\Users\piyus\Desktop\Major Project\Identification of Redundant Code Using AI> █
```

**Figure: 5.6 Final CLI Based Output of redundancy detection Model**

## 5.7 Comparison with Existing Solutions

Detecting code redundancy has been traditionally resolved by static analysis, syntax-based matching or plagiarism-checking tools. Simple ways include basic string matching, token based comparison and abstract syntax tree traversal. Although these can identify exact, near-exact matches, they tend to struggle when logic stays the same while syntax changes — a very common occurrence in programming in the real world, particularly within a learning/collaborative coding scope.

MOSS (Measure of Software Similarity), is a widely used solution that identifies similar code by means of token-based hashing. MOSS is bound by its ability to respond to surface-level changes in code, renaming of a variable for instance, or code reformatted by a programmer even when it detects academic plagiarism quite well. It is shallow in semantic spirit and therefore cannot capture functional resemblance when the implementation architecture is different.

Other tools including JPlag and CodeMatch also belong to the syntactic similarity and current structure based approaches. These systems are based mainly upon rules, and, though they offer quick outcomes, they incorrectly classify logically equivalent, but syntactically dissimilar code with unrelated. In addition, these tools are typically language-specific limited by the ability to work on new programming languages or patterns.

Nevertheless, a number of advantages apply to this deep learning-based strategy employed in this project:

- Even when code is reformatted, rewritten, reorganized, it can identify logic-level redundancy because such representations have been learned of code.
- Most fundamentally language-independent, it can be extended to cover other programming languages if the right training data are available.
- Instead of simply outputting binary judgement, the model is capable of producing interpretability and flexibility in terms of thresholds by generating a redundancy score.
- Because of its trainable and scalable nature, more data and more complicated architectures (CodeBERT and graph-based models, for example) can grow its accuracy and generalisation.

If the standard solution is appropriate for static or rule based checking, but it is not so for large scale, intelligent code analysis.

# Chapter 6: Conclusions and Future Scope

## 6.1 Conclusion

The goal for this project was the development of an AI-based system which could recognise redundancy between two code snippets (even if they are syntactically different but perform identical logical operations). Since developers create code differently, traditional string-matching or rule-based approaches fail in many cases. In order to circumvent this, a solution leveraging a data-driven, deep learning solution was created and realized through modern NLP methods using NLP applied to source code.

Step one in the project was to create a complete and diverse dataset. Bearing in mind large repositories and coding platforms of open-source, raw programming solutions were collected, categorized by the type of the problem, and placed in folders. Subsequently 50,000 code pairs, 25000 redundant and 25000 non-redundant, were generated using a custom script and stored in an organised JSON format. This information was the basis for training the model.

The preprocessing step was well-studied so as to handle the unstructured nature of the raw code. To normalise the input with logical semantics, tokenization, sequencing and padding were implemented. A robust and versatile model architecture with multiple options, LSTM, Transformer, GRU with attention and Bidirectional LSTM was implemented. A final model selected was a **Bidirectional LSTM** based Siamese network, which performed well on many evaluation measures.

To prevent overfitting the model was trained with checkpointing as well as early stopping features. Notably with precision, recall, and F1-score of 0.90 each for both classes; the model was continually able to report 90% accuracy from the 10000 pair test set. The ROC curve demonstrated spectacular discriminative capability with a score of 0.967 as the AUC. Initiatives such as redundancy score distribution, precision-recall curve, confusion matrix confirmed the effectiveness of the model.

A separate module was developed so as to fill the gap between the technical development and user application of the developed projects in real time deployment. After users input code snippets directly or from files, this deployment script yields a classification and a redundancy score. Since the system can perform both single and batch prediction, it can be used for a range of applications from refactoring to plagiarism detection and for version control analysis.

Having said all of this, this work successfully demonstrated the possibility of detecting semantic redundancy in code through deep learning. It points out how AI in software engineering can serve as necessary support-based tooling for developers, automate lengthily and error-prone work, and increase the quality of the code.

## **6.2 Future Scope**

Although promising results are obtained in a current implementation, much can be done to improve it in the process of its further evolution and expansion. There are 4 options which include domain expansion, system integration, data and model architecture.

### **1. Advanced Embeddings and Language Models:**

Keras Embedding layers are utilised alongside straightforward tokenisation under the current method. Since CodeBERT, GraphCodeBERT, or CodeT5, pre-trained models, using large-scale code corpora, are better able to capture the programming logic, data flow, and context, they may be utilized later in implementation. These embeddings could however be much more accurate and generalisable.

### **2. Support for Multiple Programming Languages:**

Nowadays, the system largely works with C/C++ code. To make the applicability of the model to different sets of developer communities higher, additional support for other languages (Python, Java, JavaScript) needs to be added. Multilingual training may also lead to the development of a cross-language redundancy detector capable of comparing logic across syntaxes.

### **3. Integration with Development Tools:**

The prediction module can be plugged in as an extension or as a plugin to version control systems (GitHub) or IDEs (VS Code, IntelliJ). As a result, potential sig code could be identified in real time during development and therefore more intelligent recommendations for code cleanup, reuse, and refactoring.



#### **4. Visual Interface for User Interaction:**

The current operation method of the system is through command-line interface. Adoption of a desktop GUI or web-based dashboard would improve usability of the tool to non-technical users, such as educators and project reviewers that may want to detect duplicate submission or copied logic from student's projects.

#### **5. Contextual and Structural Analysis:**

To identify more profound structural similarities that are invisible at the token level, the next versions may employ intermediate code representation, control-flow graphs (CFG) or abstract syntax tree (AST) parsing. For obfuscated code, or functionally complex code, such approaches would increase accuracy of the system.

#### **6. Self-Learning Capabilities:**

Online learning may be used to design the system to be able to continuously improve on sufficient user feedback again. From time to time, the model could be updated using new-labelled examples allowing it to adapt to programming styles and coding trends.

#### **7. Benchmarking with External Datasets:**

Although the model has been rigorously validated with the use of custom data, other standard public datasets (POJ-104 and CodeNet) can be utilized to continue verifying the performance of the model. This would also help in benchmarking this system against previous research and matching its generalisation ability.

# REFERENCES

- [1] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. and Zhou, M., 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.
- [2] Svajlenko, J. and Roy, C.K., 2015, September. Evaluating clone detection tools with bigclonebench. In 2015 IEEE international conference on software maintenance and evolution (ICSME) (pp. 131-140). IEEE.
- [3] Kim, D.K., 2019. Enhancing code clone detection using control flow graphs. *International Journal of Electrical & Computer Engineering* (2088-8708), 9(5).
- [4] Swaraj, A. and Kumar, S., 2023. Programming Language Identification in Stack Overflow Post Snippets with Regex Based Tf-Idf Vectorization over ANN. In ENASE (pp. 648-655).
- [5] Ragkhitwetsagul, C., Krinke, J. and Clark, D., 2018. A comparison of code similarity analysers. *Empirical Software Engineering*, 23, pp.2464-2519.
- [6] Ain, Q.U., Butt, W.H., Anwar, M.W., Azam, F. and Maqbool, B., 2019. A systematic review on code clone detection. *IEEE access*, 7, pp.86121-86144.
- [7] Van Rysselberghe, F. and Demeyer, S., 2003, September. Evaluating clone detection techniques. In *Proc. Int'l Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)* (pp. 25-36).
- [8] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," *\*ResearchGate\**, 2009.
- [9] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk : "Deep Learning Code Fragments for Code Clone Detection",2015.
- [10] Neha Saini, Sukhdip Singh, Suman : Code Clones: Detection and Management . In *International Conference on Computational Intelligence and Data Science (ICCIDS 2018)* .
- [11] Dong Kwan Kim :Enhancing code clone detection using control flow graphs. In *International Journal of Electrical and Computer Engineering (IJECE)* Vol. 9, No. 5, October 2019, pp. 3804~3812 ISSN: 2088-8708, DOI: 10.11591/ijece.v9i5.pp3804-3812 .

- [12] Ajad Kumar, Rashmi Yadav and Kuldeep Kumar: A Systematic Review of Semantic Clone Detection Techniques in Software Systems. IOP Conf. Series: Materials Science and Engineering 1022 (2021) 012074.
- [13] Morteza Zakeri-Nasrabadi, Saeed Parsa , Mohammad Ramezani, Chanchal Roy, Masoud Ekhtiarzadeh :A systematic literature review on source code similarity measurement and clone detection: Techniques ,applications ,and challenges.The Journal of Systems & Software 204 (2023) 111796.
- [14] IEEE Transactions on Power Systems, vol. 32, no. 3, pp. 2142-2151, May 2017.
- [15] K. A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," *Automated Software Engineering*, vol. 3, pp. 77–108, 1996
- [16] Z. A. Alzamil, "Application of Redundant Computation in Program Debugging," *Journal of Systems and Software*, vol. 81, no. 11, pp. 2024–2033, Nov. 2008.
- [17] M. Zhang, "Detecting Redundant Operations with LLVM," *GSoC 2015 Proposal*, unpublished, 2015.
- [18] Marcus, A., & Maletic, J. I. (2001). Identification of High-Level Concept Clones in Source Code. *Proceedings of the International Conference on Automated Software Engineering (ASE '01)*, pp. 46-53.
- [19] Zhou, T., Tian, R., Ashraf, R. A., Gioiosa, R., Kestor, G., & Sarkar, V. (2022). ReACT: Redundancy-Aware Code Generation for Tensor Expressions. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22)*.
- [20] Xu, Z., & Sheng, V. S. (2024). Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(21), 23155-23162.

**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT**  
**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING AND INFORMATION TECHNOLOGY**

**PLAGIARISM VERIFICATION REPORT**

Date: 09 May, 2025.

Type of Document: B.Tech. (CSE / IT) Major Project Report

Name: Piyush Joshi, Samriti Thakur, Prabhakar Vashistha Enrollment No.: 211397, 211150, 211327

Contact No: 90153 68 062 E-mail: 211397@juitsoan.in

Name of the Supervisor (s): Dr. Ramesh Narwal, Ms. Seema Rani

Title of the Project Report (in capital letters): Identification of Redundant Code Using AI

**UNDERTAKING**

I undertake that I am aware of the plagiarism related norms/regulations, if I found guilty of any plagiarism and copyright violations in the above major project report even after award of degree, the University reserves the rights to withdraw/revoke my major project report. Kindly allow me to avail plagiarism verification report for the document mentioned above.

- Total No. of Pages: 67
- Total No. of Preliminary Pages: 9
- Total No. of Pages including Bibliography/References: 58

Piyush Joshi  
Samriti Thakur  
Prabhakar Vashistha  
Signature of Student

**FOR DEPARTMENT USE**

We have checked the major project report as per norms and found Similarity Index 91%. Therefore, we are forwarding the complete major project report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

Signature of Supervisor

Signature of HOD

**FOR LRC USE**

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received On	Excluded	Similarity Index (%)	Abstract & Chapters Details	
Report Generated On	<ul style="list-style-type: none"><li>All Preliminary Pages</li><li>Bibliography/ Images/Quotes</li><li>14 Words String</li></ul>		Word Count	13,000
			Character Count	73,294
		Submission ID	Page Count	55
			File Size (in MB)	

Checked by

Name & Signature

Librarian

## ORIGINALITY REPORT

9%

SIMILARITY INDEX

7%

INTERNET SOURCES

6%

PUBLICATIONS

3%

STUDENT PAPERS

## PRIMARY SOURCES

1	Arvind Dagur, Karan Singh, Pawan Singh Mehra, Dharendra Kumar Shukla. "Intelligent Computing and Communication Techniques - Volume 1", CRC Press, 2025 Publication	<1 %
2	<a href="http://www.ir.juit.ac.in:8080">www.ir.juit.ac.in:8080</a> Internet Source	<1 %
3	<a href="http://link.springer.com">link.springer.com</a> Internet Source	<1 %
4	<a href="http://plg.uwaterloo.ca">plg.uwaterloo.ca</a> Internet Source	<1 %
5	<a href="http://d-nb.info">d-nb.info</a> Internet Source	<1 %
6	<a href="http://ijece.iaescore.com">ijece.iaescore.com</a> Internet Source	<1 %
7	Mehdi Ghayoumi. "Generative Adversarial Networks in Practice", CRC Press, 2023 Publication	<1 %
8	<a href="http://www.cs.uccs.edu">www.cs.uccs.edu</a> Internet Source	<1 %
9	<a href="http://arxiv.org">arxiv.org</a> Internet Source	<1 %
10	<a href="http://docplayer.net">docplayer.net</a> Internet Source	<1 %

11	harvest.usask.ca Internet Source	<1 %
12	Sharifnik, Mohammadamin. "Automating Fracture Detection and RQD Calculation in Core Images Using Deep Learning.", University of Toronto (Canada), 2024 Publication	<1 %
13	s3-eu-west-1.amazonaws.com Internet Source	<1 %
14	repositorio.uam.es Internet Source	<1 %
15	"Emerging Technologies in Data Mining and Information Security", Springer Science and Business Media LLC, 2021 Publication	<1 %
16	K. Matsumoto. "Software quality analysis by code clones in industrial legacy software", Proceedings Eighth IEEE Symposium on Software Metrics METRIC-02, 2002 Publication	<1 %
17	Li, Yi. "Learning Representations for Effective and Explainable Software Bug Detection and Fixing", New Jersey Institute of Technology, 2023 Publication	<1 %
18	par.nsf.gov Internet Source	<1 %
19	Submitted to srmap Student Paper	<1 %
20	dblp.uni-trier.de Internet Source	<1 %

# R 007

## R007

 Quick Submit Quick Submit Jaypee University of Information Technology

---

### Document Details

**Submission ID****trn:oid:::1:3244667040****Submission Date****May 9, 2025, 3:14 PM GMT+5:30****Download Date****May 9, 2025, 3:24 PM GMT+5:30****File Name****Final\_Report\_-\_Google\_Docs.pdf****File Size****1.3 MB****55 Pages****13,000 Words****73,294 Characters**

# 0% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

**Caution: Review required.**

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

## Detection Groups



0 AI-generated only 0%

Likely AI-generated text from a large-language model.



0 AI-generated text that was AI-paraphrased 0%

Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

### Disclaimer

Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

## Frequently Asked Questions

### How should I interpret Turnitin's AI writing percentage and false positives?

The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (\*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

### What does 'qualifying text' mean?

Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.

