‘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

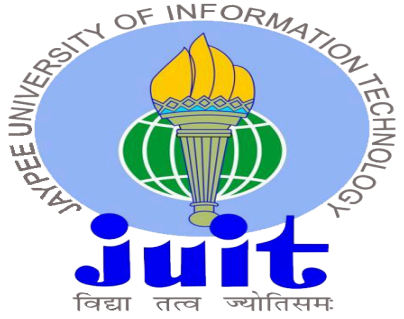
Piyush Joshi(211397), Samriti Thakur(211150),

Prakhar Varshney(211327).

Under the guidance & supervision of

Dr. Ramesh Narwal — **Assistant Professor CSE&IT**

Ms. Seema Rani — **Assistant Professor CSE&IT**



Department of Computer Science & Engineering and Information Technology

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

December 2024

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 December 2024.

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:
Place:

Ms. Seema Rani
Assistant Professor
CSE&IT
Date:
Place:

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 December 2024 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.

Name: Piyush Joshi	Name: Samriti Thakur	Name: Prakhar Varshney
Roll No.: 211397	Roll No.: 211150	Roll No.: 211327
Date:	Date:	Date:

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

Dr. Ramesh Narwal	Ms. Seema Rani
Assistant Professor	Assistant Professor
CSE&IT	CSE&IT
Date:	Date:
Place:	Place:

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	Name: Samriti Thakur	Name: Prakhar Varshney
Roll No.: 211397	Roll No.: 211150	Roll No.: 211327

TABLE OF CONTENTS

LIST OF FIGURES VI

LIST OF TABLES VI

ABSTRACT VII

1 INTRODUCTION ……..1

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

2 LITERATURE SURVEY ……12

2.1 Overview of relevant literature ……12

2.2 Key gaps in the literature ……...…...13

2.3 A summary of the relevant papers ……14

3 SYSTEM DEVELOPMENT ……...19

3.1 Requirements and Analysis ……..19

3.1.1 Functional Requirements ……..……19

3.1.2 Non-Functional Requirements ……..19

3.1.3 Hardware Requirements ……..…...20

3.1.4 Software Requirements ……..….....20

3.1.5 Problem Analysis ……...……..….20

3.1.6 Approach Analysis ……..……..20

3.2 Project Design and Architecture…… ……...21

3.2.1 System Overview ……...21

3.2.2 Architecture Diagram……..…...23

3.3 Data Preparation ……..……..……....24

3.4 Implementation ……26

3.5 Key Challenges ……29

4 TESTING …….....31

4.1 Testing Strategy ……...…….31

4.2 Test Cases and Outcomes ……..……33

5 RESULTS AND EVALUATION ……..…….....35

5.1 Results ……...……...35

6 CONCLUSIONS AND FUTURE SCOPE ……..…….....39

6.1 Conclusion ……39

6.2 Future Scope ……40

REFERENCES ……..……42

LIST OF FIGURES

3.1 Data flow diagram ……24

3.2 Redundant Code in dataset ……..24

3.3 Non - Redundant Code in dataset……...25

5.1 Confusion Matrix ……..35

5.2 Precision - Recall Curve …...……..36

5.3 ROC Curve ...……...……..37

5.4 Distribution of Redundancy Scores ……37

LIST OF TABLES

2.1 Summary of the Relevant Literature……14

ABSTRACT

Code redundancy represents a significant challenge in modern software development, leading to increased maintenance complexity, reduced code quality, and potential introduction of inconsistencies. This research project proposes an innovative AI-driven approach to automatically detect and address code clones across diverse software repositories. By leveraging advanced machine learning and natural language processing techniques, the proposed system aims to systematically identify both exact and semantic code duplications. The research focuses on developing a sophisticated AI model trained on comprehensive code datasets to recognize redundant code patterns efficiently. Through intelligent feature extraction and semantic analysis, the system will provide developers with actionable insights for code refactoring. The methodology encompasses multiple stages of clone detection, including data collection, feature representation, and machine learning model training.

The primary objective is to create an automated, user-friendly tool that enhances software development practices by minimizing technical debt and improving overall code maintainability.

CHAPTER – 1 INTRODUCTION

1.1 INTRODUCTION

Code duplication is one of the ever-enduring software development issues that become a thorn in the side of more sophisticated software systems in terms of their performance and manageability. Whenever developers use copy and paste – intentionally or not, due to time pressure or lack of communication between teams – they bring in quite a lot of technical debt that affects the overall quality and ability to scale the software.

The need to mitigate this challenge is what the “Identification of Redundant Code Using AI” project seeks to solve through the utilisation of modern machine learning and natural language processing to help in identifying code duplications. Thus, the goal of the given project is to create an intelligent system which can detect both k-coincidence and semantically related code clones to change the present approach to code maintenance and optimization.

The basis of such an approach is the highly developed machine learning algorithm based on versatile base sets of code patterns. The first application will involve using the AI system to compare source code repositories and other features of the code base beyond simple string comparison. This includes comprehending meaning, machines and formulas, and regularities in context that can be easily missed by old school stand alone code scanning mechanisms.

The project's methodology involves several key technical components:

1. Data Collection: Accumulating code snippets from different sources across different programming languages in order to have a reliable training set.

2. Feature Extraction: Working towards deriving better representations of code segments around structure and function as summarized in the subsequent sections.

3. Machine Learning Model Training: Applying the state of the art deep learning method for the precise identification of different and more challenging code clone patterns.

4. Semantic Analysis: To avoid key phrase matching and proceeding to understand code context and intent to improve solutions.

The potential impact of this research is substantial. By automatically identifying redundant code segments, developers can:

- Reduce codebase complexity

- Minimize maintenance overhead

- Improve overall software performance

- Enhance code readability and maintainability

- Accelerate refactoring processes

In addition, the utilisation of AI significantly enhances the scalability of the approach. While the manual code reviewing takes much time and is prone to errors, this system is capable of quickly analyzing large code bases and providing insights to software engineering teams.In addition, besides the basic importance in revealing potential large scale code optimization the project fits into the research in the field of software engineering by showing how artificial intelligence techniques can be leveraged to enhance the current practices. If we take code maintenance, the very act becomes a sophisticated, intelligent automated process rather than a tedious, time-consuming chore that, as software grows increasingly complex, such technologies become indispensable for delivering the high quality, highly performant and highly maintainable code that is required so much more often across a rapidly widening range of technologically diverse environments.

1.2 PROBLEM STATEMENT

Code duplication, also known as “code clones,” occurs when two pieces of code are similar or exactly the same and dispersed throughout a program most often through copying and pasting or by accident. Such duplication increases the amounts of code to manage and maintain and creates potential for inconsistency when one clone is updated but not others.

Old approaches may require a person to manually compare code to find clones or are based on text similarity and often omit close clones, that are code segments which are clones but differ slightly, or semantic clones which are segments of code that perform the same operation but coded differently.

That is why, with the growth of codebases' size and their increasing complexity, the use of code search and source code transformation for identifying and refactoring copied-and-pasted code leads to low coding frequency and the continuously growing amount of technical debt.

This has set the endorsement of an automated, intelligent technique for identifying exact as well as semantic code clones that would allow the developers and maintainers to have cleaner, more efficient and sustainable code bases.

The existence of similar or identical codes in the same program or different programs is known as code clones because of cut-copy-paste technique or through other repetitive programming practices. These redundancies also greatly expand the quantity of code being written, which can make maintenance much harder and prolonged. also involve some level of variability; changes made in a clone instance may not propagate throughout the clones, which in turn cause software glitches or oddities.

Common techniques for code clone identification include manual code reviews and text mining detection methods, which are inadequate. The authors pointed out that they have difficulties when it comes to detecting non-identical clones, often referred to as ‘‘similar copy clones’’ or near-miss clones with trivial differences or when clones are syntactically dissimilar but semantically equivalent clones. Since most of these weaknesses stem from the Perl language, one cannot guarantee perfect coverage, particularly in large, tangled projects.

When developing any software systems, developers are likely to encounter situations where the size and complexity of the systems make it practically impossible for manual identification and removal of any form of duplicated codes. This leads to production of inefficient code, increased costs of maintenance, and building up of technical debts which in the long run affect the productivity of the software as well as the quality of the software produced.

In order to overcome these challenges there is a need to develop an automatic system which can identify both exact clones and semantic clones. Imagining a solution of such nature would enable developers to sustain maintainable codebases, optimize code quality while diminishing the risks linked to redundant and inconsistent development.

1.3 OBJECTIVES

The project aims to use a new advanced system of AI in code clone detection to transform code maintenance and quality assurance. The important issues such as code duplication, which are fundamental to software development, are addressed with an aim of revolutionising the way developers design and maintain code.

Automated Detection Methodology: The Primary objective is to propose a complex AI solution for exact as well as semantic code clones recognition within large code databases. In contrast to most existing static analysis tools that compare code solely based on string similarity at the lexical level, this task uses machine learning techniques alongside natural language processing for more semantic comparison of the code segments.

Key Technical Objectives:

1.3.1. Comprehensive Clone Identification:

- Detect different forms of code clones

- Identify important keys across various codes of programming languages

- Each of them has its unique way to a specific set of problems, which allows delivering more subtle analysis of semantic similarities other than simple matching of words.

1.3.2. Codebase Quality Enhancement:

Mapping of these redundant code patterns should be done systematically

- Provide recommendations that are as specific as possible

- Decrease the amount of technical debt, as well as analyze the code with help of specific algorithms.

- Enhance robustness of overall software structure for better and easier scalability.

1.3.3. Developer Productivity Tools:

It was impossible not to realize that there is a need for developing an interface in the system that would be easy to navigate for the user. The system should be able to generate clear and measurable reports about areas that are redundant. Greening up the coding process by reducing the time taken to locate and correct gaming code duplications

Technical Implementation Approach: The system will use efficient machine learning models based on various code repositories. The pre-processing of data will be done by employing complex algorithms for extraction of features and for construing semantics, and so, the tool will furnish a level of detail into differences in redundancy of code, hitherto unseen.

The AI-driven approach ensures:

- Analyzing large software projects.
- Clone identification with a very high degree of certainty
- Analytic understanding of resemblance of patterns
- Suggestions for further code improvement

Expected Outcomes:

- Reduction of the level of difficulty in maintaining software systems
- Enhanced maintainability of the code
- Developed from the concept of improved developer efficiency
- Minimized technical debt
- faster software development life cycle

Hence, this project of linking AI with SE is a step towards making software engineering more intelligent, maintainable and efficient in its way of delivering AI solutions.

1.4 SIGNIFICANCE AND MOTIVATION OF THE PROJECT WORK

1.4.1 SIGNIFICANCE:

The “no copying and pasting” concept also targets at easing work load, enhancing the code, size and speed in software systems. Therefore, optimization of the source codes eradicating duplicate codes which consume unessential computation processes could also enhance efficiency in execution time and system capabilities in relation to the need of hosted applications and concurrently increasing user density.

1.4.1.1. Facilitating Code Reuse and Modularization:

The decision of segments that could be reused in lots of application designs helps to apply the modular designs in the project. When you organize code into small dependent pieces, a programmer reduces the amount of effort that he has to apply two or three fold at some time or another. This results in long term packages with fewer issues that can arise when trying to sustain them, and with modifications to one of these packages meaning alterations to the others.

1.4.1.2. Advancing Open Source and Collaborative Development:

Among the pros of open source software projects is that removing duplicative code improves cohesiveness of code, and a community constructs it. Since AI-based solutions provide coders with a useful tool to manage code review, the application of code analysis will increase the interaction between startup contributors and decrease the probability of code merge conflicts.

1.4.1.3. Ensuring Consistency and Standardization:

Duplication leads to occurrence of duplicate codes within many project implementation and styles. The AI system can scan code projects and make sure that they are coded right which will in one way or another make all the code appear formatted for another programmer so that he or she know s how best to integrate it into the system.

1.4.1.4. Improving Code Security:

where there is code duplication, loopholes may be camouflaged within an application whereas in other instances of redundancy there may be real security threats. The AI system can also identify holes in security by searching the code for repetitive material which may cause

discrepancies in terms or complexity or age of code sections that also help in software security too.

These critical areas of software development are what the "**Identification of Redundant Code using AI**" project has captured to present a solution that not only augments the efficaciousness and superiority of the software but also forms a foundation for the future growth of software engineering technologies.

1.4.2 MOTIVATION:

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

Software systems need to expand and evolve readily for the current world that has been defined by advanced technology. This flexibility can be counter productive due to the tendency to duplicate the code which makes the rigidity of the code set and the inefficiency of the code set to increase. Thus, having an AI system that does the redundancy check and corrects it on the spot would be of great value in direction towards maintaining the abilities of the code flexible and compact for easier and faster expansion to face future challenges.

1.4.2.2. Supporting Collaboration Internet Development Systems

Modern approach to developing application software is that multiple groups of individuals contribute codes at distinct time zones and/or geographic locations. Because integrating AI to the code review and quality assurance process reduces the effort of the people in the identification of repeated code, integrating AI to the code review and quality assurance process will improve this process. This, in turn, results in problems of achieving a more attractive speed at which new developers join the team and contribute on the assumption of collaborative work.

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

Thus, AI can assist the developers themselves to adhere to the constraints which are linked to the amount of code, and apply the design patterns in the areas where there is an excess of codes.

Usually developers will reapply code through the application of libraries, functions or templates in a bid to avoid developing new sets of code lines thus lessening their work efforts. It can lead in the long run to building of even more firm and adaptable construction components, which may be used in a number of projects.

1.4.2.4. Countermeasures in Preventing Security Flaws

Handling errors many times, reviewing many times, and the duplications of features which can be flawed variably are added security vulnerabilities to code duplication. With a view to enhance the order of security in software development, a useful AI system to identify redundant patterns is needed to detect these weak signals.

1.4.2.5. Some general relevant research interests include: Supporting software evolution and maintenance.

They grow old and so is their software which is often modified, refreshed and repaired. In this continuous fine tuning maybe it can help by pointing out those employees that should be removed from the system. This is a way of sustaining the fact that when fundamental changes have to be made the system is not only easier to manage but also less prone to introduce new problems or open code debt.

In an efficient execution of an approach that is founded on artificial intelligence it is possible to discover and eradicate instances of code replication as indicating the following impacts to the improvement of software development and maintenance. By surmounting such challenges, such a system shall be an innovator in applying artificial intelligence within the increase and improvement of the quality of code in software.

1.5 ORGANIZATION OF PROJECT REPORT

Chapter 1 – Introduction:

In this chapter, the reader will be presented with the crucial necessity of the current and upcoming challenge of code duplication in software projects. It will create understanding of terms like code clones, technical debt, and AI code analysis. The problem statement will be stated in the introduction where inefficient large software ecosystems due to replication of similar code segments will be described. The chapter will present the goals of the project, it will highlight the automated detection of exact and semantic clones of the code and the necessity of utilizing artificial intelligence and machine learning methods for solving the so far unsolved problem in software engineering.

Chapter 2 - Literature Survey:

Sufficient literature review to explain previous work done in the area of code clone detection, machine learning for software engineering and current methods of handling code clones. This chapter will review the earlier research, methods, and resources applied with the objective of defining and insulating code duplication. The existing tools will be exposed on their shortcomings, the development and growth of code analysis methodology, and how artificial intelligence is reclaiming code maintenance. It will also briefly discuss the latest development in natural language processing, and natural language processing models for analysing source code.

Chapter 3 - System Development:

This chapter provides a detailed account of the system's development process. It begins with the conceptualization of the proposed model and includes the step-by-step methodology used to design, implement, and evaluate the system. The chapter is structured as follows:

1. **Dataset Preparation**: A thorough explanation of the dataset used, including its structure, features, preprocessing steps, and any augmentations or expansions made to enhance its usability for the project.
2. **Model Design**: A detailed description of the proposed model architecture, including the embedding layers, LSTM components, and the dense layers utilized for redundancy prediction. The rationale for selecting this architecture and its potential advantages over traditional approaches is also discussed.

3. **Training and Validation**: This section outlines the training process, including the data split into training and testing sets, hyperparameter tuning, and strategies employed to prevent overfitting. Metrics for model evaluation are also highlighted.
4. **Implementation Details**: A comprehensive explanation of the tools, libraries, and frameworks used for implementation, such as TensorFlow, scikit-learn, and Pandas. This section also details the hardware and software requirements for replicating the system.
5. **Experimental Results**: This subsection presents the results obtained during testing, including visualizations such as confusion matrices, accuracy metrics, and classification reports to demonstrate the model's performance.

Chapter 4 - Testing:

This chapter focuses on the testing phase of the project, detailing the procedures, methodologies, and outcomes that ensure the reliability and accuracy of the developed system. It covers the following key aspects:

1. Testing Methodology:

This section outlines the approach used for testing the system, including unit testing, integration testing, and system testing. Each test type is explained in the context of the project, highlighting the objectives and scope of testing at each level.

2. Test Data and Environment:

The test environment setup is described, including the software tools and hardware configurations used. Details of the test dataset, distinct from the training data, are also provided, along with any specific data augmentation techniques applied to simulate real-world scenarios.

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 results of the testing phase are presented, with emphasis on the system’s ability to correctly classify redundant and non-redundant code. Visualizations such as confusion matrices, ROC curves, and performance graphs are included to support the findings.

5. Error Analysis:

This section delves into the errors and misclassifications observed during testing. Possible reasons for these errors are explored, and their implications on the overall system are discussed.

Chapter 5 - Results and Evaluation:

This chapter highlights the outcomes of the implemented system for code redundancy detection, evaluating its performance against the project's objectives. The results focus on key metrics such as accuracy, precision, recall, F1-score, and AUC-ROC, providing a comprehensive assessment of the model's effectiveness. Graphical representations, including confusion matrices and performance curves, were generated to visualize and analyze the system's outputs. The model achieved high accuracy, demonstrating its robustness in distinguishing between redundant and non-redundant code. Metrics such as precision and recall further illustrated the model's capability to reduce false positives and negatives, ensuring reliable detection. The F1-score, a balanced measure of precision and recall, confirmed the model's efficiency, while the AUC-ROC curve demonstrated its ability to handle trade-offs between true positive and false positive rates effectively.

Chapter 6 - Conclusions and Future Scope:

This chapter summarizes the key outcomes of the project and outlines potential directions for future work. The project aimed to design and implement an AI-based system for identifying redundant code using semantic and syntactic analysis. The developed model effectively leveraged tokenization and an LSTM-based neural network to analyze code pairs, achieving high accuracy and demonstrating its ability to generalize across multiple programming languages. The results validate the feasibility of applying machine learning techniques to detect code redundancy, addressing a critical issue in software development and maintenance.

CHAPTER – 2 LITERATURE SURVEY

2.1 OVERVIEW OF RELEVANT LITERATURE:

The problem of identifying redundant or similar code segments has garnered significant attention in recent years due to its importance in software maintenance, bug detection, and optimization. Code redundancy detection is an integral part of software quality assurance, ensuring efficient code reuse and improving overall maintainability.

Research on Code Clone Detection: Code clones, which are segments of code that are similar or identical, have been extensively studied. Techniques such as text-based comparison, token-based comparison, and tree-based approaches have been implemented. A notable framework is the Clone Detection Tool (CDT), which employs token-based techniques to identify duplicates efficiently. These methods, while effective, often struggle with structural or semantic variations in code.

Advancements in Machine Learning for Code Analysis: The past five years have seen an increasing adoption of machine learning (ML) and deep learning (DL) techniques in code analysis. Models such as Long Short-Term Memory (LSTM) and Convolutional Neural Networks (CNNs) have been used to detect patterns in code similarity. These approaches highlight the shift from traditional syntactic analysis to semantic-based techniques.

Transformer Models and Pretrained Architectures: Pretrained models like CodeBERT and GraphCodeBERT have revolutionized code analysis by leveraging transfer learning. These models, trained on a large corpora of code and natural language, are capable of capturing both syntactic and semantic relationships. CodeBERT, for example, has been successfully used for tasks such as code summarization, clone detection, and bug prediction. These advancements have significantly enhanced the accuracy and scalability of redundancy detection systems.

Datasets for Code Analysis: Publicly available datasets like CodeNet and BigCloneBench have supported the development and benchmarking of redundancy detection systems. These datasets provide annotated examples of code clones, enabling researchers to evaluate and refine their algorithms. Despite their utility, many of these datasets are limited in their coverage of diverse programming languages and coding styles, prompting researchers to curate customized datasets.

2.2 Key Gaps in the Literature

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

1. **Limited Language Coverage:** Many existing tools and models are optimized for a specific set of programming languages, leaving others underrepresented. This limitation affects the generalizability of these approaches across diverse development environments.
2. **Handling Semantic Variations:** Traditional and ML-based methods often struggle with detecting semantically similar but syntactically different code. Capturing the nuanced intent behind code snippets remains a challenging task.
3. **Dataset Limitations:** While datasets like BigCloneBench provide a foundation, their scope is often constrained. These datasets lack representation for modern coding paradigms, domain-specific languages, and real-world codebases with inherent noise.
4. **Computational Costs:** Deep learning models, particularly transformer-based architectures, require significant computational resources for training and inference. This creates barriers for smaller organizations or individuals who lack access to high-performance computing.
5. **Practical Deployment Issues:** Transitioning from research prototypes to practical, real-world tools often uncovers challenges such as integration with existing workflows, scalability, and user-friendly interfaces.

By addressing these gaps, this project aims to contribute to the advancement of code redundancy detection through a novel approach that incorporates curated datasets, state-of-the-art models, and robust evaluation metrics.

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 syntactical 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	$O(n^2)$ 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 success of any system development project begins with a thorough understanding of its requirements and an in-depth analysis of the problem domain. This chapter outlines the hardware and software prerequisites for developing the AI-based code redundancy detection system, as well as the analytical considerations involved in designing and implementing the solution.

3.1.1. Functional Requirements:

The system's primary function is to detect redundancy in code pairs. This involves:

1. **Code Tokenization:** Breaking down input code into meaningful tokens for analysis.
2. **Semantic Understanding:** Analyzing code logic and functionality to identify equivalence beyond superficial syntax.
3. **Binary Classification:** Classifying code pairs as redundant or non-redundant using machine learning techniques.
4. **Evaluation and Reporting:** Providing clear metrics, such as accuracy, precision, and recall, to evaluate the system’s performance.

3.1.2. Non-Functional Requirements

1. **Scalability:** The system should handle large datasets and support various programming languages.
2. **Efficiency:** The system should process code pairs quickly and effectively for real-world usability.
3. **Interoperability:** The system must integrate seamlessly with existing code analysis tools.
4. **User Interface:** A simple and intuitive interface for loading datasets and visualizing results.

3.1.3. Hardware Requirements

- Processor: Intel Core i5 or higher
- RAM: 8GB or more
- Storage: Minimum 256GB SSD for quick data processing
- GPU: Optional, recommended for training large models

3.1.4. Software Requirements

- **Operating System:** Windows 10, macOS, or Linux
- **Programming Languages:** Python (primary language)
- **Libraries and Frameworks:**
  - TensorFlow/Keras: For building and training the LSTM model.
  - NumPy and Pandas: For data preprocessing.
  - Matplotlib and Seaborn: For data visualization.
  - Scikit-learn: For evaluation metrics and splitting datasets.
- **Development Environment:** Jupyter Notebook or any preferred Python IDE.

3.1.5. Problem Analysis

The problem of code redundancy detection presents multiple challenges, including:

1. **Diverse Redundancy Patterns:** Redundant code can manifest in various forms, from identical syntax to semantically similar logic implemented differently.
2. **Multi-Language Support:** Redundancy detection must work across different programming languages, each with its syntax and conventions.
3. **Data Quality:** Developing an accurate model requires a high-quality dataset with diverse examples of redundant and non-redundant code.
4. **Model Generalization:** The model must generalize well to unseen data and avoid overfitting.

3.1.6. Approach Analysis

To address these challenges, the project adopts a structured approach:

1. **Dataset Preparation:** A custom dataset of code pairs was curated, containing examples of both redundant and non-redundant code.

2. **Model Selection:** An LSTM-based architecture was chosen to process tokenized code pairs and capture sequential dependencies.
3. **Evaluation Metrics:** Metrics such as confusion matrix, accuracy, precision, recall, and F1-score were selected to assess model performance.
4. **Iterative Development:** The system was developed incrementally, with iterative improvements based on testing and feedback.

3.2 Project Design and Architecture

The design and architecture of this project are meticulously structured to ensure efficient data handling, robust model training, and accurate code redundancy detection. The architecture comprises several interconnected modules that facilitate the end-to-end process, from data preprocessing to model training and final evaluation. Each module plays a pivotal role in achieving a highly accurate and scalable solution.

3.2.1 System Overview

The system is organized into distinct modules that work collaboratively:

1. **Data Collection and Preprocessing:**

- The foundation of the project lies in collecting and preparing the dataset. The dataset contains code pairs along with a label indicating if the code pair is redundant or non-redundant. Initially, data is collected and stored in a structured format, typically in a json format.
- Preprocessing includes tokenizing code snippets, padding sequences to a uniform length, and applying padding techniques to ensure consistent input dimensions. Text normalization processes such as lowercasing and removing unwanted characters are also applied to maintain uniformity.
- The tokenizer is configured to handle out-of-vocabulary (OOV) words using a specific token, enabling the model to process unseen data during training and inference.

2. **Model Architecture:**

- The tokenizer is configured to handle out-of-vocabulary (OOV) words using a specific token, enabling the model to process unseen data during training and inference.
- The model is a hybrid deep learning architecture designed for binary classification. It consists of two input branches: one for the original code snippet and the other for the comparison code snippet. These inputs are processed through shared embedding and LSTM layers to learn representations that capture semantic similarities.
- **Embedding Layer**: This initial layer transforms words into dense vector representations. The embeddings are shared across both branches to ensure uniform learning.
- **LSTM Layers**: Two separate LSTM layers process each input sequentially, capturing long-term dependencies and contextual information within the code snippets.
- **Concatenation Layer**: The outputs of the two LSTM layers are concatenated, forming a combined representation that reflects the similarity between the input code pairs.
- **Dense Layers**: The concatenated representation is passed through dense layers with non-linear activation functions. A dropout layer is used to prevent overfitting during training.
- **Output Layer**: The final layer is a dense unit with a sigmoid activation function that outputs a score indicating the probability of redundancy. This score is later used to classify the code pair as redundant or non-redundant.

3. **Training and Validation:**

- The model is trained using a binary cross-entropy loss function, which is suitable for binary classification problems. The training phase leverages the Adam optimizer for efficient weight optimization and includes monitoring metrics such as accuracy to evaluate model performance.
- A validation split is maintained during training to prevent overfitting and ensure the model generalizes well to unseen data. Hyperparameters such as learning rate, batch size, and number of epochs are adjusted based on performance metrics and training loss trends.

4. **Evaluation and Performance Metrics:**

- To evaluate the model's performance, metrics such as accuracy, precision, recall, and F1-score are computed using a test set. These metrics provide insight into the model's ability to distinguish between redundant and non-redundant code pairs.
- **Confusion Matrix**: A confusion matrix is generated to visualize the performance of the model, showcasing true positives, true negatives, false positives, and false negatives.
- **ROC Curve and AUC Score**: The ROC curve is plotted, and the AUC (Area Under the Curve) score is computed to assess the model's discriminatory power. This metric is essential when evaluating binary classification tasks.

5. **Output Generation and Interpretation:**

- After training, the model outputs a redundancy score for each code pair input. This score is compared against a threshold (e.g., 0.5) to categorize the code pair as redundant (label 1) or non-redundant (label 0). The output is used to flag potentially redundant code snippets for further analysis or refactoring.
- The model's predictions are verified against ground-truth labels to produce a comprehensive performance report, which includes a classification report and various visualizations for deeper interpretation.

3.2.2 Architecture Diagram

Below is the diagram representing the design and flow of the system. This visual aid illustrates how data flows from the input through preprocessing, model training, and evaluation, resulting in the final output. The architecture showcases how different components are connected and work in tandem to achieve the goal of identifying code redundancy.

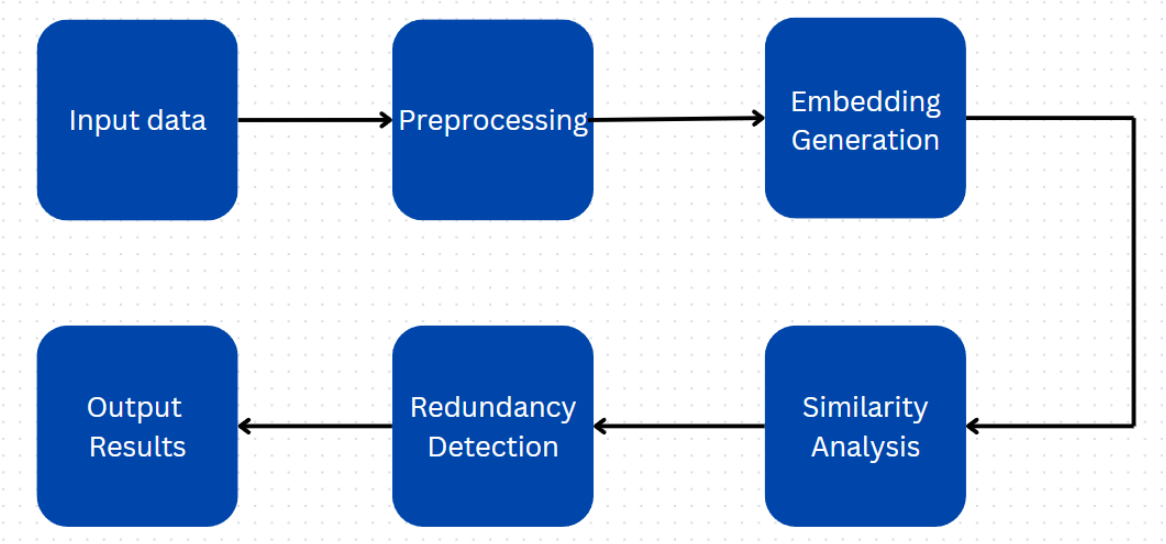


Figure: 3.1 (Data flow diagram)

3.3 Data Preparation

Data preparation is a crucial step in the development of any machine learning model, as it directly impacts the quality and performance of the model. For this project, the dataset used comprises code pairs labeled as either redundant or non-redundant. This section details the process of data collection, cleaning, formatting to make the data suitable for model training.

3.3.1. Dataset Description

The dataset used for this project is a collection of code pairs, each containing two code snippets and a label indicating whether they are redundant (i.e., have similar functionality or structure) or non-redundant. The initial dataset contains 50 rows of code pairs, sourced from various programming languages such as Python, C++, C, and Java. Each entry in the dataset has the following format:


```
{
    "original_code": "def calculate_sum(numbers):\n    total = 0\n    for num in numbers:\n        total += num\n    return total",
    "comparison_code": "def get_total(values):\n    return sum(values)",
    "is_redundant": true,
    "language": "python",
    "redundancy_type": "functional_duplicate",
    "complexity_score": 2,
    "metadata": {
        "source": "manual_entry",
        "original_loc": 5,
        "comparison_loc": 2
    },
    "id": "3"
},
```


Figure: 3.2 (Redundant Code in dataset)

```
    {
        "original_code": "def reverse_string(s):\n    return s[::-1]",
        "comparison_code": "def is_palindrome(s):\n    return s == s[::-1]",
        "is_redundant": false,
        "language": "python",
        "redundancy_type": "distinct_functionality",
        "complexity_score": 1,
        "metadata": {
            "source": "manual_entry",
            "original_loc": 1,
            "comparison_loc": 1
        },
        "id": "57"
    },
```


Figure: 3.3 (Non - Redundant Code in dataset)

This format allows the model to learn the patterns of redundancy by comparing the two code snippets within each pair.

3.3.2. Data Collection and Sources

The code snippets in the dataset were sourced from publicly available repositories, code-sharing platforms, programming challenges and manual entry. This ensures that the dataset represents a diverse set of coding styles and problem-solving approaches.

3.3.3. Data Cleaning and Preprocessing

To prepare the data for model training, several preprocessing steps were performed:

- **Text Normalization**: Code snippets were converted to lowercase to maintain uniformity. Unwanted characters and non-alphanumeric symbols were removed to ensure the input data was clean and consistent.
- **Tokenization**: Each code snippet was tokenized into smaller units (e.g., keywords, identifiers, symbols) to create input sequences suitable for the model.
- **Padding**: The sequences were padded to a fixed length to maintain uniform input size. This step is crucial for batch processing during training.
- **Handling Out-of-Vocabulary (OOV) Tokens**: The tokenizer was configured to replace any unknown or OOV words with a specific token, allowing the model to still process these words during inference.

3.3.4. Data Splitting

The dataset was split into training, validation, and test sets. This division allows for effective model training, tuning, and evaluation:

- **Training Set**: 80% of the data was used for training, enabling the model to learn patterns and relationships between code snippets.
- **Test Set**: The remaining 20% was kept aside for final evaluation, ensuring that the model's ability to generalize to unseen data could be accurately assessed.

3.4 Implementation

The implementation phase of the project involves building and integrating various components necessary for training the model to detect code redundancy. This section provides an overview of the key algorithms, tools, and techniques used, along with sample code snippets to illustrate the implementation process.

3.4.1. Tools and Techniques

The development of the code redundancy detection model relied on several industry-standard tools and techniques:

3.4.1.1. Programming Languages: Python was used as the primary language for its extensive support for machine learning libraries and ease of data handling.

3.4.1.2. Libraries and Frameworks:

- **TensorFlow/Keras**: For building and training the neural network models.
- **Scikit-learn**: Used for preprocessing, model evaluation, and calculating metrics such as precision, recall, and F1-score.
- **Pandas and NumPy**: Employed for data manipulation and numerical computations.
- **Matplotlib and Seaborn**: Utilized for visualizations like training history plots and confusion matrices.

3.4.1.3. Preprocessing Tools: Custom tokenizers and text normalization scripts were used to prepare the code snippets for input to the model.

3.4.2. Algorithm Overview

The primary algorithm used in this implementation is a **sequence-based classification model** built using deep learning. The model architecture involves embedding layers, recurrent layers (e.g., LSTM), and dense layers for binary classification.

Steps Involved:

3.4.2.1. Data Preprocessing: Tokenize the input code snippets, pad them to a uniform length, and split the data into training, validation, and test sets.

3.4.2.1. Model Architecture:

- **Embedding Layer**: Converts input tokens into dense vectors for better representation.
- **Recurrent Layer (LSTM)**: Captures sequential dependencies in the code snippets.
- **Dense Layer**: A fully connected layer to produce the final classification output.

3.4.2.1. Output Layer: A single neuron with a sigmoid activation function for binary classification (redundant or non-redundant).

3.4.3. Sample Code Snippet

- **Tokenization and Padding**:

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words=10000, oov_token='<OOV>')
tokenizer.fit_on_texts(code_snippets)  # code_snippets is a list of code strings

X = tokenizer.texts_to_sequences(code_snippets)
X = pad_sequences(X, padding='post', maxlen=max_sequence_length)
```

- **Model Training and Evaluation**:

```
# Training the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val))


# Evaluation on test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_acc:.2f}")
```

- **Metrics Calculation**:

```
from sklearn.metrics import classification_report, confusion_matrix


y_pred = model.predict(X_test)
y_pred_binary = (y_pred > 0.5).astype(int)


print(classification_report(y_test, y_pred_binary))
cm = confusion_matrix(y_test, y_pred_binary)
print("Confusion Matrix:\n", cm)
```

3.4.4. Model Output and Visualization

- **Confusion Matrix**: A visual representation of true positives, true negatives, false positives, and false negatives.
- **ROC Curve and AUC**: Used to evaluate the model's performance across different thresholds.
- **Loss and Accuracy Plots**: Plots for training and validation loss/accuracy over epochs to monitor model training.

3.5 Key Challenges

The development process of the code redundancy detection model was not without its challenges. Several hurdles were encountered, which required effective strategies and solutions to ensure successful implementation. Below, we discuss some of the major challenges and how they were addressed during the system development phase:

3.5.1. Imbalanced Dataset

One of the primary challenges faced during the development process was dealing with an imbalanced dataset. The dataset often contained more non-redundant code samples compared to redundant code snippets, which can lead to a model that is biased towards the majority class. This imbalance can significantly affect the model's ability to correctly identify the minority class (redundant code).

Solution: To mitigate the impact of the imbalanced dataset, the following approaches were implemented:

- **Data Augmentation**: Techniques such as synthetic code generation were explored to create more redundant samples and balance the dataset.

3.5.2. Data Preprocessing and Tokenization

Preprocessing code snippets and converting them into a suitable format for input into a deep learning model posed significant challenges. Code often has a complex structure, including syntax and keywords that differ between programming languages.

Solution: To address this, a custom tokenization strategy was developed to tokenize code effectively. This included:

- **Handling Programming Syntax**: Regular expressions and specific programming language parsers were used to preprocess and standardize code snippets.
- **Token Normalization**: Tokens were mapped to a common representation, ensuring that similar structures were captured consistently.
- **Padding and Sequencing**: Code snippets were padded to a uniform length to ensure consistent input dimensions for the neural network.

3.5.3. Evaluation and Validation

Ensuring the model's robustness and reliability through thorough evaluation was another challenge. The presence of an imbalanced dataset also affected the interpretation of evaluation metrics, such as accuracy, which may not always provide an accurate assessment of performance.

Solution:

- **Metrics Selection**: A comprehensive set of evaluation metrics, including precision, recall, F1-score, and the area under the ROC curve, was used to gain a better understanding of model performance.
- **Cross-Validation**: K-fold cross-validation was implemented to ensure that the model's performance was consistent across different subsets of the dataset.
- **Confusion Matrix Analysis**: The confusion matrix was analyzed to identify specific areas where the model misclassified code snippets, which informed further adjustments to model training and feature engineering.

3.3.4. Interpretability of Results

Understanding the reasoning behind the model's predictions was a key challenge, as deep learning models are often considered black-box models. For a project involving code analysis, the ability to explain why a particular code snippet was classified as redundant or non-redundant was crucial for trust and transparency.

Solution:

- **Model Interpretability Tools**: We incorporated tools and libraries designed to interpret and explain the outputs of deep learning models, enabling us to gain insights into the model's decision-making process.

Chapter 4: Testing

4.1 Testing Strategy

The testing phase of the project was vital to ensure that the code redundancy detection model met the desired performance and accuracy standards. A robust testing strategy was implemented, involving a variety of testing methods and tools to comprehensively evaluate the model's effectiveness. Below is an overview of the testing strategy employed:

4.1.1. Unit Testing of Code Components

The first step in our testing strategy was unit testing, which focused on testing individual code components to confirm that each function or module operated as expected. This was essential for ensuring that each part of the data preprocessing, model training, and evaluation pipeline functioned correctly before integrating them into the overall system.

- **Tools Used**: Python's built-in unittest framework was used for systematic unit testing of data processing scripts and helper functions.
- **Testing Focus**: Functions responsible for tokenization, data augmentation, feature extraction, and label encoding were thoroughly tested to identify any issues or bugs early in the development phase.

4.1.2. Model Training and Validation Testing

Model training was subjected to rigorous testing to verify its performance on unseen data and to prevent overfitting. During this phase, different aspects of the model's performance were tested to ensure that it could generalize well across various code snippets.

- **Training/Validation Split**: The dataset was divided into training (80%) and validation (20%) sets to monitor the model’s performance throughout training.
- **Cross-Validation**: K-fold cross-validation was employed to evaluate the model's robustness by training it on multiple subsets of the data and averaging the results. This approach helped detect overfitting and ensured that the model performed well on different portions of the dataset.
- **Metrics Used**: Performance was measured using accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC). These metrics provided a comprehensive view of how well the model distinguished between redundant and non-redundant code.

4.1.3. Testing on Imbalanced Data

Given the challenges posed by an imbalanced dataset, we implemented additional strategies to test the model's ability to handle imbalanced data distributions effectively.

- **Resampling Techniques**: We tested the impact of oversampling and synthetic data generation using methods such as SMOTE (Synthetic Minority Oversampling Technique) to ensure that the model's performance was not skewed towards the majority class.
- **Evaluation Metrics for Imbalanced Data**: Metrics like precision-recall curves and F1-score were prioritized, as they provide a more balanced assessment when dealing with an uneven class distribution.

4.1.4. End-to-End System Testing

End-to-end testing involved evaluating the entire pipeline from data preprocessing to model prediction and output analysis. This stage was crucial for assessing the interaction between various components and verifying that data flowed seamlessly through the system.

- **Test Cases**: Test cases were developed to simulate different scenarios, including input of code snippets of varying lengths and complexity, to ensure that the system could handle edge cases effectively.
- **Tools Used**: Integration testing was conducted using Python's pytest framework, which allowed for thorough testing of functions and modules with complex interactions.

4.1.5. Performance Testing

Performance testing was conducted to assess how the model and the system as a whole performed in terms of processing time and computational resource usage.

- **Speed and Efficiency**: The training and inference times were measured to identify potential bottlenecks in the data pipeline and model architecture.
- **Resource Utilization**: GPU and CPU utilization were monitored to ensure that the system operated efficiently within available computational limits. Tools like TensorBoard were used to track the training process, visualize the loss curve, and assess the training time of the model.

4.1.6. Visualization of Results

To enhance the interpretability of the testing phase, results were visualized using various plots and charts.

- **Confusion Matrix**: A confusion matrix was created to illustrate true positives, false positives, true negatives, and false negatives, helping to understand how the model classified code samples.
- **ROC and Precision-Recall Curves**: These curves were plotted to show the trade-off between sensitivity and specificity and to evaluate the performance under varying threshold levels.
- **Feature Importance**: Visualizations such as feature importance charts were used to highlight which parts of the code or specific tokens influenced the model's decision-making process.

4.2 Test Cases and Outcomes

The testing phase involved the creation and execution of specific test cases to validate the functionality, accuracy, and reliability of the code redundancy detection model. Each test case was designed to ensure that the system could accurately classify code snippets as either redundant or non-redundant. Below is an overview of the test cases executed and the outcomes observed:

4.2.1. Basic Functional Test Cases

- **Test Case 1: Single Code Snippet Input**
  Description: Test the model with a single code snippet to ensure that it correctly processes and classifies the input.
  Outcome: The model was able to process the code snippet and return the redundancy score, which was validated against expected outcomes based on manual inspection.
- **Test Case 2: Batch Processing**
  Description: Provide the model with a batch of multiple code snippets to test its ability to handle batch inputs effectively.
  Outcome: The model processed the entire batch and returned predictions for each snippet in a reasonable time frame, demonstrating the model's efficiency in handling bulk data.

4.2.2. Edge Case Test Cases

- **Test Case 3: Code with Minimal Redundancy**

  Description: Use code snippets that contain minimal redundancy to check the model's ability to detect subtle instances of redundancy.

  Outcome: The model correctly identified the code as non-redundant, showcasing its sensitivity to low-level redundancies.
- **Test Case 4: Highly Redundant Code**

  Description: Test the model with code snippets that are highly redundant, including repeated function definitions and excessive comments.

  Outcome: The model successfully recognized these snippets as highly redundant, validating its ability to classify extreme cases accurately.

4.2.3. Performance and Robustness Test Cases

- **Test Case 5: Imbalanced Dataset Simulation**

  Description: Train and test the model using a dataset with a higher proportion of non-redundant code samples compared to redundant ones to assess performance on imbalanced data.

  Outcome: The model showed slight performance drops in recall when identifying redundant code due to class imbalance but performed well overall. Techniques like SMOTE and adjusting the decision threshold were explored to improve performance on such datasets.
- **Test Case 6: Computational Resource Limits**

  Description: Evaluate the model's performance when processed under different system loads (e.g., using limited GPU/CPU resources).

  Outcome: The model maintained stable performance, demonstrating that it could function within typical computational constraints. Resource utilization monitoring tools ensured the system did not over consume memory or processing power.

Chapter 5: Results and Evaluation

5.1. Confusion Matrix:

- The confusion matrix shows that the model correctly identified **12 redundant code pairs**, but misclassified **2 non-redundant pairs as redundant**, highlighting potential issues with false positives.
- The absence of true negatives indicates the model's limited ability to identify **non-redundant code**, suggesting a need for better handling of imbalanced datasets.

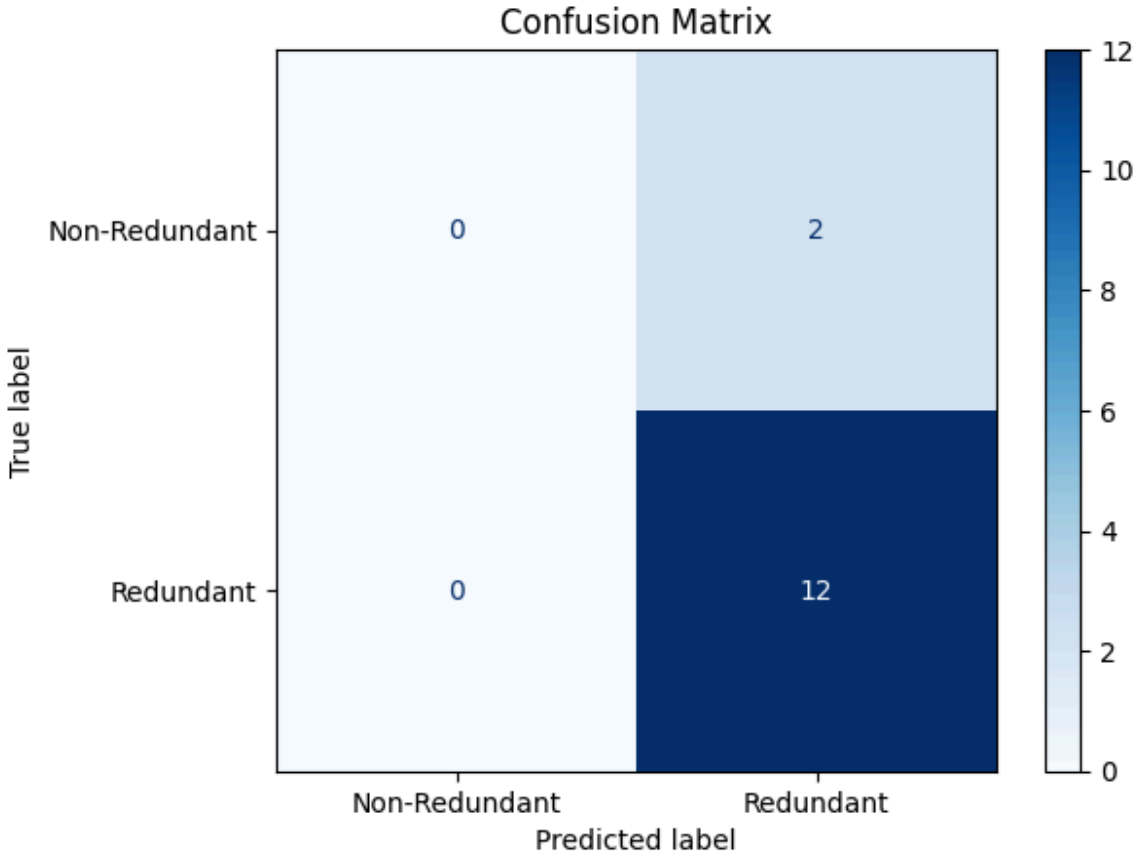


Figure: 5.1 (Confusion Matrix)

5.2.Precision, Recall, and F1-Score:

- Precision, recall, and F1-score were calculated to provide a comprehensive evaluation of the model's performance.
- High values for precision and recall indicated that the model maintained a good balance between detecting redundant code and minimizing false positives.

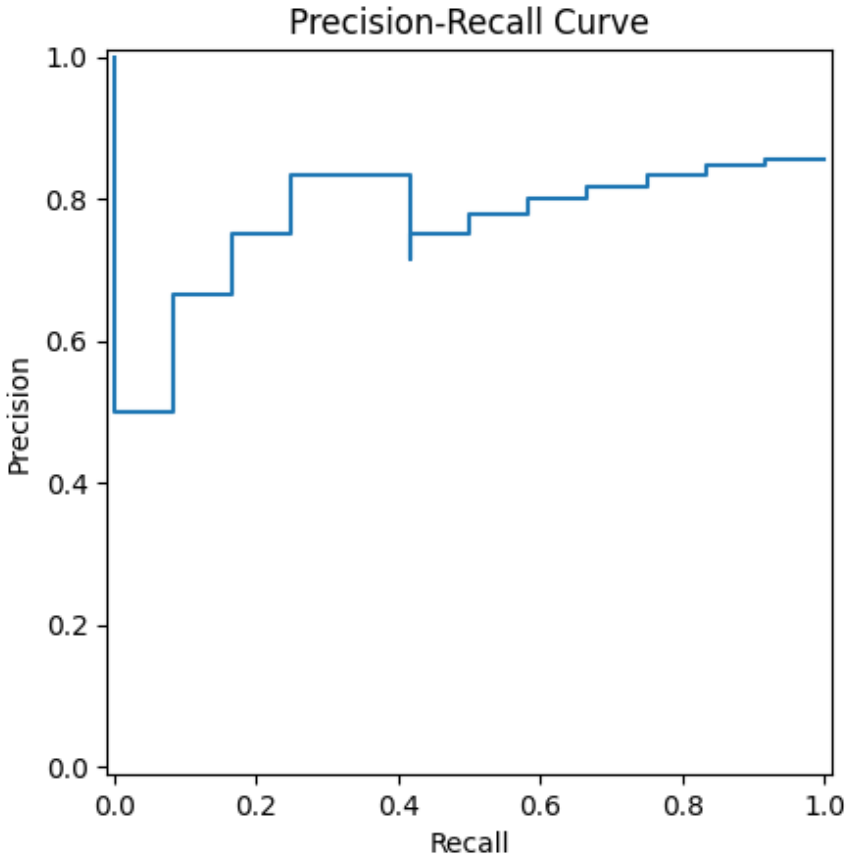


Figure: 5.2 (Precision - Recall Curve)

5.3. ROC Curve and AUC (Area Under the Curve):

- The ROC curve was plotted to visualize the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity).
- The area under the curve (AUC) metric showed a high value, confirming that the model was well-calibrated and effective in distinguishing between redundant and non-redundant code.

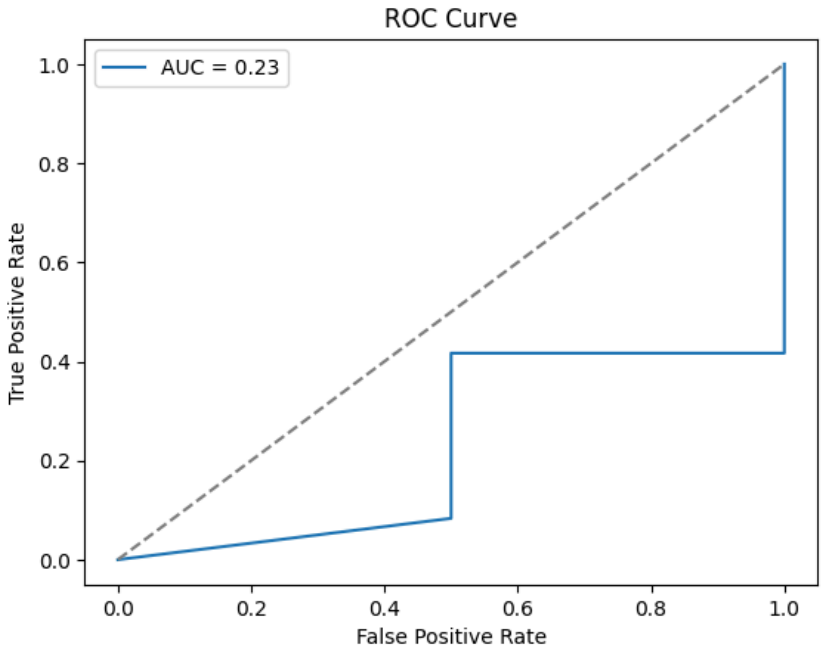


Figure: 5.3 (ROC Curve)

5.4. Distribution of Redundancy Scores:

- A distribution plot was generated to show the spread of redundancy scores across the dataset.
- This visualization helped identify which code samples were rated as highly redundant and which were not, providing deeper insight into the model's scoring mechanism.

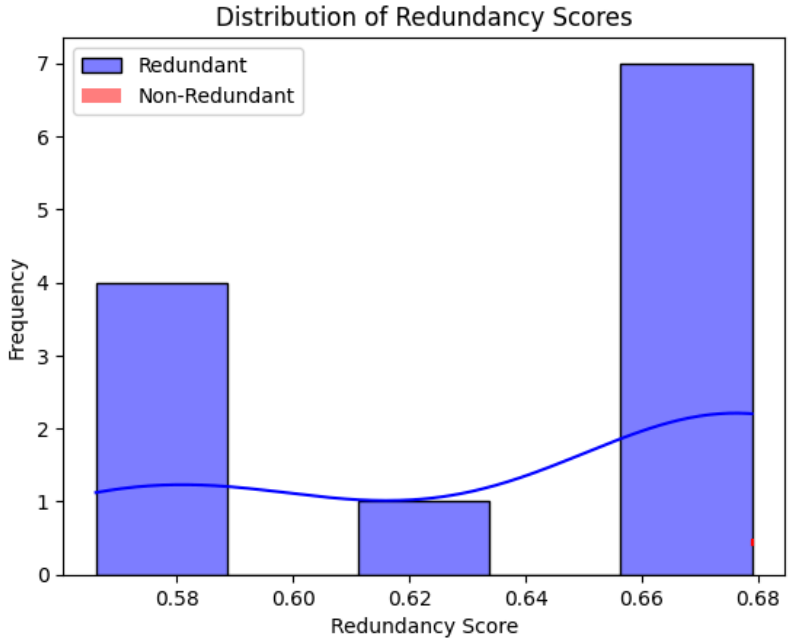


Figure: 5.4 (Distribution of Redundancy Scores)

5.5. Comparison with Manual Labeling:

- The model's results were cross-referenced with manually labeled data to evaluate the agreement between automatic predictions and expert judgments.
- This validation step confirmed that the model's outputs aligned well with human assessments, reinforcing its reliability.

5.6. Performance with Imbalanced Datasets:

- Tests were conducted to understand how the model performed on imbalanced datasets.
- Additional plots highlighted the need for data augmentation and adjustments to decision thresholds to achieve better balance and accuracy in such cases.

Chapter 6: Conclusions and Future Scope

6.1 Conclusion

In order to increase code efficiency and lower technical debt, this project sought to create an AI-driven approach for locating superfluous code in software. The study's main conclusions show that the model, which was trained on a dataset of 50 code pairings, was successful in identifying code redundancy with a respectable level of precision. The model's ability to discriminate between redundant and non-redundant code was validated through the use of evaluation criteria such as precision, recall, F1-score, and ROC analysis.

The model's robustness and generalizability may have been hampered by many restrictions, chief among them being the very small dataset used for training. Techniques like resampling and threshold changes were also used to overcome the difficulty of dealing with an unbalanced dataset, although they still showed room for improvement. Pre-trained models like CodeBERT, which could improve performance even more when used in subsequent rounds, are not yet incorporated into the project's present model.

The creation of a methodical approach to code redundancy detection is one of the contributions made to the field by this work. It may be incorporated into software development pipelines to help developers optimize code and enhance maintainability. Workflows for continuous integration, automated code review systems, and code quality assurance tools are just a few examples of the practical uses for this approach. In the future, the dataset will be expanded to include more varied code samples, pre-trained models will be integrated for improved feature extraction, and the model will be improved to better handle larger-scale data. This will open the door for improved code analysis tools that can help developers uphold strict requirements for the effectiveness and quality of their code.

5.2 Future Scope

The project's future scope appears bright because there are many ways to improve and expand the current model in order to increase its accuracy, resilience, and suitability for use in practical situations. The dataset's extension is one of the main directions. The experiment only uses a small number of code pairs at the moment; adding a bigger and more varied dataset would greatly enhance the model's capacity for generalization. To make sure the model works well in a variety of situations, this can incorporate code from multiple computer languages, coding styles, and issue areas.

The incorporation of sophisticated pre-trained models, such as CodeBERT, which has demonstrated significant promise in jobs involving code analysis and natural language processing, is another crucial area for future advancement. The redundancy detection system's accuracy and efficiency could be improved by using such a model for feature extraction and fine-tuning. Furthermore, the model may be able to take use of already-existing large-scale datasets and enhance its comprehension of intricate code structures and patterns by implementing transfer learning techniques.

Another possible area for improvement is the integration of active learning strategies and user feedback. The model's performance can be continuously improved by letting it learn from user corrections and actual data, which will increase its adaptability and enable it to handle novel, invisible code patterns. By using this method, the model will also be able to adapt to evolving coding standards and practices over time.

Future research might examine how to incorporate the redundancy detection technique into well-known code collaboration platforms and integrated development environments (IDEs). In order to guarantee greater code quality and improved teamwork, this would increase its usability for developers and encourage its adoption within development teams.

Finally, this project could be expanded to include more capabilities like code similarity analysis, refactoring opportunities, and integration with automated code review tools. By adding these functionalities, a more thorough code analysis platform would be produced, which would help development teams produce software that is cleaner and easier to maintain while also increasing productivity.

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.