INVESTIGATING THE IMPACT OF PROGRAMMING STYLES TO IMPROVE CODE QUALITY USING MACHINE LEARNING AND SOCIOLINGUISTIC FEATURES

DEEN MOHAMMAD ABDULLAH
Master of Science, University of Lethbridge, 2020

A thesis submitted
in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

THEORETICAL AND COMPUTATIONAL SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

INVESTIGATING THE IMPACT OF PROGRAMMING STYLES TO IMPROVE CODE QUALITY USING MACHINE LEARNING AND SOCIOLINGUISTIC FEATURES

DEEN MOHAMMAD ABDULLAH

Date of Defence: November 28, 2025

Dr. Jackie Rice Thesis Supervisor	Professor	Ph.D.
Dr. Wendy Osborn Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. John Zhang Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Reid Holmes External Examiner University of British Columbia Vancouver, BC	Professor	Ph.D.
Dr. Andrew Fiori Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

To three extraordinary women in my life -

To my mother, who was my first teacher in life, early childhood, primary and secondary education;

To my wife, whose support, ideas, and encouragement have been instrumental throughout my M.Sc. and Ph.D. journeys;

And to my daughter, whose presence has been my source of strength and inspiration throughout this journey.

This work is for you!

Abstract

In this research we investigated whether sociolinguistic factors such as gender, region, and expertise influence programming styles and code quality. We collected and processed over 700,000 C++ programs from GitHub and Codeforces to build data sets for training Random Forest and BERT models to classify programmer groups. While capturing stylistic patterns, experimental results showed that context-based models outperform metrics-based models. To measure code quality, we combined the Maintainability Index and difficulty metrics to label code as compliant or non-compliant. We further fine-tuned the T5 model for code transformation to generate stylistically improved code. However, due to the limitations of encoder–decoder LLMs, the generated code samples were non-executable. To address this, we developed a CodeBERT-based recommendation model that generates targeted, metric-driven guidance to improve code quality. Finally, we implemented a prototype tool that combines classifications, code quality, and improvement suggestions, providing pedagogically meaningful feedback for learners and researchers.

Acknowledgments

All praise be to the Almighty.

First of all, I would like to thank my academic supervisor, Dr. Jackie Rice, for giving me an opportunity to work under her supervision as a PhD student. With her kindness, caring nature, and continuous support, she has always inspired me during my PhD journey. Dr. Rice has always supported and guided me with her insightful suggestions which helped me in improving my writing and academic thinking. I feel fortunate to work under such a friendly and humble person.

I am also grateful to my committee members, Dr. Wendy Osborn and Dr. John Zhang, for their valuable suggestions, and trust in me throughout my doctoral studies. Beyond reviewing my research, they have supported me in various scholarship applications through their generous references. They have always managed their schedules to attend my committee meetings and examinations. As a co-chair, Dr. Wendy has always helped me secure teaching and TA opportunities. I also greatly appreciate Dr. John—his Data Mining and Deep Learning course was incredibly motivating and sparked my deeper interest in cutting-edge technologies.

I gratefully acknowledge the financial support from the Alberta Innovates – Data Enabled Innovation (AI-DEI) and Alberta Graduate Excellence Scholarship program. I am also thankful to Dr. Rice and the School of Graduate Studies for their financial and academic support throughout this journey.

I want to extend my heartfelt thanks to Dr. Marc R. Roussel. I still remember the day he kindly offered me a coffee while I was fasting—though I didn't explain the reason, his kindness touched me deeply. Speaking with him always relieves me. I am genuinely

thankful for his presence and encouragement, which have helped me reach the finish line.

I sincerely thank all the Department of Mathematics and Computer Science members and my research group, for creating a collaborative and intellectually enriching environment.

Words are less to thank my parents, Dr. Md. Mohiuddin Abdullah and Late Professor Afsir Begum, for their unconditional love, prayers, and unwavering support.

To all my well-wishers—though too many to name here—please know that you are always remembered and appreciated.

Finally, I want to thank my wife Sara, my daughter Shifa, and my son Shezil. You are my powerhouse. Your love, patience, and encouragement have given me the strength to keep going even during the most challenging times. This achievement is as much yours as it is mine.

Contents

Dedication iii

Abstract iv

Acknowledgments v

List of Tables xi

List of Figures xiii

List of Abbreviations xiv

1 Introduction 1
- 1.1 Research Questions and Hypothesis . . . 3
  - 1.1.1 Contributions . . . 8
- 1.2 Organization of Thesis . . . 10

2 Background 12
- 2.1 Sociolinguistics . . . 12
- 2.2 Software Metrics . . . 13
  - 2.2.1 Product Metrics . . . 13
- 2.3 Data . . . 22
  - 2.3.1 Data Sources . . . 22
  - 2.3.2 Data Preprocessing . . . 23
  - 2.3.3 Application Programming Interface (API) . . . 23
- 2.4 Machine Learning (ML) . . . 24
  - 2.4.1 Supervised Learning . . . 24
  - 2.4.2 Classification algorithm: Random Forest . . . 24
  - 2.4.3 Large Language Models . . . 24
  - 2.4.4 Transformer architecture . . . 26
  - 2.4.5 BERT model (Transformer-based Encoder-Only model) . . . 28
  - 2.4.6 CodeBERT Model (Transformer-based Encoder for Code Understanding) . . . 29
  - 2.4.7 T5 model (Transformer-based Encoder-Decoder model) . . . 30
  - 2.4.8 Performance Evaluation Metrics . . . 31
- 2.5 Related work . . . 33
  - 2.5.1 Sociolinguistics and Programming . . . 33
  - 2.5.2 Code Stylometry and Authorship Attribution . . . 35

2.5.3 Software Metrics and Quality Measurement . . . 35
2.5.4 Perceived versus Measured Quality . . . 36
2.5.5 Traditional Models and Transformer-based Architectures . . . 37
2.5.6 Static Analysis and Transformation Tools . . . 38
2.5.7 Summary . . . 38

3 Data Collection and Preparation 40
3.1 Data Collection . . . 40
3.1.1 GitHub Data Collection . . . 44
3.1.2 Codeforces Data Collection . . . 49
3.2 Feature Collection . . . 51
3.2.1 Sociolinguistic features . . . 52
3.2.2 Programmer's skill . . . 55
3.2.3 Software metrics . . . 58
3.3 Data Representation . . . 59
3.3.1 Standardized numeric input values for the Random Forest model . . 59
3.3.2 Vectorizing program (text input) for the LLMs . . . 62
3.4 Data Sets . . . 65
3.4.1 GitHub Data Sets . . . 66
3.4.2 Codeforces Data sets . . . 67
3.4.3 Data set for code transformation . . . 69
3.4.4 Data set for code recommendation . . . 72

4 Experiments and Results 78
4.1 Overview of Experiments . . . 78
4.2 Programming Environment . . . 80
4.2.1 Visual Studio Code (VS Code) . . . 80
4.2.2 Google Colab Pro . . . 80
4.3 Experiment parameters and operational details . . . 81
4.3.1 Phase 1: Classification using Random Forest and software metrics . 81
4.3.2 Phase 2: Classification by fine-tuning BERT model with source code 83
4.3.3 Phase 3: Fine-tuning T5 model to transform non-compliant code into compliant code . . . 85
4.3.4 Phase 4: Code recommendation by fine-tuning CodeBERT model . 87
4.4 Results . . . 88
4.4.1 Gender classification results . . . 88
4.4.2 Region classification results . . . 92
4.4.3 Expertise classification results . . . 95
4.4.4 Phase 3: Code transformation results (Experiment 16) . . . 97
4.5 Overview and next Phase . . . 98

5 Recommendation Model and Tool 100
5.1 Recommendation Model . . . 100
5.1.1 Model Overview . . . 101
5.1.2 Software Metrics and Guideline Derivation . . . 102

5.1.3 Summary of Data Preparation Pipeline . . . 106
5.1.4 Model Training . . . 107
5.1.5 Evaluation and Performance . . . 110
5.2 Code Insights: A Web-Based Code Recommendation Tool . . . 116
5.2.1 Functionality Overview . . . 116
5.2.2 Scope and Expectations . . . 117
5.2.3 Interactive Code Evaluation and Refactoring with Code Insights . . . 118

6 Analysis **125**
6.1 Correlation Between Software Metrics and Code Quality . . . 125
6.1.1 Rationale for Selected Features . . . 126
6.1.2 Numerical Correlation Analysis . . . 126
6.1.3 Graphical Analysis of Relationships . . . 128
6.1.4 Interpretation . . . 128
6.2 Performance comparison between BERT and Random Forest . . . 131
6.3 Analysis of Code Transformation Task . . . 133
6.3.1 Training Performance and Early Stopping . . . 134
6.3.2 Pros and Cons of Encoder-Decoder LLMs (T5) . . . 135
6.3.3 Pros and Cons of Other Approaches . . . 137
6.3.4 Potential for Future Applications . . . 140

7 Conclusion **142**
7.1 Future work . . . 143

Bibliography **146**

A Additional details **156**
A.1 Information gain . . . 156
A.2 Problemsets from Codeforces . . . 158

B APIs **159**
B.1 APIs for accessing GitHub data . . . 159
B.2 APIs for accessing Codeforces data . . . 160
B.3 Feature calculation API . . . 160
B.4 APIs for accessing balanced data set . . . 160
B.4.1 Balanced ternary gender data set (Codeforces) . . . 160
B.4.2 Balanced binary gender data set (Codeforces) . . . 161
B.4.3 Balanced ternary gender data Set (GitHub) . . . 161
B.4.4 Balanced binary gender data set (GitHub) . . . 161
B.4.5 Balanced binary region data Set (Codeforces) . . . 161
B.4.6 Balanced binary region data Set (GitHub) . . . 161
B.4.7 Balanced ternary expertise data Set (Codeforces) . . . 162
B.4.8 Balanced binary expertise data Set (Codeforces) . . . 162
B.5 API for accessing code transformation data set . . . 162

C Undersampling Approach **163**
C.1 Balanced Gender (GitHub) 163
C.2 Balanced Region (GitHub) 163
C.3 Balanced Gender (Codeforces) 166
C.4 Balanced Region (Codeforces) 169
C.5 Balanced Ternary Expertise (Codeforces) 169
C.6 Balanced Binary Expertise (Codeforces) 169

D Sample programs in Data sets **172**

List of Tables

2.1 Comparison between compliant and non-compliant code samples. 20
2.2 Number of operators in code samples. 21
2.3 Number of operands in code samples. 22

3.1 List of features. 52
3.2 Number of instances for each gender label. 54
3.3 Number of instances for each region label. 56
3.4 Number of instances for each ternary expertise class from Codeforces data set. 57
3.5 Number of instances for each binary expertise class from Codeforces data set. 58
3.6 Sample instances from GitHub and Codeforces. 59
3.7 Standardized value of sample instances from GitHub. 62
3.8 Data representation of LLMs, derived from the program sample in Listing 3.2. 64
3.9 Number of instances for each code quality class. 70
3.10 Number of instances for each code quality class. 74
3.11 Constructing the vector of binary labels for the recommendation data set. . 76
3.12 Quality distribution of non-compliant code samples. 76

4.1 Phase 1 – Experiments and data sets for traditional machine learning approach. 79
4.2 Phase 2 – Experiments and data sets for LLM approach. 79
4.3 Phase 3 – Experiment and data set for code transformation. 80
4.4 Phase 4 – Experiment and data set for code recommendation. 80
4.5 Software metrics features. 82
4.6 Gender classification using Random Forest and software metrics. 88
4.7 Gender classification by BERT with source code. 90
4.8 Region classification using Random Forest and software metrics. 92
4.9 Region classification by BERT with source code. 94
4.10 Expertise classification using Random Forest and software metrics. 95
4.11 Expertise classification by BERT with source code. 96
4.12 Performance of Code Transformation model. 98

5.1 Guidelines to improve code quality. 105
5.2 Definition of positive/negative classes and their interpretation in confusion matrices. 111
5.3 Recommendation model performance. 114

6.1 Correlation between code quality and software metrics. 127

6.2 Performance comparison between fine-tuned BERT model and Random Forest (RF) classification model. 132

A.1 Sample data on credit scores and incomes of home buyers. 156
A.2 Frequency of class for 'Buy home' depending on 'Credit score' attribute. . 157
A.3 Frequency of class for 'Buy home' depending on 'Income' attribute. 157
A.4 List of problem sets from the Codeforces. 158

C.1 Programmers distribution for gender label 'male' over Region and Code Quality classes in GitHub. 164
C.2 Programmers distribution for gender label 'other' over Region and Code Quality classes in GitHub. 165
C.3 Programmers distribution for region label 'high GDP' over Gender and Code Quality classes. 166
C.4 Programmers distribution for gender label 'male' over region, expertise and code quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively. 167
C.5 Programmers distribution for gender label 'other' over region, expertise and code Quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively. 168
C.6 Programmers distribution for region label 'high GDP' over Gender, Expertise and Code Quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively. 169
C.7 Programmers distribution for expertise label 'Intermediate' over Region, Gender and Code Quality classes. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively. 170
C.8 Number of instances of binary expertise class from Codeforces data set. . . 171

List of Figures

2.1 Control flow graph of factorial function. 15
2.2 Random Forest Classifier. 25
2.3 The Transformer - model architecture. 27
2.4 Fine tuning BERT model for classification task. 29

3.1 Data collection steps. 41
3.2 GitHub data collection steps. 45
3.3 Codeforces data collection steps. 50
3.4 Choosing compliant code samples for Problem set i, $1 \leq i \leq 50$. 72
3.5 Pairing non-compliant and compliant code approach for Problem set i, $1 \leq i \leq 50$. 73
3.6 Recommendation data set preparation. 75

5.1 Recommendation approach. 101
5.2 Summary of the data preparation pipeline for the recommendation model. . 107
5.3 Confusion Matrix for Halstead Volume (HV) prediction. 111
5.4 Confusion Matrix for Lines of Code (LOC) prediction. 112
5.5 Confusion Matrix for Cyclomatic Complexity (CC) prediction. 112
5.6 Confusion Matrix for Comments prediction. 113
5.7 Confusion Matrix for Difficulty prediction. 113
5.8 Overview of our research workflow. 115
5.9 Interface of *Code Insights* prototype. 119
5.10 Analysis result for the non-compliant factorial code (Listing 5.1). 121
5.11 Updated result after reducing LOC in the factorial code (Listing 5.2). 123
5.12 Final analysis after reducing CC in the factorial code (Listing 5.3). 124

6.1 Scatter plot of lines of code (LOC) versus code quality. 128
6.2 Scatter plot of Halstead volume (HV) versus code quality. 129
6.3 Scatter plot of cyclomatic complexity (CC) versus code quality. 129
6.4 Scatter plot of comments versus code quality. 130
6.5 Scatter plot of difficulty versus code quality. 130
6.6 Training and Validation Loss of Code Transformation Model. 134
6.7 Response from ChatGPT. 138

List of Abbreviations

Abbreviation	Full Form
AI-DEI	Alberta Innovates – Data Enabled Innovation
API	Application Programming Interface
AST	Abstract Syntax Tree
BERT	Bidirectional Encoder Representations from Transformers
BLEU	Bilingual Evaluation Understudy
CC	Cyclomatic Complexity
CPU	Central Processing Unit
CSV	Comma Separated Values
ETL	Extract, Transform, Load
FFN	Feed Forward Network
FN	False Negative
FP	False Positive
GDP	Gross Domestic Product
GPU	Graphics Processing Unit
HV	Halstead Volume
ICT	Information and Communication Technology
LCS	Longest Common Subsequence
LLM	Large Language Model
LLOC	Logical Lines of Code

LOC	Lines of Code
MI	Maintainability Index
ML	Machine Learning
MLM	Masked Language Model
NLP	Natural Language Processing
NSP	Next Sentence Prediction
REB	Research Ethics Board
RF	Random Forest
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
RTD	Replaced Token Detection
SCAP	Source Code Authorship Profiles
SQL	Structured Query Language
T5	Text-to-Text Transfer Transformer
TCPS	Tri-Council Policy Statement
TN	True Negative
TP	True Positive
VS Code	Visual Studio Code

Chapter 1

Introduction

Language is one of the ways that we can express our experiences. The languages that humans use can be divided into two broad categories: natural languages and artificial languages [101]. Research has demonstrated that sociolinguistic variables such as gender, region, and ethnicity influence the use of natural language in society [6, 64]. Similarly, these variables may have other impacts on the use of artificial languages. Programming language is one of the artificial languages. Therefore, variations in the use of programming languages may correlate with sociolinguistic characteristics [2].

Previous studies have demonstrated the possibility of classifying computer programs using metric-based features (e.g., lines of code, cyclomatic complexity, Halstead metrics) and sociolinguistic attributes such as gender and region of the programmer [2, 5, 101, 132]. However, these studies were conducted on relatively small data sets and relied on traditional machine learning models such as Bayes Net, Sequential Minimal Optimization, Classification via Regression, Decision Table, and Random Forest. Building on this line of research, our work employs more advanced machine learning techniques—particularly large language models—to analyze programming styles more effectively at scale, especially when applied to larger and more diverse data sets.

Recently, the widespread availability of online programming tools and forums has made it easier for programmers—especially novices—to access instant code samples and solutions. While helpful, these approaches tend to impart copy-paste skills instead of fostering greater awareness, which can hinder the development of healthy coding habits [15]. We

can assess the quality of a program using various software metrics [62], which measure aspects of maintainability, readability, and complexity. In this work, we examine how such widespread exposure to online resources influences programming styles, and whether software metrics can still differentiate stylistic variation across programmers when applied to a much larger and more diverse data set.

Contrary to existing online resources that typically provide direct solutions or complete code samples, our approach offers guidance that helps users improve the quality and maintainability of their own code. Our goal is not to generate full executable solutions, but rather to provide feedback that helps users produce code that is more maintainable and stylistically consistent—what we refer to as compliant code (described in Section 2.2.1.5). We first experimented with a code transformation model that converts non-compliant code into compliant versions. Although this model performed well on similarity metrics (e.g., ROUGE), many of the generated outputs failed to compile, limiting its usefulness for instructional feedback [46]. To address this limitation, we developed a code recommendation model that analyzes a learner's code and suggests stylistic and structural improvements without overriding the original submission. This approach supports reflective learning [96] and aligns with our broader objective of promoting high-quality coding practices. In addition, our system includes a categorization component that identifies stylistic tendencies associated with demographic and experiential groups such as region and expertise.

In this research, we undertake four main tasks that collectively address these challenges. First, we build classification models using both software metrics and context-based large language models to investigate whether programmers exhibit measurable sociolinguistic stylistic patterns. Second, we construct a code quality measure that labels programs as compliant or non-compliant based on maintainability-related software metrics. Third, we explore code transformation using a fine-tuned encoder–decoder model to generate more maintainable versions of non-compliant code. Finally, to provide practical, pedagogically useful feedback, we develop a CodeBERT-based recommendation model that delivers tar-

geted, metric-driven suggestions for improving code quality. These four tasks form the foundation of the work presented in this thesis.

1.1 Research Questions and Hypothesis

Programming languages are artificial languages, yet they share many features with natural language in how they reflect the practices, preferences, and communities of their users. Prior research has demonstrated that sociolinguistic factors, including gender, region, and expertise influence patterns of communication in natural language [6, 64]. Burnett *et al.* [18] examined how gender differences influence the confidence, problem-solving, and information-processing approaches of programmers. The authors reported that each programmer is an individual with personal programming and problem-solving styles. However, when programmers fall into specific gender clusters, each cluster tends to exhibit certain statistical patterns as a group [18]. In a different research Sharafi *et al.* in [110] suggested that female participants tend to focus more on attention to detail and reducing errors, while male participants often prioritize speed and quick solutions. These tendencies may be reflected in programming styles — for instance, more conservative coding practices, increased use of documentation, or distinct patterns in error handling. These studies have helped us to determine our first research question:

> ***Question 1:*** *Do sociolinguistic variables such as gender and region influence the way programmers write, structure, and maintain their code?*

This question is important for understanding the diversity of programming practices and for designing feedback mechanisms that can guide learners toward more maintainable coding styles. Our research aims to understand and further explore whether sociolinguistic features influence programming styles.

While working with programming languages, a programmer must follow concrete syntax rules that determine the valid structure of source code. Although these rules constrain how programs are written, they do not prevent programmers from producing code that

is difficult to read or poorly organized. The concrete syntax of a program is parsed and transformed into an abstract syntax tree (AST), which compilers and interpreters use for semantic analysis, optimization, and execution [97]. Even with a shared underlying syntax, programmers may adopt different problem-solving strategies and algorithmic approaches, leading to noticeable stylistic variation. For example, one programmer may select a tree data structure and apply tree-traversal algorithms, whereas another may choose a queue and implement a breadth-first search (BFS) traversal.

In addition to algorithmic preferences, programmers also differ in their organization and presentation of code. Some programmers nest logic deeper with nested levels of "if" or loop constructs. Other programmers simplify logic through the use of early returns or flags. The usage of inline comments, block comments, even spacing, or indentation can also vary across programmers. While these choices do not influence the functioning of the program, they might affect how easily other people can understand, maintain, or extend the code. Such stylistic differences may reflect a person's habits, education, or community standards [2, 89, 101, 132].

To identify a programmer's coding style and measure code quality, software metrics are widely used [26, 92, 121]. Oman *et al.* in [92] discussed that lines of code (LOC), nesting depth, cyclomatic complexity, and comments are all influenced by the programmer's stylistic preferences. These software metrics show a quantifiable way to compare the coding styles of different programmers and assess whether these differences are influenced by sociolinguistic factors such as region, expertise, or gender.

In software development, multiple teams usually work together from planning to deployment. In such a situation, one team might prefer using more conditional statements with fewer lines of code (LOC) and comments. Another team might prioritize writing more lines of code with extensive documentation or detailed comments for further maintenance. A company may wish to use software metrics to determine which style is preferable. For example, the Maintainability Index (MI) is a software metric that provides insights into how

maintainable a piece of code is [130]. A higher MI (as described in Section 2.2.1.4) means the written code is likely easy to maintain. In contrast, a lower MI means the written code may be challenging to maintain. Therefore, using this metric, a software company can try to predict future maintenance cost for the code written by different project teams.

To assess code quality, maintainability, and programmer expertise, several studies have combined software metrics with machine learning techniques [49, 115], as discussed in Chapter 2. For instance, a model may be trained using features such as lines of code (LOC), cyclomatic complexity (CC), Halstead volume (HV), or comment density to classify whether a novice or expert programmer writes a code sample, or whether it adheres to maintainability standards. In such cases, machine learning models learn patterns from metric-based features and make predictions on code quality or programmer behaviour.

These studies motivated us to adopt software metrics as a foundation for examining stylistic variation across code samples, which in turn led us to our second and third research questions:

> ***Question 2:*** *Which of the many metrics and features used in previous research correlate best with code readability, maintainability and code quality?*

> ***Question 3:*** *To what extent can traditional software metrics (e.g., LOC, CC, HV, Difficulty, MI) help in classifying different programmer groups?*

However, due to the availability of online tools and forums, programmers are now using coding snippets suggested in such forums, or choosing similar problem-solving approaches [15]. While this trend promotes code reuse and standardization, it also raises questions about whether individual or group-level stylistic patterns in code are consistent. Although traditional approaches rely on software metrics such as lines of code (LOC), nesting depth, cyclomatic complexity, and comments assist in determining stylistic difference [92], Large Language Models (LLMs) like CodeBERT [33] can instead learn directly from raw source code tokens and their contextual representations. Based on these research, we have proposed our fourth research question:

> ***Question 4:*** *Do context-based LLMs outperform traditional software metrics in identifying programmers' stylistic patterns?*

To explore this, we plan to use LLMs and analyze whether programmers—despite potentially using similar logic—continue to exhibit distinct stylistic signatures in how they structure, comment, and organize their code.

The use of software metrics with machine learning also supports the process of code transformation. For example, if a segment of code has high complexity and low maintainability, an automated tool can suggest refactoring actions like simplifying nested logic or adding comments [59]. Recent advances in LLMs have demonstrated their ability to perform code understanding and generation tasks such as translation and transformation [4, 122]. Insights from these studies have helped us to decide our fifth research question:

> ***Question 5:*** *Can stylistic patterns and large language models be used for code transformation to produce more maintainable versions of code?*

Several studies observed that students or learners who use machine-generated code may restrict their own learning and development process. In [36], the authors discussed that automatically generated code may benefit students to achieve better marks, in contrast it may reduce their motivation to understand the problem. Becker *et al.* [15] reported that the learners who use code generation tools may avoid thinking about problem-solving steps. These studies suggest that rather than providing direct solutions, recommendation-based feedback can better support learners in improving code quality and maintainability. Such feedback-oriented learning may be more pedagogically effective in fostering deeper understanding. Building on these perspectives, we have formulated our sixth research question:

> ***Question 6:*** *Can our research generate metric-based recommendations that help learners improve their code in terms of maintainability and quality?*

To add to these contributions, the research in this dissertation necessitated the development of APIs (discussed in Appendix B) to manage the extract, transform and load (ETL)

process for our data. To answer the research questions, we extract data from open-source code repositories and then calculates software metric features from those source code samples. Part of our work includes formulating an equation using software metrics such as LOC, HV, CC, comment and difficulty to predict the code quality. Additionally, this research explores an alternative approach to classifying different programmer groups that may mitigate some of our known threats. These threats include reduced effectiveness of software metrics due to the size of the data sets and the use of online available tools by the programmers. Classification of different programmer groups may help identify labels that can be used to prepare the data set for the code transformation task. While performing code transformation, we plan to fine-tune an encoder-decoder-based LLM (as discussed in Section 4.3.3) to rewrite more maintainable versions, guided by stylistic patterns identified across programmer groups. Finally, we propose an encoder-based LLM (as discussed in Section 5.1) which analyzes non-compliant code and generates targeted recommendations to improve code quality and maintainability.

Based on the research questions outlined above, we propose six hypotheses which are as follows:

Hypothesis 1: Sociolinguistic variables such as gender, region, and programming expertise influence the way programmers structure, document, and maintain their code.

Hypothesis 2: Among the many metrics studied in prior research, a subset—including cyclomatic complexity (CC), lines of code (LOC), Halstead volume (HV), comments, and difficulty—correlates most strongly with code readability, maintainability, and overall quality.

Hypothesis 3: Software metrics (e.g., LOC, CC, HV, Comment, Difficulty, and Maintainability Index) may help in classifying programmer groups using traditional ML models.

Hypothesis 4: Context-based large language models (LLMs) may outperform metric-based models in identifying programmers' stylistic patterns.

Hypothesis 5: Fine-tuned encoder–decoder LLMs may leverage stylistic patterns to

transform non-compliant code into compliant versions while aligning with group-level stylistic tendencies.

Hypothesis 6: Encoder-based LLMs, guided by software metrics, may generate targeted recommendations that help learners improve non-compliant code in terms of maintainability and quality.

In addition to addressing these hypotheses, this work also contributes to the development of APIs for extraction, transformation, and loading tasks related to data management for this and future research involving programming languages. We also present a prototype tool that can compute software features, classify different programmer groups (e.g., region, expertise), and determine whether a code sample is compliant or not. Finally, for non-compliant samples, the tool provides recommendations for improving maintainability and overall code quality.

1.1.1 Contributions

This dissertation contributes to the study of programming styles by connecting sociolinguistic factors, software metrics, and large language models (LLMs). In particular, it explores whether programmer groups defined by gender, region, or expertise leave measurable stylistic footprints in their code, and how these footprints can be analyzed to support maintainability and pedagogical learning. The main contributions of this research are as follows:

- **Data sets and APIs for large-scale analysis:** In this research we prepare ten data sets from two major open-source repositories: 626,706 code samples from GitHub[1] and 116,379 from Codeforces[2]. These data sets are filtered, balanced, and annotated for tasks related to sociolinguistic style classification, code quality assessment, code transformation and code recommendation. To support these tasks, we develop custom APIs that facilitate data extraction, transformation, and loading (ETL). These

[1] `https://github.com/gousiosg/github-mirror`.
[2] `https://codeforces.com/problemset`.

resources can support future research on large-scale code analysis.

- **Code quality formula:** We propose a measure of code quality that combines the Maintainability Index and Difficulty into a score ranging between 0.0 and 1.0. It may potentially help classification, transformation, and recommendation tasks by providing a numerical measure of whether a code sample is compliant or non-compliant.

- **Sociolinguistics in programming:** We analyze the role of sociolinguistic factors—gender, region, and programming expertise—in shaping coding styles. Our results show that such factors help in determining how programmers structure, document, and maintain their code.

- **Classification using software metrics:** We evaluate whether traditional software metrics such as lines of code (LOC), cyclomatic complexity (CC), Halstead volume (HV), percentage of comments, difficulty, and the Maintainability Index can classify programmers into sociolinguistic groups. Our results show that while metrics reveal certain stylistic tendencies, they are limited in reliably distinguishing programmer groups.

- **Context-based analysis with LLMs:** We compare metric-based approaches with a context-based LLM, specifically the BERT model (discussed in Section 2.4.5). BERT learns directly from the raw code samples, enabling it to understand the contextual representations of the code samples. Our experimental results show that BERT achieves higher classification performance than metric-based models in distinguishing programmer groups (e.g., by gender, region or expertise), demonstrating its effectiveness for stylistic classification tasks.

- **Code transformation using encoder–decoder LLMs:** We fine-tune the T5 model (discussed in Section 2.4.7) to transform non-compliant code into more maintainable versions. The transformation is designed to remain consistent with the sociolinguistic

stylistic footprints of the same programmer group. Although the generated outputs achieve measurable lexical similarity with compliant targets, many fail to compile, underscoring the challenges of code-to-code transformation in practical settings.

- **Recommendation model for actionable feedback:** To address the limitations of transformation, we develop a CodeBERT-based recommendation model. Instead of generating entire code snippets, this model identifies deviations in software metrics between non-compliant and compliant code and produces targeted recommendations for improvement. This approach avoids the risks of producing uncompilable code while providing learners with feedback aligned to software metrics such as lines of code (LOC), cyclomatic complexity (CC), Halstead volume (HV), comments, and difficulty.

- **Tool:** We implement a tool called *Code Insight* (discussed in Section 5.2) to demonstrate the usability of our work. The tool visually presents code quality measurements, highlights compliant and non-compliant code, classifies stylistic alignment with programmer groups, and provides targeted recommendations for improvement. Unlike generative code assistants, it preserves the learner's original code and encourages pedagogical learning by showing how to improve maintainability and style.

1.2 Organization of Thesis

The remainder of this thesis is organized as follows.

In Chapter 2, definitions and a literature review are provided to help with understanding of the related studies.

Chapter 3 presents our methodologies for the data collection and preparation process. We describe how we collected and managed data from open-source repositories using our developed APIs. Following this, the data preparation and feature calculation process is discussed.

In Chapter 4, the experimental details and results of code classification and transformation models are described.

Chapter 5 evaluates the effectiveness of our code recommendation model, which provides targeted suggestions for improving non-compliant code without requiring executable output. Finally, it introduces our prototype tool, Code Insights, which integrates these capabilities to support programmers in enhancing code maintainability and style.

Chapter 6 presents a detailed analysis of the results of our model-driven and feature-based experiments. The chapter outlines how sociolinguistic and software metrics contribute to understanding programmer characteristics and code quality. It also discusses the expectations, evaluation, and limitations of our code transformation model.

Chapter 7 concludes the thesis with a summary and offers future research directions.

Chapter 2

Background

This chapter provides the background necessary to understand the methods and analyses used in this research. It begins with key concepts from sociolinguistics and how they relate to programming practices. We then describe the software product metrics used in this work and introduce the code quality measure developed for this research. Next, the chapter presents the structure and preprocessing of our data sets, followed by an overview of the machine learning models applied, including Random Forest and transformer-based large language models. The chapter concludes with a summary of the evaluation metrics used and a review of related work.

2.1 Sociolinguistics

Sociolinguistics can be described as the study of language in relation to society [123]. Sociolinguistics explores the correlation between language and social variables and examines how different social variables such as region, gender, ethnicity, and age influence the use of language in society [56]. William Labov, one of the pioneer researchers in sociolinguistics, introduced sociolinguistics as the study of social diversity in language, language change, narrative, and related areas [67]. Jan-Petter *et al.* [57] also defined sociolinguistics as an attempt to identify correlations between social structure and linguistic structure and to determine any changes that occur due to this correlation. In this research our goal is to investigate whether sociolinguistic variables, including gender and region, show any connection to how programmers use programming languages.

2.2 Software Metrics

A software metric is a mathematical definition mapping the entities of a software system (e.g., methods, classes, packages, program, or lines of code) to numeric values [75]. According to Tu *et al.* [53], "Software Metrics provide a measurement for software and the process of software production. It is giving quantitative values to the attributes involving in the product or the process". These metrics evaluate some attributes of a piece of software that can be either computable or countable [62].

There are three types of software metrics: product metrics, process metrics, and procedure metrics [53, 75]. In this research we are using product metrics as explained in the following section. We selected product metrics over process and procedure metrics because our research focuses on analyzing the characteristics of the code samples and the sociolinguistic traits of the programmers who wrote them, rather than the development process or adherence to procedures.

2.2.1 Product Metrics

Product metrics are used to measure the quality of a software product. These metrics help a team assess the quality of the product and the efforts likely to be required to maintain the final product. For example, lines of code, cyclomatic complexity, and maintainability index are categorized as product metrics [53].

Chowdhury *et al.* in [25] addressed the issue of whether code metrics such as lines of code and McCabe's cyclomatic complexity are still useful indicators for future maintenance. After analyzing 730K Java code, the authors discussed how code metrics still have an impact on maintenance effort. Although these metrics are 30-40+ years old, researchers continue to use them, including developing mappings between these metrics and modern code metrics and languages [25].

The following sections describe the code product metrics focused on in this work-

2.2.1.1 Lines of Code (LOC)

Lines of code (LOC) or total lines of code is one of the software metrics that measures the size of the program. LOC consists of commented and non-commented executable source lines of code [34]. Non-commented lines of code include executable and non-executable statements, headers, variables, conditional declarations. In contrast, commented lines of code include comments and blank lines.

- **Logical lines of code (LLOC):** This is the number of statements in a programming language including executable and non-executable statements, headers, variables, conditional declarations which is also called effective lines of code. Comments and blank lines are excluded from LLOC.
- **Blank lines:** Multiple sequences of whitespace which does not hold any character or number make an empty line which is called blank line.
- **Comments:** When a programmer writes non executable lines of code for documentation purposes (i.e., to provide information on that piece of source code), these lines of code are called comments.

Examples of each metric are given in Section 2.2.1.6.

2.2.1.2 Cyclomatic Complexity (CC)

Cyclomatic Complexity was introduced by McCabe [81] as a measure of the logical complexity of a program. It indicates how many distinct execution paths exist in a piece of code, and therefore how difficult the code may be to test, understand, or maintain. McCabe [81] argued that programs with lower cyclomatic complexity tend to be easier to comprehend and pose less risk when modified, because they contain fewer decision points and fewer possible execution flows.

Cyclomatic complexity is computed using the number of *linearly independent paths* in a program's control flow graph. A linearly independent path is a unique execution path that

introduces at least one new edge not included in any previously counted path. In practical terms, it reflects the number of different routes through a program created by conditional statements such as `if`, `while`, or `for`.

Figure 2.1 shows the control flow diagram of Listing 2.1. In this example, the number of edges is $e = 4$, the number of nodes is $n = 4$, and there is one connected component ($p = 1$). Using McCabe's formula, the cyclomatic complexity is calculated as:

$$CC = e - n + 2p = 4 - 4 + 2(1) = 2.$$

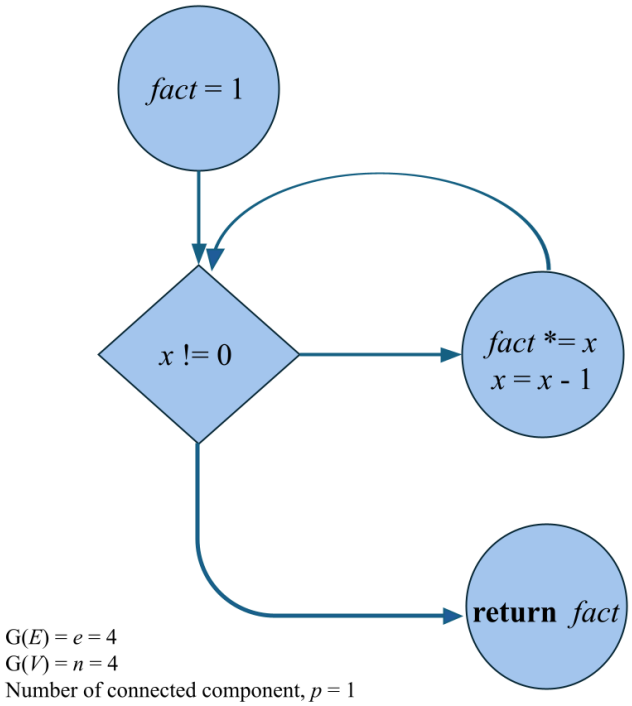

Figure 2.1: Control flow graph of factorial function.

2.2.1.3 Halstead Metrics

Bailey *et al.* [11] discussed Halstead's formulas, which quantify a program's vocabulary, length, and complexity using operator and operand counts [48]. The measures used in this

research are defined as follows.

Program Vocabulary (*n*) represents the total number of distinct operators and operands used in the code.

$$n = n_1 + n_2 \tag{2.1}$$

Actual Program Length (*N*) captures the total occurrences of operators and operands.

$$N = N_1 + N_2 \tag{2.2}$$

Estimated Program Length ($\hat{N}$) approximates the program size expected from its vocabulary.

$$\hat{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2) \tag{2.3}$$

Difficulty (*D*) reflects the cognitive effort required to understand or modify the program.

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2} \tag{2.4}$$

Volume (*HV*) represents the amount of information contained in the implementation.

$$HV = N \times \log_2(n) \tag{2.5}$$

Effort (*E*) estimates the total mental effort needed to develop or comprehend the code.

$$E = D \times HV \tag{2.6}$$

where n_1 is the number of distinct operators, n_2 is the number of distinct operands, N_1 is the total number of operator occurrences, and N_2 is the total number of operand occurrences.

2.2.1.4 Maintainability Index (MI)

Maintainability Index (MI) is a value between 0 and 100 that indicates how easy a piece of code is to understand, modify, and maintain [130]. Higher MI values correspond to better maintainability. In practice, threshold values are used to interpret MI scores; for example, an MI of 50 typically indicates that the corresponding code will require moderate effort to maintain. In this work, we combined the MI equations from both Software Engineering Institute (SEI) and Microsoft© [60], following the approach of Radon [73].

$$MI = max[0, \frac{171 - 5.2\ log(HV) - 0.23\ CC - 16.2\ log(LOC) + 50\ sin(\sqrt{2.46\ perCom})}{171} \times 100] \tag{2.7}$$

where HV is Halstead's volume, CC is cyclomatic complexity, LOC is lines of code, and $perCom$ is the percentage of comments, defined as

$$perCom = \frac{Comments}{LOC} \times 100. \tag{2.8}$$

Heričko *et al.* [52] discussed different threshold values of MI which are explained as below:

$85-100$: High maintainability. The code is very easy to maintain. It is well documented, has low complexity, and is generally easy to understand and modify.

$65-84$: Moderate maintainability. The code is relatively easy to maintain. It may have some complexity or areas that need improvement but is generally well-structured and documented.

$50-64$: Moderate to low maintainability. The code is starting to become difficult to maintain. It may have higher complexity, less documentation, or other factors that make it harder to work with.

< 50: Low maintainability. The code is difficult to maintain. It likely has high complexity, poor documentation, and may be hard to understand and modify.

2.2.1.5 Code Quality

One of the contributions of this work is the formulation of a code quality metric that quantifies the overall maintainability of a code sample using established software metrics. In line with prior

research on code compliance [108]—where compliant code refers to code that adheres to predefined standards, best practices, and structural conventions, and non-compliant code refers to code that deviates from these norms and may hinder readability, maintainability, or performance—we propose an interpretable, probabilistic score ranging from 0 to 1.

In constructing this metric of code quality, we weighted two aspects: the Maintainability Index (MI) and the Difficulty measure. MI is a combined measure that contains four established software attributes: lines of code, cyclomatic complexity, comments, and Halstead volume. As such, it provides a holistic representation of maintainability. In contrast, the Difficulty metric (from Halstead's complexity measures) captures the cognitive effort required to understand or modify code, making it a valuable supplement to MI's structural focus.

While MI is widely used for measuring maintainability, it has certain limitations [26]. It focuses only on structural attributes—such as lines of code, control-flow complexity, comment density, and Halstead volume—and does not directly capture the cognitive effort required to understand the code. For example, two code fragments may have the same MI value, one using longer, linear code with a higher number of lines of code (LOC) and the other using compact logic with more highly nested conditional statements, of higher cyclomatic complexity (CC). Having the identical MI scores, the latter may be far more difficult to understand, especially for newbies. This kind of difficulty is not directly captured by MI alone. We compensate for this by adding Halstead's difficulty measure [47], which considers the number of distinct operators and operands found in the code. Adding this measure reflects how difficult it is to read or understand the code, thus complementing MI and giving a more balanced measure of the quality of code.

We assign a weight of 80% to the Maintainability Index and 20% to Difficulty, indicating the greater range and larger information base of MI. The ratio $80/20$ is a heuristic decision: MI integrates four significant code attributes, while Difficulty quantifies one, but essential, cognitive aspect. The resulting formula is:

$$\text{QUALITY} = \frac{4}{5} \times \frac{\text{MI}}{100} + \frac{1}{5} \times \frac{1}{\text{Difficulty}} = \frac{1}{125} \times \text{MI} + \frac{1}{5} \times \frac{1}{\text{Difficulty}} \tag{2.9}$$

The outcome of this equation gives us a quality score between 0.0 and 1.0. We then set:

- **Compliant code:** Quality ≥ 0.5 — code demonstrating adequate structural maintainability and reasonable cognitive simplicity.
- **Non-compliant code:** Quality < 0.5 — code lacking in maintainability or imposes undue cognitive burden on readers and maintainers.

The 0.5 threshold was selected for interpretability and because it provides a clear midpoint for distinguishing between compliant and non-compliant code. While recent works [52, 113] define maintainability below 50% as low quality, our inclusion of Difficulty necessitated a brief rescaling and review. After initial experimentation with a 0.45 threshold, we opted for a more readable and traditional 0.5 threshold to be able to replicate similar work and ensure readability for future applications.

This definition offers a high-resolution, reproducible means of code assessment. It also serves as the foundation for our classification of code samples as either compliant or non-compliant, which informs both the recommendation and transformation phases of our system.

2.2.1.6 Example

To demonstrate how software metrics impact code quality, we present two versions of the same C++ function in Listings 2.1 and 2.2. The non-compliant version (Listing 2.2) intentionally introduces redundant conditional statements, additional operators, and unnecessary lines of code. These modifications increase cyclomatic complexity (CC), lines of code (LOC), Halstead volume (HV), and Difficulty, all of which contribute to a reduced Maintainability Index (MI) and a lower overall code quality score. In contrast, what we refer to as the compliant version (Listing 2.1) implements the same functionality using more apparent logic, fewer control structures, and more concise syntax, resulting in better software metric values and a higher code quality score.

Table 2.1 shows the respective software metrics for each version, including MI, Difficulty, and the computed code quality value using Equation 2.9. Tables 2.2 and 2.3 show the operators and operands in Listings 2.1 and 2.2. While some algorithms inherently demand higher complexity, this example illustrates how thoughtful implementation—even in algorithmically simple problems—can lead to clearer, more maintainable code.

Here are the detailed calculations for the code sample from Listing 2.1:

```c
// Calculates the factorial of a given number
int factorial (int x)
{
    int fact = 1;

    while(x != 0){
        fact = fact * x;
        x = x - 1;
    }

    return fact;
}
```

Listing 2.1: Compliant version of factorial function.

Table 2.1: Comparison between compliant and non-compliant code samples.

	Listing 2.1	**Listing 2.2**
LOC	12	19
Blank line	2	5
Comments	1	0
Volume	60.23	100.32
CC	2	4
Difficulty	5.2	12
MI	66.04	57.55
Code quality	0.57	0.48

LOC = 12

Blank line = 2

Comments = 1

LLOC = LOC - Blank line - Comments = 12 - 2 - 1 = 9

CC = e - n + 2p = 4 - 4 + 2 * 1 = 2

Operands and operator = n1 = 4, n2 = 5, N1 = 6, N2 = 13

Program vocabulary = n1 + n2 = 4 + 5 = 9

Program length = n1 * log_2 (n1) + n2 * log_2 (n2) = 4 * log_2 (4) + 5 * log_2 (5) = 19.61

N = N1 + N2 = 6 + 13 = 19

Volume (HV) = N * log_2 (Program vocabulary) = 19 * log_2 (9) = 60.23

Difficulty = (n1 / 2) * (N2 / n2) = (4 / 2) * (13 / 5) = 5.2

```
int factorial (int x)
{
    int fact;

    if (x < 0)
         return -1;

    if (x == 0)
         return 1;

    fact = 1;

    while(x != 0){
        fact = fact * x;
        x = x - 1;
    }

     return fact;
}
```

Listing 2.2: Non-compliant version of factorial function.

Table 2.2: Number of operators in code samples.

Listing 2.1			**Listing 2.2**	
=	3		<	1
!=	1		-	2
*	1		==	1
-	1		=	3
n1 = 4	**N1 = 6**		!=	1
			*	1
			n1 = 6	**N1 = 9**

Effort = Difficulty * Volume = 5.2 * 60.23 = 313.19

perCom = Comments / LOC * 100 = 1 / 12 * 100 = 8.33

MI = (171 - 5.2 * log(HV) - 0.23 * CC - 16.2 * log(LOC) + 50 * sin(sqrt(2.46 * perCom))) * 100 / 171 = 66.04

Code quality = (1/125.0) * MI + (1/5.0) * (1 / Difficulty) = 0.57

Table 2.3: Number of operands in code samples.

Listing 2.1			**Listing 2.2**	
factorial	1		factorial	1
x	5		x	7
fact	4		fact	5
1	2		1	4
0	1		0	3
n2 = 5	**N2 = 13**		**n2 = 5**	**N2 = 20**

2.3 Data

Machine learning models help identify patterns in data. A data set consists of multiple instances (rows), and each instance is described by a set of attributes (columns). Attributes may take different forms, such as numeric values, nominal or categorical labels, or text. Numeric attributes contain measurable or countable values, whereas nominal attributes represent discrete categories such as object names, gender, or ethnicity [126]. In this research, source code was treated as text data and then transformed into numeric features (software metrics) and nominal labels (gender, region, and expertise). These representations, along with the original source code, were used for classification, transformation, and recommendation tasks.

2.3.1 Data Sources

The data used in this research was collected from two major open-source platforms: GitHub and Codeforces. GitHub provides a large repository of publicly available C++ projects developed by programmers from diverse backgrounds and levels of experience. In contrast, Codeforces contains competitive programming submissions that reflect time-constrained problem-solving approaches. Using both platforms allows the data set to capture a broad range of programming practices, stylistic tendencies, and levels of expertise.

To collect the data, a set of Web APIs was developed to retrieve C++ source files and associated metadata from each platform. The APIs extract the raw code, user information when available, and other repository details required for feature calculation. These collected data sets form the basis for the subsequent preparation steps described in the following sections and are used throughout this

work for classification, transformation, and recommendation tasks.

2.3.2 Data Preprocessing

Data preprocessing forms the critical component of any machine learning approach as it ensures that the input data is clean, consistent, and well-structured to be fed into learning algorithms. Before training a model, we should handle raw data, especially when collected from various sources, which often contain irrelevant, missing, inconsistent, or noisy values.

In this research we performed *data cleaning*, which involves removing irrelevant, incomplete and duplicate records from the data to retain only valid and meaningful entries for analysis. We then employed *normalization* to scale numerical features into a standard range. Normalization is important for features like lines of code (LOC), cyclomatic complexity (CC), or Halstead volume (HV), which may otherwise dominate the learning process due to their differing scales.

Once cleaned and normalized, the data was structured into feature vectors—numerical representations of each code sample—so it could be used as input to machine learning models. Then we aligned data into learning algorithms, which involves getting data into appropriate format for specific model's requirements (e.g., matrices for supervised learning, sequences for transformers).

Overall, the preprocessing workflow allows raw data to be converted into clean, normalized, and machine-readable data.

2.3.3 Application Programming Interface (API)

An Application Programming Interface (API) is a set of rules, protocols, and tools that allows different software applications to communicate with each other [35]. APIs enable systems to exchange data and functionalities in a structured and standardized way without needing to understand the internal workings of each system. In this work, we developed several Web APIs to fetch data from different data sets as well as calculate features of a given source code. We used the *GET* method to request specific resources from each endpoint, such as raw source code, metadata, and computed metric values. FastAPI (version 0.115.12) [13] was used to implement these endpoints in Python (version 3.13.3). These APIs were necessary to facilitate this work, and we anticipate they will also facilitate future research in this area.

2.4 Machine Learning (ML)

Samuel Arthur, one of the pioneers of machine learning (ML), defined machine learning as a field of study that gives computers the ability to learn without being explicitly programmed [107]. According to Mitchell, machine learning is when a computer program is able to learn from experience E with respect to some task T and some performance measure P, if its performance P improves with the experience E [82, 117]. Popular machine learning approaches include supervised learning, unsupervised learning and reinforcement learning. In this research we utilize a supervised learning approach.

2.4.1 Supervised Learning

In supervised learning a classification algorithm is trained with labeled input data and the learning algorithm generates an inference function to predict the class label of a new data based on the labeled input data. For example, we trained the Random Forest [126] model with input data labeled as 'newbie' or 'skilled' and then used the trained model to predict the expertise of a programmer.

2.4.2 Classification algorithm: Random Forest

The Random Forest algorithm is a supervised machine learning algorithm which creates a forest with multiple decision trees [126]. A decision tree is a structure with a "root node", "connected nodes", "leaf nodes" where all the nodes are connected with "edges". For a given data instance the algorithm uses information gain to select the best feature and then splits the tree into further nodes. Following the same approach the algorithm creates a forest of multiple decision trees and each tree provides a decision. Combining the results of the multiple decision trees the algorithm then predicts a more accurate class label for the given data instance. Figure 2.2 illustrates how the Random Forest model generates n decision trees and predicts the class label for a given instance based on the majority decision of the trees.

2.4.3 Large Language Models

Large language models (LLMs) can perform diversified tasks with excellent efficiency, particularly in natural language processing (NLP) [31]. Extensive training data helps the models produce

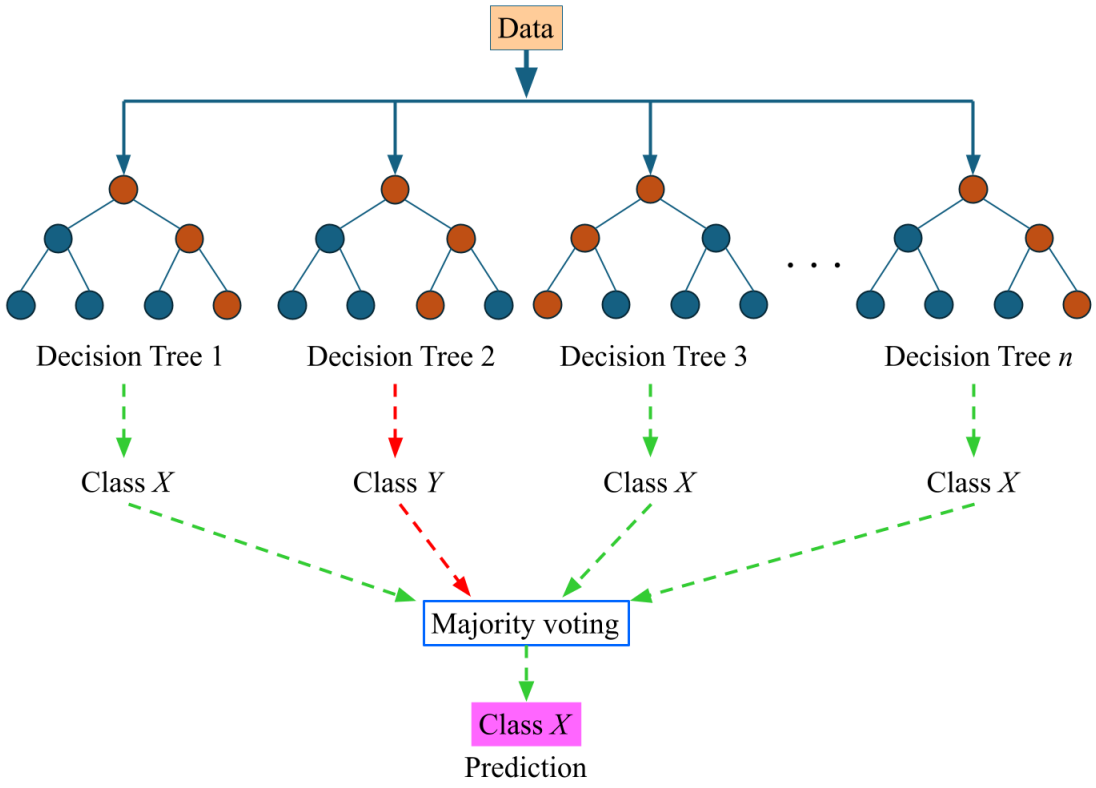

Figure 2.2: Random Forest Classifier.

coherent and fluent text. Additionally, after large-scale pre-training on large text corpora, the LLMs can be fine-tuned for specific tasks. During pre-training, a model is exposed to a large and diverse data set to learn the data's general patterns, structures, and features. A trained model with a diverse data set can be further fine-tuned on a smaller, task-specific data set. Although LLMs were initially pre-trained with text data, they can also be fine-tuned with source code for programming-related tasks [33, 105, 122].

Most of the state-of-the-art LLMs use a transformer architecture [120] where they can be categorized as encoder-only, encoder-decoder, or decoder-only models. This categorization depends on which part of the model the transformer architecture was used to analyze the context during the training process, considering the end tasks [54]. An encoder-only model such as BERT (Bidirectional Encoder Representations from Transformers) [31] is used for understanding the context of the input sequence, an encoder-decoder model such as T5 [102] is used for understanding the context of the input to generate the output sequence, and a decoder-only model such as Llama [116] is used for

generating the output sequence. The encoder-only model BERT analyzes the long input sequence and reaches a conclusion based on the analysis, which is why we chose BERT as an alternative classification model to Random Forest. We also chose the transformer-based encoder-decoder model T5 that may assist in understanding the long input context of the non-compliant code through the encoder, and then generating the targeted compliant code as output through the decoder. In addition, we fine-tuned CodeBERT—an encoder-only transformer pre-trained on both natural language and programming code—for our recommendation model. Further detail and explanation of each of these models are provided in the following subsections.

2.4.4 Transformer architecture

The transformer model was introduced by Vaswani *et al.* [120], and revolutionized the field of natural language processing (NLP) by replacing recurrent and convolutional layers with self-attention mechanisms. This architecture is primarily composed of two main components: the encoder and the decoder. Both encoder and decoder use stacked layers of multi-head self-attention and position-wise feed-forward networks, which allow them to handle sequences in parallel, significantly improving training efficiency and performance.

Together, the encoder and decoder of the transformer model utilize self-attention to detect subtle relationships and enable parallelization, thus making them very powerful for sequence-to-sequence (seq-to-seq) tasks such as translation. A seq-to-seq task is to convert an input sequence into the corresponding output sequence, where the input and output may differ in length or structure. Typical applications are machine translation (e.g., translating a sentence from English to French), text summarization, and code conversion. Fig 2.3 illustrates the overall architecture of the transformer.

2.4.4.1 Input Embedding

The input tokens (e.g., words in a sentence or tokens in source code) are first converted into fixed-size continuous vectors through an embedding layer. These embeddings are continuous in the sense that they are real-valued vectors in a high-dimensional space, allowing the model to capture nuanced relationships between tokens. This allows the model to process and learn patterns over numerical representations instead of raw tokens.

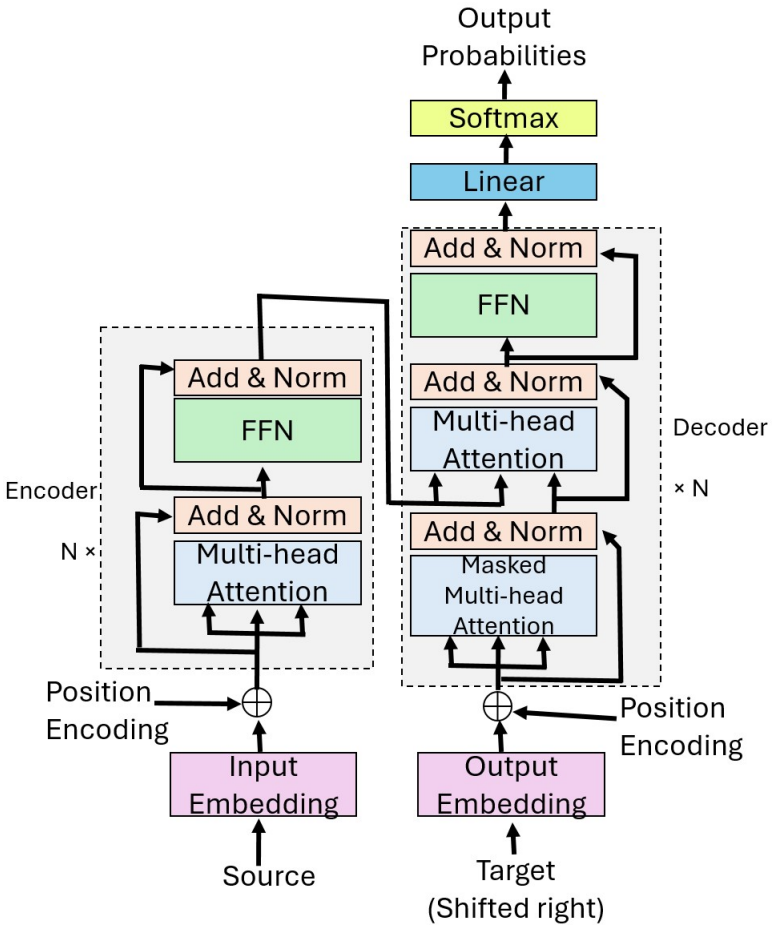

Figure 2.3: The Transformer - model architecture.

2.4.4.2 Positional Encoding

Since the transformer model does not have recurrence or convolution, positional encoding is added to the input embeddings to incorporate information about the position of tokens in the sequence. These encodings are added to the input embeddings before passing them into the encoder.

2.4.4.3 Encoder Stack

The encoder processes the input sequence and converts it into numerical vectors. The transformer based encoder consists of a stack of N identical layers that comprise the following subcomponents in every layer:

Multi-Head Self-Attention: This mechanism allows each word in the input to attend to all other words in the sequence, learning context-dependent representations.

Add & Norm: After each sub-layer, there is a residual connection followed by a layer normalization, which helps to stabilize and speed up training.

Feed-Forward Networks: Each word representation is passed through a fully connected feed-forward network (FFN), applied independently to each position.

2.4.4.4 Decoder Stack

The decoder generates the output sequence based on the encoder's representations. The decoder in transformer is also composed of a stack of N identical layers and contains the following sub-components:

Masked Multi-Head Self-Attention: This mechanism ensures that predictions for a given position can only depend on previous positions in the output sequence, maintaining causality.

Multi-Head Attention over Encoder's Output: This layer enables the decoder to attend to the encoder's output, incorporating information from the entire input sequence.

Add & Norm: As in the encoder, residual connections and normalization are applied after each sub-layer.

Feed-Forward Networks: Similar to the encoder, each position in the decoder is passed through a feed-forward network (FFN).

2.4.4.5 Output Embedding

Output embedding layer maps earlier tokens—ground truth tokens during training or previously generated tokens during inference—to continuous vectors that serve as inputs to the decoder.

2.4.4.6 Linear Layer

The output of the decoder is passed through a linear layer and the softmax function to produce probability distributions over the target vocabulary for every output position.

2.4.5 BERT model (Transformer-based Encoder-Only model)

In [31], Devlin *et al.* introduced BERT (Bidirectional Encoder Representations from Transformers) as a transformer-based model designed to improve natural language understanding. This model uses the bidirectional attention mechanism to understand the context of the given input.

For classification tasks, BERT operates using an encoder-only architecture, where the input sequences are passed through multiple layers of transformers, each consisting of self-attention and feed-forward networks. The final hidden state corresponding to the [CLS] (a classification token added at the start of the input) is used as a representation of the entire sequence. This representation is then fed into a classifier (usually a simple feed-forward neural network) to make predictions such as sentence classification, sentiment analysis, or question answering as shown in Figure 2.4.

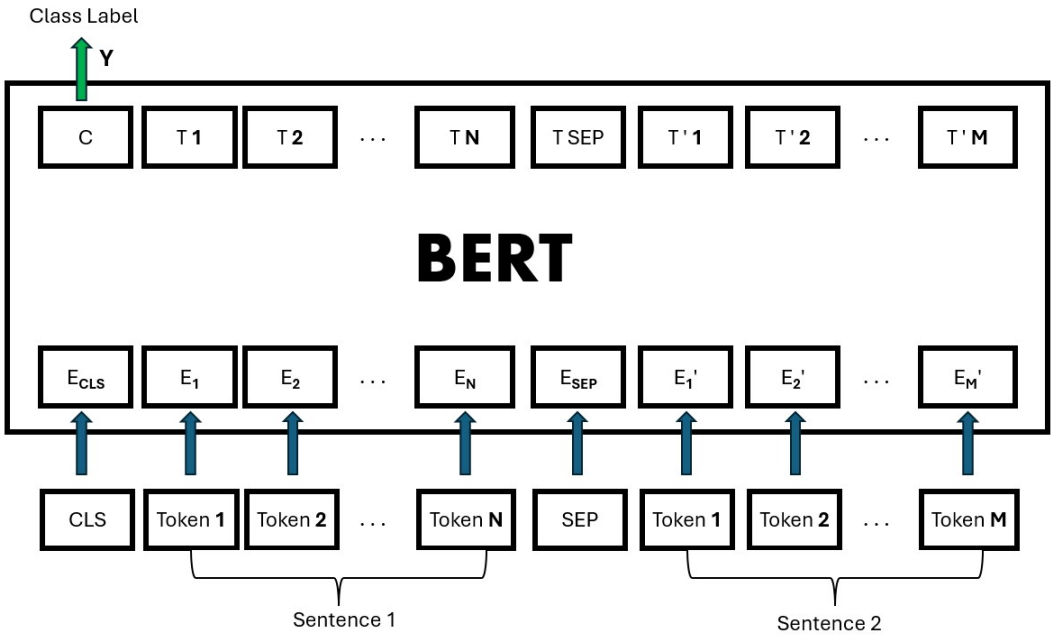

Figure 2.4: Fine tuning BERT model for classification task.

BERT's pre-training involves two key objectives: the Masked Language Model (MLM), where random words are masked and the model is trained to predict them, and the Next Sentence Prediction (NSP), where the model learns to predict whether one sentence logically follows another. After pre-training on a large corpus, BERT can be fine-tuned on a specific classification task with minimal task-specific architecture modification, demonstrating state-of-the-art performance across a wide range of NLP benchmarks.

2.4.6 CodeBERT Model (Transformer-based Encoder for Code Understanding)

CodeBERT [33] is fine-tuned from the BERT model for tasks involving both source code and natural language. While BERT is trained on natural language data alone, CodeBERT is trained on

both natural language documentation and code pairs, which enables it to learn the semantic and structural relationships between them.

Like BERT, CodeBERT uses an encoder-only transformer architecture and utilizes the [CLS] token to generate a representation of the entire input. This makes it directly usable for classification tasks, like our multi-label code recommendation model. CodeBERT was pre-trained using a combination of masked language modelling (MLM) and replaced token detection (RTD) tasks on a large data set of six programming languages, along with their documentation. Such a training method enables the model to effectively capture both syntactic and semantic patterns from the source code.

Since it can understand code tokens in context and project them onto descriptive patterns, CodeBERT is extremely useful for tasks involving code understanding. We utilize CodeBERT in our work to suggest quality improvements in source code by learning from earlier experiences with both compliant and non-compliant code instances.

2.4.7 T5 model (Transformer-based Encoder-Decoder model)

The T5 model (Text-to-Text Transfer Transformer) is a unified framework for a wide range of NLP tasks, introduced by Raffel *et al.* [102]. T5 is based on the transformer architecture and treats NLP tasks as a text-to-text problem, where both input and output are text sequences.

Some key features of the T5 model are described below:

- Unified Framework: T5 formulates different tasks including classification, translation, summarization, question answering as text-to-text problems. For example, a classification task might be transformed into generating a label like ‘positive’ or ‘negative’ from a given input text.

- Pre-training and Fine-tuning: Like BERT, T5 uses a pre-trained model that is fine-tuned on specific tasks. However, the key distinction is its consistent text-to-text setup, allowing the same architecture to be applied to all tasks.

- Transfer Learning: The T5 model leverages transfer learning by pre-training on a large, diverse text corpus C4 (Colossal Clean Crawled Corpus, introduced by Raffel *et al.* [102]) in a

denoising autoencoder setup, and then fine-tuning on task-specific data. This helps the model generalize across different types of tasks.

- Text-to-Text Generation: For text generation tasks, T5 generates a sequence of words based on an input sequence. The input is typically prefixed with a task-specific prompt (e.g., 'translate English to French:') to guide the model's behavior. The model's encoder processes the input text, and the decoder generates the corresponding output text.

The T5 model is a compelling encoder-decoder model for text-to-text tasks and has state-of-the-art performance on a diverse range of NLP benchmarks such as GLUE and SuperGLUE for text classification, and CNN/DailyMail for summarization [102]. This high performance, along with its text-to-text formulation, motivated us to adapt it to our code transformation task.

2.4.8 Performance Evaluation Metrics

This section describes the evaluation metrics used for classification (prediction) and transformation tasks.

2.4.8.1 Evaluation metrics for classification and recommendation tasks

We used accuracy, precision, recall, and F1-score [111] to evaluate the classification and recommendation performance of machine learning models. These metrics are based on the predictions made by a model. There are four possible scenarios when predicting the class of an input sample [111].

- True Positive (TP) — a prediction that correctly classifies a sample as belonging to the positive class.
- True Negative (TN) — a prediction that correctly classifies a sample as belonging to the negative class.
- False Positive (FP) — a prediction that incorrectly classifies a sample as belonging to the positive class when the sample actually belongs to the negative class.

- False Negative (FN) — a prediction that incorrectly classifies a sample as belonging to the negative class when the sample actually belongs to the positive class.

Accuracy: Accuracy is the proportion of correct predictions (both true positives and true negatives) out of all predictions made.

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Samples}} \tag{2.10}$$

Precision: Precision is the proportion of correctly predicted positive instances out of all instances predicted as positive.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}} \tag{2.11}$$

Recall: Recall measures the proportion of actual positive instances that were correctly identified by the model out of all possible instances.

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \tag{2.12}$$

F1-score: The F1-score is the harmonic mean of Precision and Recall. It provides a balance between the two metrics, especially when you need to strike a balance between minimizing false positives and false negatives.

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.13}$$

2.4.8.2 Evaluation metrics for transformation task

ROUGE score: In the context of text generation tasks, ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a widely used set of evaluation metrics that compares the overlap between n-grams (sequences of n words) in the generated text and reference text [74]. Specifically, we utilized ROUGE-N (where N denotes the n-gram size) and ROUGE-L (which measures the longest common subsequence) to assess the quality of our generated content.

ROUGE-N evaluates the overlap of unigrams (ROUGE-1) and bigrams (ROUGE-2) between

the generated text and the reference text, capturing precision, recall, and F1-score. These measures provide insights into the extent to which the generated output preserves word choice and sequence from the reference.

ROUGE-L, on the other hand, computes the longest common subsequence (LCS) between the generated and reference texts. The LCS helps assess the structural similarity and coherence of the generated text. In our study, we reported ROUGE-L recall, precision, and F1-score, which respectively measure the extent of overlap, the proportion of overlap relative to the generated text, and the harmonic mean of recall and precision, providing a balanced measure of the generation quality.

BLEU score: BLEU (Bilingual Evaluation Understudy) score is a widely used automatic machine translation evaluation metric for assessing the quality of machine-generated text in tasks like machine translation, text generation, and summarization [94]. It compares the overlap of n-grams (contiguous word sequences) between the generated text and the reference text. We calculated the BLEU scores by setting equal weights for all four n-grams precision: 1-gram (unigrams), 2-gram (bigrams), 3-gram (trigrams), and 4-gram (4-grams).

2.5 Related work

2.5.1 Sociolinguistics and Programming

Hudson [56] defined sociolinguistics as 'the study of language in relation to society'. The author discussed how the usage of words varies based on the gender of the speaker. Wardhaugh [123] in other research described how sociolinguistics analysis is related to the regional and social dialects in identifying the variations of different groups or classes. For example, the Indonesian students living in USA use English for their academic formal language whereas they use Indonesian (Bahasa Indonesia) for other daily activities. Again, the author noted an example of gender based variation of language by describing how the Spanish speaking population uses Spanish (the official language) and Guaraní (the language of countryside) for communication. He theorized that upper-class males tended to use Guaraní as a sign of friendship, while upper-class females emphasized using Spanish with friends.

Previous studies showed how linguistic variations are connected to the gender of a person. Arg-

amon *et al.* [6, 7, 8, 9] performed various research on natural language based textual documents, used machine learning techniques and conducted different analyses to predict the gender of the author. Similarly, other work has been performed to identify the relationship between sociolinguistics variables and programming languages [2, 5, 101, 114, 124, 132]. Webb [124] analyzed the role of linguistic background in learning programming languages, noting that cultural and educational differences can affect how programmers adopt new syntax and constructs.

Naz *et al.* [89] examined C++ programs and determined the usage of C++ programming keywords and operators based on the gender of the programmers. The authors reported that male programmers used *bool*, /, ==, >= operators, whereas female programmers used *char*, *double* keywords on their written programs. In other work, Rafee [101] used the software metric 'lines of code' to identify the relationship between programming language and the sociolinguistics variables such as gender and region. In addition, Alam [5] further investigated sociolinguistics features with program complexity. The author suggested that female programmers' programs showed higher values for the difficulty measurement than male programmers. Additionally, the use of exception handling functions was higher in male written programs than that of female written programs.

Again, in [2, 132], authors investigated whether sociolinguistics features such as gender and region influence the program writing styles of a programmer. The author developed machine learning models and applied statistical approaches to determine the correlations of the software metrics such as lines of code, Halstead metrics, and cyclomatic complexity. From the analysis the author reported that the female programmers tended to use more logical decisions and write longer programs than the male programmers, and programs from the Eastern region used more memory whereas programs from the Western region showed a higher difficulty value which requires more effort to maintain. In [114], Tasnim categorized computer programs based on the expertise of the programmers. The author showed that expert programmers use more user defined functions, executable statements and blank lines compared with beginner level programmers.

We can see a similar pattern in research on global software engineering. Herbsleb and Moitra in [51] studied distributed teams and found that they often ran into difficulties because developers did not always share the same coding practices, tools, or documentation styles. Noll *et al.* [90] reported the same kinds of challenges, where they noted that in many cases these differences could

be traced back to cultural background or to the kind of training programmers had received. Their work highlights that code is never produced in isolation; programmers inevitably carry their own habits and norms into projects, reflecting the communities they come from.

2.5.2 Code Stylometry and Authorship Attribution

Code stylometry determines unique stylistic fingerprints in source code written by different programmers, including identifier naming conventions, code formatting, control structure usage, and comment styles. Ullah *et al.* [118] proposed a hybrid machine learning approach to capture the structural and stylistic features of the source code. The authors claimed to have high precision in author identification from code samples using semantic and syntactic features. Similarly, Frantzeskou *et al.* [38] utilized the SCAP (Source Code Authorship Profiles) methodology, which relies on byte-level n-gram models to recognize programming style between samples. The authors discussed how their SCAP approach successfully identified a particular author's pattern in source code that was contributed to by multiple authors. While these studies focused primarily on authorship attribution, the same stylistic markers often reflect broader sociolinguistic traits, linking this line of work to our interest in demographic and regional programming styles.

Code stylometry has several practical uses. In forensics, it helps attribute malicious code such as ransomware or malware to suspected developers, mainly when investigators cannot rely on content-based clues [133]. In education, stylometric methods aid in plagiarism detection by flagging when a student's submission carries stylistic similarity to past solutions, even if the code has been altered [19]. In security, researchers have suggested using it for insider threat detection, since an unfamiliar style can stand out within a project [3]. These applications show that code style carries information about authorship and identity that goes beyond correctness or logic.

2.5.3 Software Metrics and Quality Measurement

Software metrics research provides another foundation for evaluating code quality and style. McCabe's cyclomatic complexity (CC) [81] is a widely used metric that provides a graph-based count of independent control paths. A high CC score usually means a module will be harder to test or maintain. Halstead's metrics [47] take a different approach, focusing on operators and operands

to calculate vocabulary, length, volume, difficulty, and effort. Simple metrics such as LOC and comment density have also been used widely as indicators of program size and readability. Among these, the Maintainability Index (MI) is especially relevant because it later serves as the foundation for our composite quality score that integrates structural and cognitive aspects of code.

Welker *et al.* in [125] proposed the Maintainability Index (MI) that measures whether a source code will require minimal or more efforts during future maintenance. MI combines several metrics—including Halstead volume, CC, and LOC—into a single composite score [125]. High MI values suggest easier maintenance, while low values imply more difficulty and cost. Coleman *et al.* in [26] demonstrated that MI could be used to predict maintenance effort in practice, and Oman and Hagemeister [92] validated it in large systems. Later works such as Heitlager [50] and Núñez-Varela *et al.* in [91], confirmed that MI is both practical and interpretable, which explains why it has been integrated into tools like Microsoft Visual Studio and SonarQube.

While MI is widely used for measuring maintainability, it has certain limitations [26]. Different pieces of code can receive the same MI score while differing vastly in readability. A long, linear piece of code may score the same as a short but highly nested one, though the latter is often much more challenging to read. It has been observed that MI emphasizes structural properties while overlooking aspects of cognitive effort. To bridge this gap, researchers have suggested augmenting MI with additional measures such as Cognitive Complexity, which better reflect human comprehension demands. Barón *et al.* in [84] validated Cognitive Complexity as a metric aligned with understandability, and Lavazza *et al.* in [68] further evaluated its added value in prediction models compared to traditional metrics. Consistent with these insights, our work integrates MI with Halstead's difficulty to create a more nuanced assessment of code quality.

2.5.4 Perceived versus Measured Quality

Metrics provide objective scores, but developers also make subjective judgments about quality. Spinellis [112] emphasized that software quality always has to be balanced against cost and delivery time. Curtis *et al.* [27] noted that developers sometimes disagree with metric based assessments, especially when readability or style is at stake. A heavily commented program might look maintainable on paper, but feel cluttered in practice. In contrast, concise code may be admired by ex-

perienced programmers but penalized by metrics for lacking documentation. This divergence points to the importance of pairing computed scores with feedback that developers can readily interpret, which shaped our approach to metric-based recommendations.

This division highlights the need for feedback that bridges numbers and perception. Rather than just showing a low MI score, developers are more likely to act on a clear suggestion such as "reduce nesting for readability". Studies in human–computer interaction show that feedback is more persuasive when it is actionable and interpretable [70]. Our work builds on this insight by not only calculating metric-based scores but also framing them in ways that align with how developers perceive code quality.

2.5.5 Traditional Models and Transformer-based Architectures

Traditional machine learning techniques have made substantial contributions to the field of code analysis. Random Forests [17] and related classifiers have been widely applied to defect prediction [71, 100], maintainability classification [121], and sociolinguistic prediction [2, 132] tasks. These models rely on structured, software metric features, whereas transformer-based approaches can learn directly from raw source code tokens and representations.

A significant development in recent years has been the adoption of transformer-based architectures [120], capable of learning directly from raw source code. BERT [31] and its adaptations for programming tasks—such as CodeBERT [33] and CodeT5 [122]—have shown strong performance in classification, code understanding, and generation. Ahmad *et al.* [4] demonstrated the value of unified pre-training across code tasks. In contrast, Kulal *et al.* [65] and Chakraborty *et al.* [21] showed how transformer-based models can be applied to pseudocode translation and code editing. Bairi *et al.* [12] pushed this further by integrating planning into repository-level coding with LLMs. In education, MacNeil *et al.* [79] and Nam *et al.* [87] studied how LLMs can assist in learning and comprehension tasks, while Mastropaolo *et al.* [80] examined automated feedback for students. Together, these advances indicate that transformer-based models can address a wider range of tasks than traditional models, from sociolinguistic classification to code recommendation.

2.5.6 Static Analysis and Transformation Tools

Static analyzers and linters such as PMD, Checkstyle, ESLint, Clang-Tidy, and SonarQube have long been used to enforce coding standards and identify defects [128]. They work well for detecting surface-level issues but struggle with flexibility, as they rely on fixed rule sets [85]. Research has extended beyond rule enforcement toward code transformation. In [10], Bagge *et al.* designed a code transformation framework called CodeBoost that transforms and optimizes C/C++ code by using abstract syntax trees (AST) which represents the structure of a program. Gupta *et al.* in [46] applied deep learning in DeepFix to repair common errors in C programs, though often producing non-compilable code. However, many of these systems struggled with producing compilable outputs, a challenge that influenced our decision to pursue recommendations rather than direct code generation. Recent studies have incorporated LLMs for repair and generation such as Code Llama [105], CodePlan [12], and Codit [21].

In addition to research on code stylometry and transformation, several intelligent tutoring systems and code feedback platforms have been developed to provide learners with interpretable, actionable guidance. AutoStyle [24] automatically reformats code to match a desired style (such as indentation, brace placement, spacing, and naming adjustments), using program analysis and transformation rules to align with instructor-specified conventions. The system aims to help students internalize stylistic best practices without altering the underlying program logic. OverCode [43] aggregates large sets of student solutions, clusters them according to structural similarity, and allows instructors to provide feedback at the cluster level. This approach reduces instructor workload while ensuring consistent and broadly applicable feedback for learners. Both tools demonstrate how automated analysis and structured feedback can support learning and style adoption at scale—concepts that inspired the design of our *Code Insights* tool (discussed in Section 5.2).

2.5.7 Summary

Research across sociolinguistics, code stylometry, software metrics, and tools shows that programming style varies across contexts and developers. It can reflect demographic influences, encode authorship, and shape both measured and perceived quality. Metrics provide structure but often diverge from developer intuition. Traditional ML models have leveraged these metrics effectively, but

transformer-based architectures expand the scope by learning directly from raw code sample and enabling classification, transformation, and recommendation tasks. Static analyzers and feedback systems continue to play an important role but are gradually being complemented by more adaptive, learning-based approaches.

Our research builds on these insights. We address the lack of large sociolinguistically labelled data sets, propose a composite quality measure that blends MI with difficulty, and develop a recommendation model that does not rely on code generation, thereby sidestepping the challenges encountered in code transformation. Our work combines sociolinguistic perspectives, software metrics, and transformer-based recommendations, creating a new way to connect social factors, quality evaluation, and useful feedback in programming research.

Chapter 3

Data Collection and Preparation

This chapter describes how we collected our data, what challenges we faced, and how we solved them. It also discusses the initial and balanced data sets from the two data sources, how we collected the relevant features, and how we represented the data for machine learning tasks.

3.1 Data Collection

The goal of this research is to investigate whether sociolinguistic factors influence programming styles and to evaluate how software metrics and learning models can provide meaningful insights into code quality. Therefore, we required large-scale, sociolinguistically labelled code samples that could support both statistical analysis and machine learning experiments. The data sets we prepared serve as the foundation for exploring classification tasks, code transformation experiments, and the development of our recommendation model.

Figure 3.1 represents an overview of our data collection and preparation process. Sections 3.2.1 and 3.2.2 explain the class labels. We developed APIs (discussed in Appendix B), applied different cleaning techniques and represented the data as a series of numeric vectors to prepare the data for training the selected machine learning tools.

This research uses data from two publicly available websites: GitHub[3] and Codeforces[4]. We chose these sites because they provide open access to source code submitted by users along with the user data, including voluntarily shared attributes such as name and region. Unfortunately other

[3] `https://github.com/gousiosg/github-mirror`.
[4] `https://codeforces.com/problemset`.

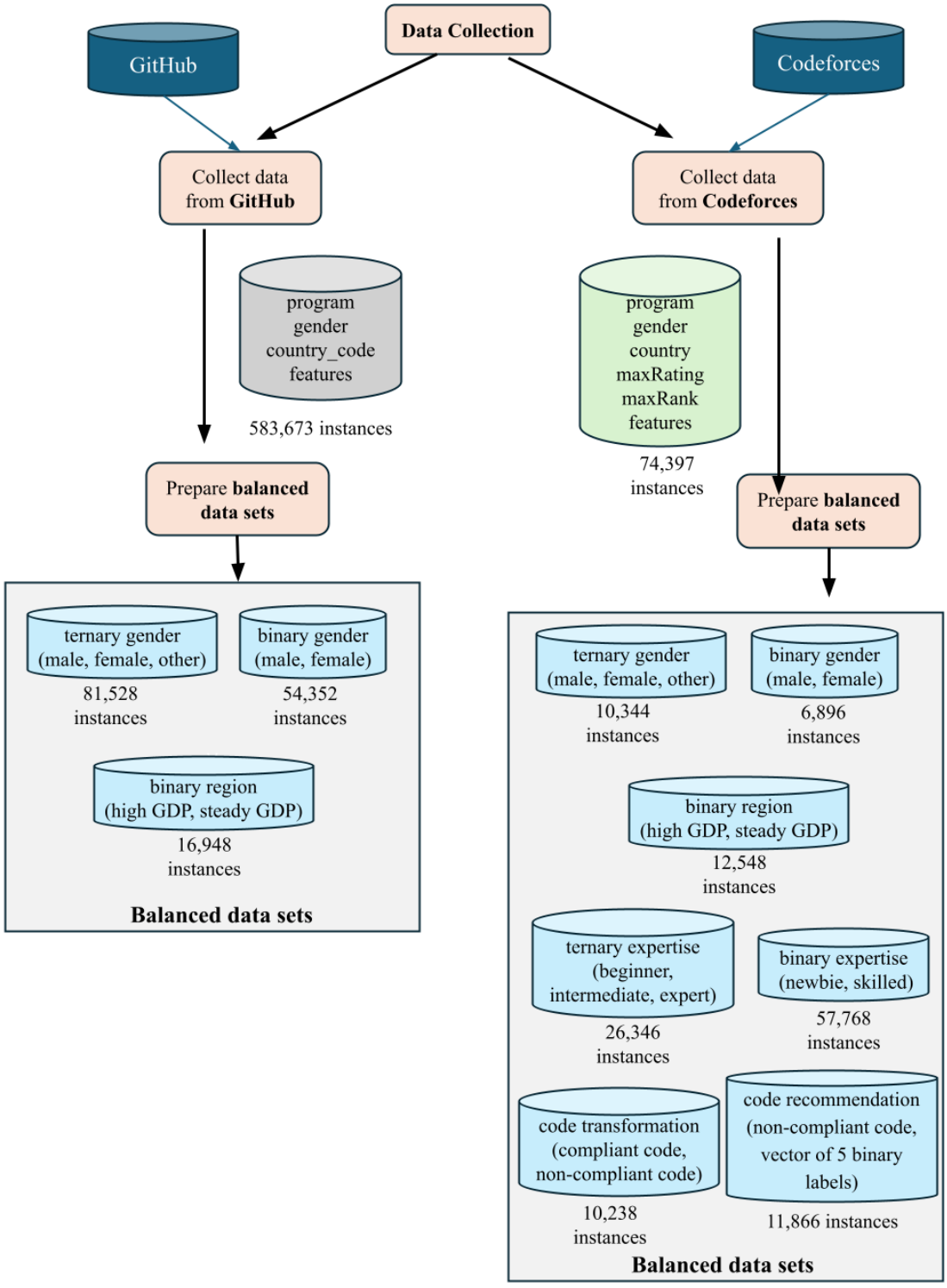

Figure 3.1: Data collection steps.

programming sites such as CodeChef[5], LeetCode[6], and HackerRank[7] restrict access to source code and user metadata, while platforms like Bitbucket[8] and GitLab[9] primarily host private repositories.

Prior research has utilized these platforms to investigate the relationships between programming behaviour and sociolinguistic factors such as gender, region, and expertise [2, 5, 114, 132]. In line with this, we gathered code samples and associated author metadata for analysis of how sociolinguistic characteristics might influence coding style across specific and diverse problem sets. For our research, we analyzed C++ code samples as it is a widely used programming language [28] for different programmer groups, including learners and experts.

Importantly, this study involved no direct interaction with individuals, and no attempt was made to identify or re-identify users. All data were publicly available, anonymized where appropriate, and analyzed only in aggregate. By Article 2.2 of the Tri-Council Policy Statement (Ethical Conduct for Research Involving Humans - TCPS 2, 2022)[10], this research is exempt from Research Ethics Board (REB) review since it relies solely on publicly accessible information for which there is no reasonable expectation of privacy.

We faced challenges when extracting, cleaning, and processing the data. GitHub and Codeforces only provide programmers' name and county information. After obtaining the first name of each programmer, we used genderize.io[11] to determine gender based on the probability that their first name was tied to a specific gender. This API predicts the gender (male or female) of a person based on their first or full name and provides a confidence score (probability) for each prediction [109, 119]. It returns 'nil' for unclassified names and has been widely used in gender-sensitive computational studies, achieving prediction accuracies of up to 96.6% [119].

We obtained country information for programmers from both GitHub and Codeforces, allowing us to determine their regions. The Codeforces data also includes rankings of the programmers, which we used to categorize the expertise of each programmer.

During data cleaning, we removed only those instances for which both the country information

[5] https://www.codechef.com/.
[6] https://leetcode.com/.
[7] https://www.hackerrank.com/.
[8] https://bitbucket.org/.
[9] https://gitlab.com/.
[10] https://ethics.gc.ca/eng/policy-politique_tcps2-eptc2_2022.html.
[11] https://genderize.io/.

and gender information were missing. However, when at least one of these demographic attributes was available, the instance was retained. In particular, entries with missing gender but valid country information were kept, as they were required for later analyses and were included as part of the 'other' category in our ternary gender classification data sets. Then we calculated 12 programming features (discussed in Section 2.2.1) from the source code samples:

- LOC,
- Blank lines,
- CC,
- Vocabulary,
- Program length,
- LLOC,
- Comments,
- Volume,
- Difficulty,
- Effort,
- MI, and
- Quality.

After preparing the data and adding the features, the resulting data set from GitHub contained $583,673$ instances while the resulting data set from Codeforces contained $74,397$ instances, as shown in Figure 3.1. Then we prepared five balanced data sets from Codeforces for training classification tasks:

- balanced ternary gender (male, female, and other)
- balanced binary gender (male, female)

- balanced binary region (high GDP, steady GDP)
- balanced ternary expertise (beginner, intermediate, and expert)
- balanced binary expertise (newbie, skilled)

We also prepared two data sets from Codeforces for code transformation and recommendation tasks:

- code transformation (compliant, non-compliant code pairs)
- code recommendation (non-compliant code with metric-based vectors)

Additionally, we prepared three different data sets from GitHub for training classification tasks:

- balanced ternary gender (male, female, and other)
- balanced binary gender (male, female)
- balanced binary region (high GDP, steady GDP)

Although we recognize that higher-level structural characteristics such as coupling, directory structure, and library dependencies enhance software maintainability, these attributes are most relevant in large, collaborative, and sometimes enterprise-scale codebases [72]. In this research our focus is on individual coding practice rather than team-based or organizational software development practices. By obtaining problem-specific programs from Codeforces and diverse, single-user-authored programs from GitHub, we aimed to achieve balanced data sets that capture both structured and unstructured coding environments such that our model can generalize across a wide range of real-world coding tasks, from formulaic problem-solving to open-ended development contexts. Since the goals of our work include style classification, code transformation, and recommendation, we deliberately emphasized stylistic and structural patterns at the function level rather than higher-level architectural decisions.

3.1.1 GitHub Data Collection

To capture a broad range of real-world coding styles, we collected diverse, single-user-authored C++ programs from GitHub, representing unstructured coding environments that complement the more structured sources in our study. To create our GitHub data sets, we collected information about

projects, including project URLs and their author's information from GitHub public repositories. We next collected the users' names and used genderize.io to identify their gender. Finally, we extracted C++ programs for each project URL and added those programs to our data set. Figure 3.2 represents the GitHub data collection and preparation process.

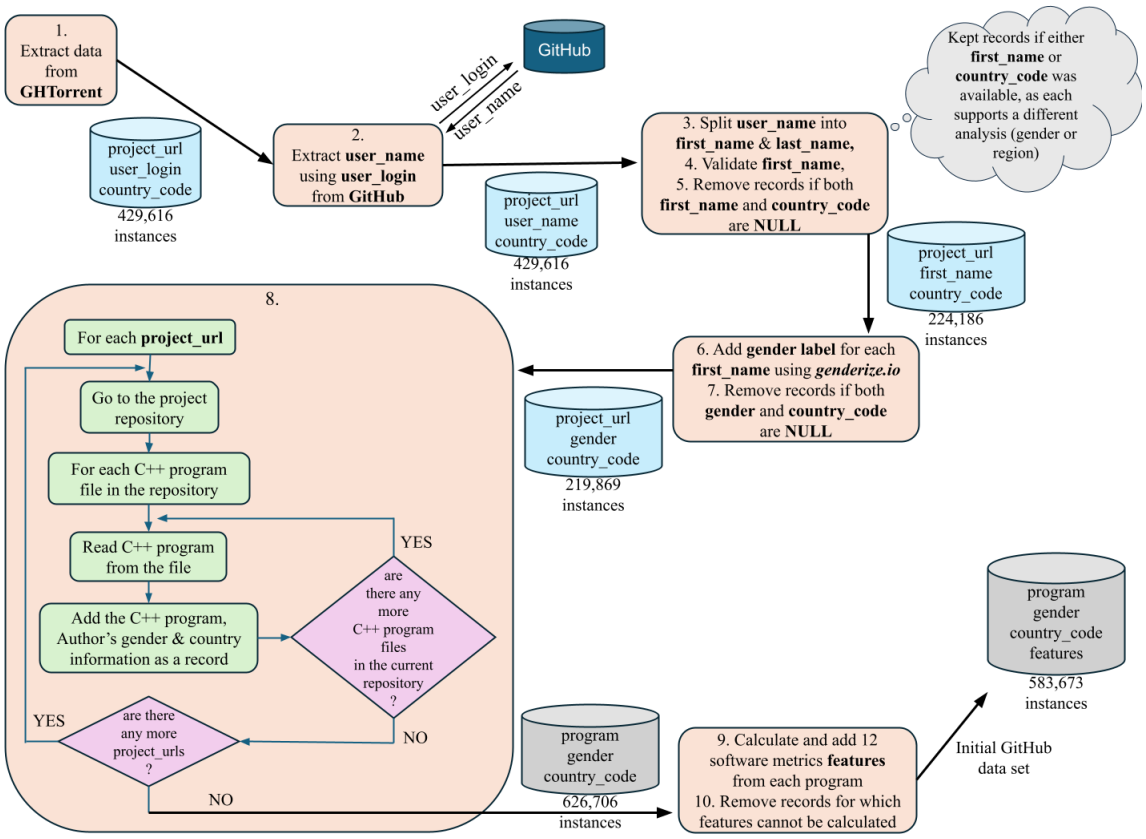

Figure 3.2: GitHub data collection steps.

Due to the nature of public GitHub repositories, questions may arise regarding the ownership of code in projects with multiple contributors. To resolve these complications, we limited our data extraction to repositories owned by personal user accounts (rather than those owned by organizations), as indicated in our SQL statements (users.type = 'USR', shown in Listing 3.1). As we focused on capturing code associated with a single primary maintainer, we assumed the repository owner had substantial authorship or oversight over the code. Since calculating Halstead metrics requires at least one function in an error-free, executable file [48], we excluded any files (including machine-generated files [61]) for which these metrics could not be computed during the feature calculation

step, retaining only functionally valid and analyzable C++ source code samples.

3.1.1.1 Data extraction from GitHub public repositories

GHTorrent [45] stores the information for GitHub public projects and commits. We used Google's BigQuery [44] to extract initial data from GHTorrent and save data to a CSV (Comma Separated Values) file. After executing the query (shown in Listing 3.1) using BigQuery, we received 429,616 instances from GHTorrent. Then, we started to extract the names of users for each instance.

```
SELECT
    users.id as users_id,
    users.login as users_login,
    users.created_at as users_created_at,
    users.type as users_type,
    users.country_code as users_country_code,
    projects.id as projects_id,
    projects.url as projects_url,
    projects.name as projects_name,
    projects.language as projects_language,
    projects.created_at as projects_created_at
FROM
    `ghtorrent-bq.ght.projects` as projects
INNER JOIN
    `ghtorrent-bq.ght.users` as users
ON
    users.id = projects.owner_id
WHERE
    projects.deleted != true and        -- No deleted projects
    projects.forked_from is NULL and    -- No forked projects
    users.deleted != true and           -- No deleted users
    users.fake != true and              -- No fake users
    users.type = 'USR' and    -- Must be USR (User not the organization)
    projects.language = 'C++';          --Change to desired language
```

Listing 3.1: SQL statements to extract initial data from GhTorrent.

3.1.1.2 Extracting user names from GitHub

GitHub provides an API for extracting a user's name based on the user's login information. However, their APIs have limited access [40], which suspends the session for 1 hour after every 100 fetches. To maintain our research timeline, we developed custom API, *git_users_name* (described in Appendix B.1) with Python code and FastAPI (version 0.115.12) [13] to extract user's name. We

used BeautifulSoup4 (version 4.13.4) [103] to scrape user names from GitHub using the user login information.

We recognize the moral implications of bypassing API rate limits through web scraping. GitHub permits scraping of publicly available information for research, provided that it meets their 'Acceptable Use Policies' [39] and 'Privacy Statement' [41]. As per Clause 7 ("Information Usage Restrictions") of GitHub's Acceptable Use Policies, our usage was academic research and non-intrusive, and did not attempt to collect private or confidential information. All information we collected was public and did not attempt to re-identify users beyond their voluntarily displayed names.

Furthermore, our scraping approach—accessing only the user's public profile page using their login and extracting the displayed name—required fewer server requests than repeated API calls, thereby reducing potential load on GitHub's infrastructure while still complying with their policies.

3.1.1.3 Validating names

To be able to provide contextually relevant gender inference, we wanted to explore names closely resembling real human first names. We used regular expressions to identify and remove names with non-alphabetic characters, numbers, or naming styles that deviated from standard naming practices. We observed that some first names appear, on average, between one and six times, and consist of random letters (e.g., 'mpf', 'cbmk', 'zzzen'). Some of these were not even human names (e.g., 'bigboom', 'peoplecoin', 'blackmagicfine'). We retained names that appeared multiple times (at least 7) in the data and were among known commonly used first names (e.g., 'David' (2,492 instances), 'Daniel' (2,200 instances), 'Michael' (2,097 instances), and 'Alex' (1,678 instances)). After filtering, we retained 224,186 instances that included valid first names and country information.

We acknowledge that this approach introduces biases based on several assumptions. Specifically, we assumed that:

- Human names do not contain numbers or special characters.
- Common naming patterns are a valid proxy for identifying gender using tools such as genderize.io.
- Users who provide non-standard names (e.g., pseudonyms like 'blackmagicfine' or stylized

user names) are likely not sharing their real names, making gender inference unreliable.

These assumptions will exclude individuals who possess less standard or culturally diverse naming conventions, and they will not capture those who actively conceal or misrepresent their online identity. As our study aims to examine gender-based trends in programming style, we prioritized name entries for which we can infer gender fairly reliably through external APIs. While this decision introduces sampling bias, the trade-off was unavoidable with publicly available data.

3.1.1.4 Gender prediction for GitHub data

We used genderize.io to generate gender information for users. Since genderize.io has an API restriction of 100 fetches per day, we opted to purchase their basic subscription ($10/month for $25,000$ fetches). During data cleaning, we removed only those instances for which both the gender prediction and the country information were unavailable. Instances with missing gender but valid country information were retained, as they were later included in the 'other' category for the ternary gender classification. After merging the gender information, the data set contained a total of $219,869$ instances.

3.1.1.5 Extracting programs

We then attempted to collect programs from $219,869$ project instances. Some project URLs were no longer valid, so we ignored those instances. However, some users have multiple programs in their project directory, so we collected all available C++ programs. Although GitHub has open-source repositories with diverse purposes and varying levels of completeness, we confined our analysis to C++ programs in the range of 500 to 1000 characters. This range served two primary purposes. First, it increased the likelihood that extracted samples represented self-contained, meaningful programs—rather than trivial fragments or huge codebases with ambiguous structure. Secondly, the upper size limit ensured compatibility with the maximum token input window of the LLMs used for subsequent modeling tasks [22]. Although we subsequently removed incomplete or poorly formed programs during feature extraction, this early filtering based on size helped to provide greater uniformity in the data set. Capping program length at 500–1000 characters inevitably excluded some longer, more complex programs, introducing a potential bias. However, this constraint

aligned with our research goal of analyzing programming styles in manageable, self-contained programs and providing recommendations for learners and beginners, for whom excessively complex problems would be less relevant. After merging the programs, we collected 626,706 instances.

3.1.2 Codeforces Data Collection

Figure 3.3 provides an overview of the data collection and filtering steps used to construct the Codeforces data set. To represent structured, problem-specific programming contexts, we collected data from Codeforces, a competitive programming platform where users solve well-defined algorithmic challenges under standardized constraints. We used Codeforces API[12] to extract user information, including first name, last name, country, maxRating, and maxRank. We found 147,926 rated users on Codeforces. Rated users are participants in at least one rated contest on Codeforces. A rated contest on Codeforces is a timed programming competition in which each problem carries a predefined score, and participants submit solutions that are automatically judged against hidden test cases. The contest is rated because the results are used to adjust each participant's numerical rating — strong performance increases the rating, while weaker performance decreases it. Codeforces regularly hosts such contests across different divisions based on participant rating, allowing users of similar skill levels to compete under the same conditions.

Using genderize.io, we collected gender information of all users and added it to the user information. Out of 147,926 users, 35,586 were male and 4,489 were female. However, 107,851 users did not have any gender information. This gap arises for two main reasons. First, when registering a Codeforces account, providing a name is optional, and some users may leave this field blank for privacy, avoiding public disclosure on the platform. Second, genderize.io may be unable to predict gender from specific names—such as random strings, arbitrary characters, or non-human names—which some users provide.

Since programming contest problems can vary widely, we set some criteria to select the contest problems for the data set. Codeforces assigns each problem a difficulty rating—ranging from 800 (the easiest) to 3500 (the most difficult)—based on the problem definition and the knowledge required to solve it. Following the observations of previous research [132], we avoided problems at

[12] `https://codeforces.com/api/user.ratedList`.

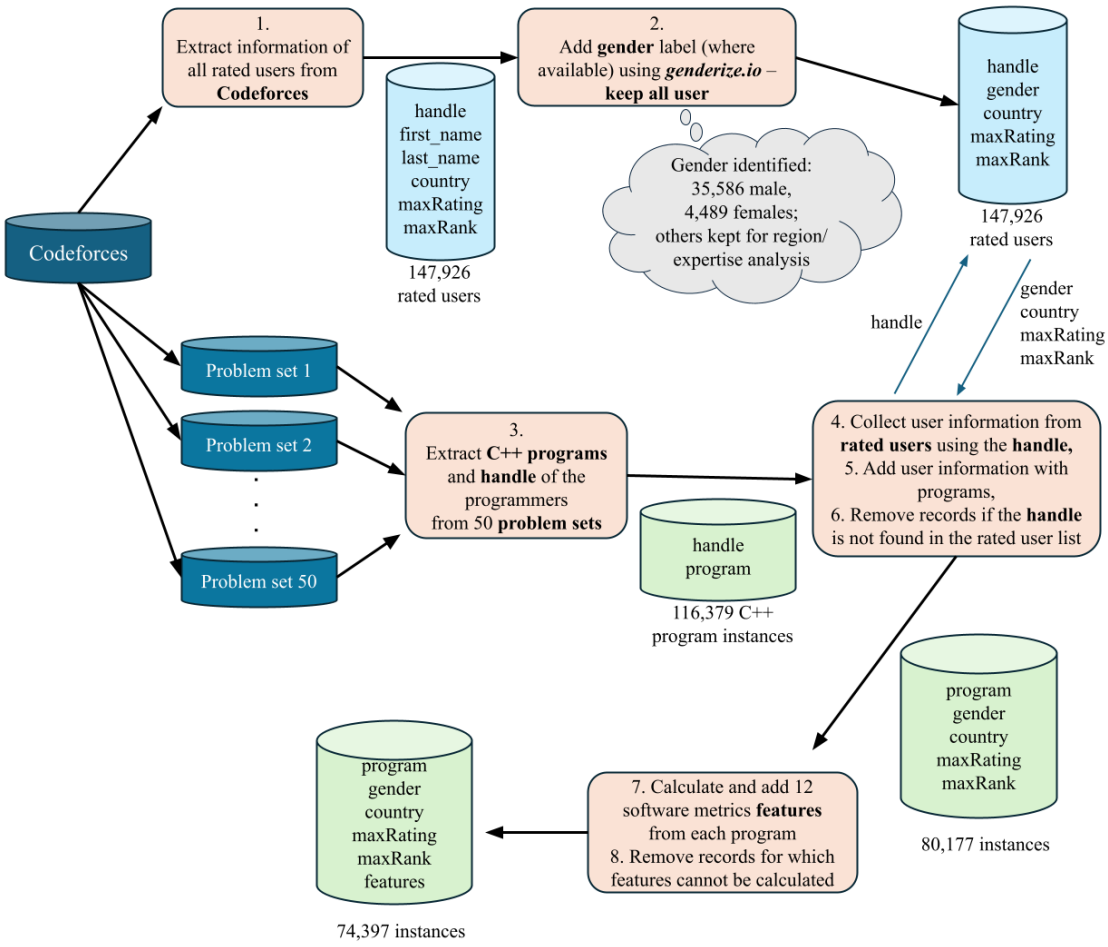

Figure 3.3: Codeforces data collection steps.

the extreme ends of the scale:

- **High-difficulty problems** (2300–3500+) attract very few participants (≈ 100), resulting in too few submissions for meaningful analysis.
- **Low-difficulty problems** (800–1600) attract extensive participation ($6,000$–$100,000$ participants) but are often solved with extremely short code (e.g., one or two lines), limiting feature extraction.
- **Medium-difficulty problems** (1700–2200) typically have $1,500$–$4,500$ participants and produce programs of sufficient length and complexity (e.g., at least two logic statements for cyclomatic complexity calculation).

To keep our Codeforces data set thematically coherent and reduce problem diversity (since GitHub provided our diverse problem set), we fixed the difficulty level at **2100**, near the upper bound of the medium-difficulty problems range and the median (2150) value. This ensured that all selected problems were of similar challenge level and shared a common problem-solving background. Among 393 filtered problem sets, we selected 50 for our data sets based on two criteria: (i) some problem sets had insufficient participant data, and (ii) not all submissions were in C++. As detailed in Appendix A.2, only 42 problem sets included more than $2,000$ C++ submissions from unique users. To broaden the data set, we added six more problem sets at the same 2100 difficulty level with just under $2,000$ submissions, and two additional sets—also at 2100—with slightly fewer submissions, finalizing 50 problem sets for the Codeforces data set.

We scraped $116,379$ programs from the 50 selected problem sets, submitted by $39,776$ unique participants. Earlier, when extracting user information via the Codeforces API, we obtained the complete list of $147,926$ **rated users** on the platform—participants who had competed in at least one rated contest. From this whole list, we identified $35,586$ male and $4,489$ female users using genderize.io, while the remainder had no gender information.

However, not all of these $147,926$ rated users participated in our selected 50 problem sets. Among the $39,776$ participants in these sets, $21,249$ were rated users (present in the earlier API data set), and $18,527$ were unrated users for whom no demographic information was available. Since our user information existed only for rated users, we filtered the programs to include submissions from these $21,249$ rated users, resulting in $80,177$ program instances from the original $116,379$ programs.

3.2 Feature Collection

In this research we worked with three types of features: sociolinguistics features, features relating to a programmer's skill, and software metric features, shown in Table 3.1. During software metric feature calculation, we excluded records where features could not be computed—such as cases with malformed or incomplete source code—reducing the Codeforces data set from $80,177$ to $74,397$ program instances and the GitHub data set from $626,706$ to $583,673$ instances.

Table 3.1: List of features.

Features			
Sociolinguistics	**Programmer's skill**	**Software metrics**	
Gender	Expertise	LOC	Comments
Region		Blank line	Volume
		CC	Difficulty
		Vocabulary	Effort
		Program length	MI
		LLOC	Code quality

3.2.1 Sociolinguistic features

While we acknowledge the sensitivity surrounding the use of sociolinguistic features such as gender and region, their inclusion in our study is deliberate and grounded in sociolinguistic theory. Prior work in both natural and programming languages has shown that stylistic variation is often shaped by social and cultural context [7, 56, 67]. Our aim is not to classify or label individual programmers, but rather to investigate population-level trends in programming styles—anonymized and statistically analyzed—that may reflect broader sociocultural influences on code construction.

Clustering solely by style, although helpful in identifying surface-level groupings, would limit our ability to interpret why particular coding patterns emerge or how they relate to social dynamics. By incorporating sociolinguistic dimensions, we can investigate correlations between code style and factors such as region or gender. This analysis is intended to support a contextualized understanding rather than reinforce stereotypes. For example, understanding how stylistic conventions differ across communities could enable us to design more adaptable recommendation systems or educational approaches that recognize rather than reject diversity in programming styles.

Significantly, all data were anonymized, and no inferences were made regarding the capacity or identity of individuals. Our methodology avoids essentialist conclusions and instead promotes an inclusive perspective that values different programming styles. We believe this approach makes a meaningful contribution to understanding how social context influences code and can inform the design, education, and evaluation of more equitable tools in software engineering.

3.2.1.1 Gender

As gender information is not provided by GitHub or Codeforces, it was inferred from users' first names, as noted in Section 3.1. This inference was performed using the genderize.io API, which predicts binary gender labels along with a confidence score. Predictions with a confidence of at least 0.5 were treated as reliable. Names with lower-confidence predictions or no prediction were retained in the data but were not used for binary (male–female) classification tasks; instead, they were included in the 'other' category when constructing the ternary gender data set.

To make the system more reliable, we performed a name validation procedure prior to gender inference (Section 3.1.1.3). We filtered out names containing special characters and numbers, and removed names that appeared fewer than seven times in the data set. This helped avoid incorrect predictions for entries that were unlikely to represent real personal names (e.g., asdf, qwerty), which could still produce results from the API.

For ternary gender classification, we added a third label, *other*, for situations where genderize.io did not make a prediction (i.e., null). We note that this category does not capture people who explicitly self-identify as non-binary, but rather those whose names could not be identified by the API. Section 4.4.1 discussed some possible reasons for the ternary classification results. We also mentioned why we chose only binary gender-based classification in our next research steps.

Although gender inference was not the primary focus of this study, it was a useful step for analyzing differences in programming styles across demographic lines. We employed automated predictions and frequency-based name filtering to make data more reliable while maintaining a large and diverse data set.

Table 3.2 shows the total number of instances we collected for the gender data set. After calculating the programming features from the Codeforces and GitHub programs, we counted the number of instances for male, female, and other groups of programmers. Within the Codeforces data we had a total of $74,397$ instances where $31,486$ were from male programmers, $3,448$ from female programmers, and $39,463$ were from programmers whose gender was unknown. In the GitHub data we had a total of $583,673$ instances where $477,507$ were from male programmers, $27,176$ were from female programmers, and $78,990$ were from programmers whose gender was unknown.

Table 3.2: Number of instances for each gender label.

Gender	Number of Instances (Codeforces)	Number of Instances (GitHub)
Male	31486	477507
Female	3448	27176
Other	39463	78990
Total	**74397**	**583673**

3.2.1.2 Region

We selected Gross Domestic Product (GDP) as a representative regional indicator because it is a widely recognized measure of a nation's overall economic performance, reflecting the market value of all goods and services produced within a country over a specified period [69]. The world operates on economic drivers, and success—whether in industry, technology, or social development—is often measured through economic growth and profitability [69]. Although geographic boundaries traditionally define regions, the software industry transcends these boundaries due to global communication, trade, and knowledge exchange. However, economic capacity still plays a decisive role in shaping a nation's participation in the global software ecosystem. We hypothesize that countries do not contribute equally to software development: two geographically close nations can have vastly different levels of participation depending on their economic strength. High GDP countries generally show greater participation in programming due to better infrastructure, education systems, and investment in technology, while countries with steadily growing GDP also demonstrate increasing involvement—albeit at a lower scale—than the highest-GDP nations [42, 88]. Our observations confirm that economic performance correlates with programming participation, supporting findings from prior work that innovation-driven economies not only contribute more to global programming but also benefit reciprocally, as advances in software and ICT sectors feed back into sustained GDP growth [37, 127]. Thus, GDP offers a meaningful, quantifiable, and globally comparable lens for analyzing how regional factors influence programming styles and participation.

To examine regional coding practice, we selected ten countries and classified them into two categories: high GDP and steady GDP countries. We used 2023 GDP data from the United Nations data set [88], file "All countries for all years – sorted alphabetically," retrieved in 2024. We also

compared GitHub activity using country-level account creation data from the Innovation Graph [42], focusing on 2023. We identified countries with over one million GitHub accounts created in 2023 and a GDP of over 1 trillion as high GDP countries. These included the United States, China, India, Russia, and Japan. On the other hand, countries with more than $600,000$ GitHub accounts created in 2023, but with a GDP of less than 1 trillion, were classified as steady GDP countries. These included Argentina, Singapore, Vietnam, Bangladesh, and Egypt. Although account creation is only a proxy for developer activity, this metric enabled us to identify areas with stable growth in programming activity, which aligns with studies showing that expansion in ICT and software development sectors is often associated with broader economic growth [37, 127].

We performed this analysis to create well-defined country groups for our experiments, allowing us to investigate whether regional and economic factors influence programming practices. This grouping step was necessary because, as discussed previously, regional factors such as access to global learning resources and technology infrastructure can shape coding styles and participation levels. Table 3.3 shows the total number of instances we collected for the region data set. For high GDP countries, we had $30,701$ instances from Codeforces and $186,426$ from GitHub. For the steady GDP countries, we had $6,274$ instances from Codeforces and $8,474$ instances from GitHub. After grouping countries by GDP, we observed the same trend—instances from the high GDP group were substantially higher than those from the steady GDP group in both data sets, indicating that our findings are consistent with prior research [42, 88]. This supports what earlier studies have reported: participation in programming activity is often correlated with GDP.

This outcome supports our earlier hypothesis that regional and economic factors—particularly GDP—can be linked to differences in programming participation, reinforcing the role of sociolinguistic context in shaping coding practices.

3.2.2 Programmer's skill

We also investigated whether programmers exhibited differences in their coding styles based on their programming skills. Previous work [114] has shown how various software metric features can reflect differences in programmers' skill levels. Measuring skill directly is challenging without access to a person's specialty or certification records. In Codeforces, however, each programmer is

Table 3.3: Number of instances for each region label.

Region	Country	GDP (trillion)	Number of instances (Codeforces)	Number of instances (GitHub)
High GDP	U.S.	28.78	2009	100139
	China	18.53	14382	46976
	India	3.94	9939	16949
	Russia	2.06	3240	9757
	Japan	4.11	1131	12605
	Total		**30701**	**186426**
Steady rise in GDP	Argentina	0.6	230	1935
	Singapore	0.53	366	1583
	Vietnam	0.47	2171	931
	Bangladesh	0.46	2344	3067
	Egypt	0.35	1163	958
	Total		**6274**	**8474**

assigned a numerical rating that is updated after every rated contest, based on their performance relative to other participants. This rating determines the programmer's rank category (e.g., Specialist, Expert, Candidate Master), and both the rating and rank change over time as new contests are held. Codeforces also records the highest rank and rating a programmer has ever achieved—maxRank and maxRating, which are historical peaks rather than separate concurrent measures. We used these maximum values in our study because the performance of a programmer in one contest can differ based on various factors such as problem set difficulty, time constraints, or even external distractions [66, 104]. In contrast, the maximum over time represents their highest exhibited skill level ever. The distribution of these ranks and their corresponding ratings is summarized in Table 3.4.

For a ternary expertise data set, we labelled grandmaster ranked programmers as 'expert' with a rating range from 2400 to 3000+, master ranked programmers as 'intermediate' with a rating range from 1600 to 2399, and other primary ranked programmers as 'beginner' with rating range from 0 to 1599.

We defined two classes for the binary expertise data set: 'newbie' and 'skilled'. The 'newbie' class includes instances where the maxRank was newbie, pupil, specialist, or expert. In contrast, programmers with master and grandmaster maxRanks were classified as 'skilled'.

We chose both ternary and binary classifications for our experiments as we aimed to see how programming style evolves with expertise, and skill acquisition tends to occur incrementally over time rather than in a single leap from a beginner to an expert. Ternary classification allowed us to record this *intermediate* stage, providing a more subtle impression of potential stylistic evolution. However, classification often becomes harder as the number of target classes increases, which can lead to lower performance in ternary setups versus binary [29], and our results reflected this trend. One reason is that the *intermediate* group can be heterogeneous, where some programmers may still display beginner-like styles and others resemble experts, making it harder for the model to learn consistent patterns. To address this, we also used a binary classification, collapsing the expertise levels into two broader categories to improve classification performance.

As shown in Tables 3.4 and 3.5, we had a total of 74,397 instances for the expertise data sets. This number reflects the final Codeforces data set after removing instances with incomplete or malformed code during software metrics feature calculation. Since GitHub does not have any rating or ranking, we used only Codeforces data to prepare an expertise data set.

Table 3.4: Number of instances for each ternary expertise class from Codeforces data set.

Ternary Expertise (Range of rating)	**maxRank (Codeforces feature)**	**Number of Instances (Codeforces)**	**Total Number of Instances per Expertise Class**
Beginner (0∼1599)	Newbie	2160	9392
	Pupil	1971	
	Apprentice	0	
	Specialist	5261	
Intermediate (1600∼2399)	Expert	19492	56223
	Candidate Master	16332	
	Master	16482	
	International Master	3917	
Expert (2400∼3000+)	Grandmaster	4634	8782
	International Grandmaster	3209	
	Legendary Grandmaster	939	
Total Number of instances			**74397**

Table 3.5: Number of instances for each binary expertise class from Codeforces data set.

Binary Expertise (Range of rating)	**maxRank (Codeforces feature)**	**Number of Instances (Codeforces)**	**Total Number of Instances per Expertise Class**
Newbie (0∼1799)	Newbie	2160	28884
	Pupil	1971	
	Apprentice	0	
	Specialist	5261	
	Expert	19492	
Skilled (1800∼3000+)	Candidate Master	16332	45513
	Master	16482	
	International Master	3917	
	Grandmaster	4634	
	International Grandmaster	3209	
	Legendary Grandmaster	939	
Total # of instances			**74397**

3.2.3 Software metrics

Software metrics can reflect variations in the programming styles of different groups of programmer [5, 101, 114, 132]. In this research we evaluated program quality using five software metrics: LOC, CC, comments, Halstead's volume, and difficulty. We used LLOC and the number of blank lines to calculate the number of comments. Additionally, we measured program length, vocabulary, and effort as these metrics are also influenced by programming style [132]. We calculated MI to integrate it into our proposed code quality equation. A detailed description of these features is given in Section 2.2.

We wrote a Python program to calculate the software metrics. We used the Lizard library (version 1.17.31)[13] to calculate CC and LLOC. The number of comments was derived by analyzing the values of LOC, blank lines, and LLOC. We counted the number of operators and operands to determine the Halstead metrics: vocabulary, program length, volume, difficulty, and effort.

[13] `https://pypi.org/project/lizard/`.

3.3 Data Representation

During each step of the data processing, we stored all the information in CSV (Comma Separated Values) files so we could later access them using APIs. While training and testing the machine learning models, we used two data types: numeric and string. Sample data from both GitHub and Codeforces are shown in Table 3.6. While applying the traditional ML model (Random Forest) to classify different programmer groups, we used software metric features in numeric form as the input. In contrast, we used LLMs to classify different programmer groups and transform non-compliant code into compliant code, with the program as input. We stored programs as text, which is a string data type.

Table 3.6: Sample instances from GitHub and Codeforces.

Features	**Sample GitHub Instances**		**Sample Codeforces Instance**
program	Listing D.1	Listing D.2	Listing D.3
LOC	37	34	49
Blank_Line	8	3	15
CC	6	5	5
Vocabulary	18	27	34
Program_Length	63.36	102.71	140.34
LLOC	18	28	27
Comments	11	3	7
Volume	175.14	285.29	595.23
Difficulty	2.7	12.35	25.07
Effort	472.87	3524.21	14923.35
MI	53.62	51.10	46.05
Quality	0.5	0.43	0.38

3.3.1 Standardized numeric input values for the Random Forest model

The Random Forest model requires numeric feature vectors as input and nominal values as output for classification. In this work, each instance was represented as a vector of software metrics: LOC, blank line, CC, vocabulary, program length, LLOC, comments, volume, difficulty, effort, MI,

and code quality. For example, a GitHub sample (Listing D.1) from Table 3.6 can be represented as: [37, 8, 6, 18, 63.36, 18, 11, 175.14, 2.7, 472.87, 53.62, 0.5].

Before we passed these values into the model, we standardized all of the features with *sklearn*'s *StandardScaler* module, which standardizes the data so that it has unit variance and zero mean:

$$\text{Standardized Value} = \frac{X_i - \mu}{\sigma} \tag{3.1}$$

where X_i is the original value of a feature,

μ is the mean of the feature in training or testing sets, and

σ is the standard deviation of the feature in the training or testing sets.

The formula to calculate the mean is:

$$\mu = \frac{X_1 + X_2 + ... + X_n}{n} \tag{3.2}$$

where n is the number of instances.

The standard deviation for each feature is computed as:

$$\sigma = \sqrt{\frac{(X_1 - \mu)^2 + (X_2 - \mu)^2 + \cdots + (X_n - \mu)^2}{n}} \tag{3.3}$$

3.3.1.1 Illustrative two-instance example

To demonstrate the computation process step-by-step, we take a simplified example with two sample programs (Listing D.1 and D.2) from Table 3.6, thus $n = 2$.

The mean for each feature is:

$\mu_{LOC} = \frac{37+34}{2} = 35.5$,

$\mu_{Blank\_Line} = \frac{8+3}{2} = 5.5$,

$\mu_{CC} = \frac{6+5}{2} = 5.5$,

$\mu_{Vocabulary} = \frac{18+27}{2} = 22.5$,

$\mu_{Program\_Length} = \frac{63.36+102.71}{2} = 83.035$,

$\mu_{LLOC} = \frac{18+28}{2} = 23$,

$\mu_{Comments} = \frac{11+3}{2} = 7,$

$\mu_{Volume} = \frac{175.14+285.29}{2} = 230.215,$

$\mu_{Difficulty} = \frac{2.7+12.35}{2} = 7.525,$

$\mu_{Effort} = \frac{472.87+3524.21}{2} = 1998.54,$

$\mu_{MI} = \frac{53.62+51.10}{2} = 52.36,$ and

$\mu_{Quality} = \frac{0.5+0.43}{2} = 0.465.$

Now, we can calculate the standard deviation of each feature using the above formula:

$\sigma_{LOC} = \sqrt{\frac{(37-35.5)^2+(34-35.5)^2}{2}} = 1.5,$

$\sigma_{Blank\_Line} = \sqrt{\frac{(8-5.5)^2+(3-5.5)^2}{2}} = 2.5,$

$\sigma_{CC} = \sqrt{\frac{(6-5.5)^2+(5-5.5)^2}{2}} = 0.5,$

$\sigma_{Vocabulary} = \sqrt{\frac{(18-22.5)^2+(27-22.5)^2}{2}} = 4.5,$

$\sigma_{Program\_Length} = \sqrt{\frac{(63.36-83.035)^2+(102.71-83.035)^2}{2}} \approx 19.675,$

$\sigma_{LLOC} = \sqrt{\frac{(18-23)^2+(28-23)^2}{2}} = 5,$

$\sigma_{Comments} = \sqrt{\frac{(11-7)^2+(3-7)^2}{2}} = 4,$

$\sigma_{Volume} = \sqrt{\frac{(175.14-230.215)^2+(285.29-230.215)^2}{2}} \approx 55.075,$

$\sigma_{Difficulty} = \sqrt{\frac{(2.7-7.525)^2+(12.35-7.525)^2}{2}} \approx 4.825,$

$\sigma_{Effort} = \sqrt{\frac{(472.87-1998.54)^2+(3524.21-1998.54)^2}{2}} \approx 1525.67,$

$\sigma_{MI} = \sqrt{\frac{(53.62-52.36)^2+(51.10-52.36)^2}{2}} \approx 1.26,$ and

$\sigma_{Quality} = \sqrt{\frac{(0.5-0.465)^2+(0.43-0.465)^2}{2}} = 0.035.$

Using Equation 3.1, we then standardize each feature. For example, for LOC in the first GitHub instance (Listing D.1):

$$\frac{37-35.5}{1.5} = 1$$

The resulting standardized vector for the first GitHub instance (Listing D.1) is:

$$[1,\ 1,\ 1,\ -1,\ -1,\ -1,\ 1,\ -1,\ -1,\ -1,\ 1,\ 1]$$

Table 3.7 shows the complete standardized values for both GitHub instances.

3.3.1.2 Real training scenario

In the actual model training process, we use thousands of instances from the data set, not just two. In that case, μ and σ are calculated over the entire training set, producing standardized values distributed across a continuous range (e.g., from approximately -2.5 to $+3.1$) rather than the binary ± 1 seen in the two-instance illustration. The simplified example here is provided solely to demonstrate the computation steps.

Table 3.7: Standardized value of sample instances from GitHub.

Features	Sample GitHub instance 1 (Listing D.1)	Sample GitHub instance 2 (Listing D.2)	μ	σ	Standardized value	
					Instance 1	Instance 2
LOC	37	34	35.5	1.5	1	-1
Blank line	8	3	5.5	2.5	1	-1
CC	6	5	5.5	0.5	1	-1
Vocabulary	18	27	22.5	4.5	-1	1
Program length	63.36	102.71	83.035	19.675	-1	1
LLOC	18	28	23	5	-1	1
Comments	11	3	7	4	1	-1
Volume	175.14	285.29	230.215	55.075	-1	1
Difficulty	2.7	12.35	7.525	4.825	-1	1
Effort	472.87	3524.21	1998.54	1525.67	-1	1
MI	53.62	51.10	52.36	1.26	1	-1
Quality	0.5	0.43	0.465	0.035	1	-1

3.3.2 Vectorizing program (text input) for the LLMs

We used three LLMs: a fine-tuned BERT model to classify programmer groups by analyzing the program context, a fine-tuned T5 model to transform non-compliant code into compliant code, and a fine-tuned CodeBERT model to analyze non-compliant code and generate targeted suggestions for improving its maintainability and structural quality. These three models have different vocabularies, which differ in order and size. When the LLM takes any text input, it tokenizes them and assigns the tokens a corresponding token ID. Token IDs are the order number of each token in the vocabulary list, which is called a dictionary. The sequence of tokens in a string then forms the sequence of token

IDs, called input IDs. After the LLM gets the input IDs, it generates output IDs, which are also a sequence of token IDs that are then converted into corresponding tokens to generate text output.

The BERT model uses BertTokenizer function [31], and the T5 model uses T5Tokenizer function [102] to transform input text into input IDs. We used AutoTokenizer[14] for CodeBERT with model name 'codebert-base'[15].

For example, we represent a sample coding segment (shown in Listing 3.2) into input IDs using BertTokenizer, T5Tokenizer, and AutoTokenizer in Table 3.8. In BERT, [CLS] and [SEP] are the special tokens representing each input sequence's starting and ending. The model transforms the value of [CLS] into 0 or 1 as output to classify the input sequence. In the T5 model, </s> represents the end of the input sequence, and <unk> is the out-of-vocabulary token. Here, '{' and '}' are the out-of-vocabulary tokens in the T5 tokenizer. In the CodeBERT tokenizer, <s> and </s> are the starting and ending tokens of the input sequence. The tokenizer represents whitespace with Ġ and new lines with Ċ, which ensures proper indentation in the coding segment.

```
// Sample function
int fact (int x)
{
 if (x == 1)
  return 1;
 else
  return x * fact(x - 1);
}
```

Listing 3.2: Sample program.

Before fine-tuning the model, we use a tokenizer to convert raw input text into a structured sequence of tokens or token IDs that the model can interpret and learn from during training. As observed in Table 3.8, BERT and T5 tokenizers do not always tokenize code-specific tokens—such as `//`, `==`, or `int`—in a manner aligned with compiler tokenization rules. However, this behaviour does not imply any deficiency in the quality of data. These tokenizers were pre-trained on large-scale corpora primarily consisting of natural language, and we utilized them without modification to process input data prior to model fine-tuning. The examples shown simply illustrate how the existing pretrained tokenizers interpret C++ source code when converting it into tokens for LLM input.

[14] https://huggingface.co/docs/transformers/en/main_classes/tokenizer.
[15] https://huggingface.co/microsoft/codebert-base.

Table 3.8: Data representation of LLMs, derived from the program sample in Listing 3.2.

BertTokenizer		**T5Tokenizer**		**CodeBERT**	
Token ID	**Token**	**Token ID**	**Token**	**Token ID**	**Token**
101	[CLS]	13751	▁//	0	<s>
1013	/	12474	▁Sample	42326	//
1013	/	1681	▁function	16269	ĠSample
7099	sample	16	▁in	5043	Ġfunction
3853	function	17	t	50118	Ċ
20014	int	685	▁fact	2544	int
2755	fact	41	▁(	754	Ġfact
1006	(	77	in	36	Ġ(
20014	int	17	t	2544	int
1060	x	3	▁	3023	Ġx
1007	)	226	x	43	)
1063	{	61	)	50118	Ċ
2065	if	3	▁	45152	{
1006	(	2	<unk>	50118	Ċ
1060	x	3	▁	114	Ġif
1027	=	99	if	36	Ġ(
1027	=	41	▁(	1178	x
1015	1	226	x	45994	Ġ==
1007	)	3274	▁=	112	Ġ1
2709	return	2423	=	43	)
1015	1	8925	▁1)	50118	Ċ
1025	;	1205	▁return	1437	Ġ
2842	else	209	▁1	671	Ġreturn
2709	return	117	;	112	Ġ1
1060	x	1307	▁else	131	;
1008	*	1205	▁return	50118	Ċ
2755	fact	3	▁	1493	Ġelse
1006	(	226	x	50118	Ċ
1060	x	1429	▁*	1437	Ġ
1011	-	685	▁fact	671	Ġreturn
1015	1	599	(	3023	Ġx
1007	)	226	x	1009	Ġ*
1025	;	3	▁	754	Ġfact
1065	}	18	-	1640	(
102	[SEP]	209	▁1	1178	x
		3670	);	111	Ġ-
		3	▁	112	Ġ1
		2	<unk>	4397	);
		1	</s>	50118	Ċ
				24303	}
				2	</s>

Unlike compiler front-ends, LLM tokenizers do not have to comply with a programming language's grammars [22]. They are designed instead to be optimized for statistical learning operations and to convert sequences to subword units in a way that suits modelling context. Thus, irregularities such as splitting known tokens or skipping braces are naturally a part of tokenizer design and not symptoms of corrupted data. Notably, the CodeBERT tokenizer—pretrained on both natural and programming language data—demonstrated markedly improved handling of source code, preserving the structural integrity of code tokens more effectively than its counterparts.

The input IDs using BertTokenizer is [101, 1013, 1013, 7099, 3853, 20014, 2755, 1006, 20014, 1060, 1007, 1063, 2065, 1006, 1060, 1027, 1027, 1015, 1007, 2709, 1015, 1025, 2842, 2709, 1060, 1008, 2755, 1006, 1060, 1011, 1015, 1007, 1025, 1065, 102], using T5Tokenizer is [13751, 12474, 1681, 16, 17, 685, 41, 77, 17, 3, 226, 61, 3, 2, 3, 99, 41, 226, 3274, 2423, 8925, 1205, 209, 117, 1307, 1205, 3, 226, 1429, 685, 599, 226, 3, 18, 209, 3670, 3, 2, 1], and using CodeBertTokenizer is [0, 42326, 16269, 5043, 50118, 2544, 754, 36, 2544, 3023, 43, 50118, 45152, 50118, 114, 36, 1178, 45994, 112, 43, 50118, 1437, 671, 112, 131, 50118, 1493, 50118, 1437, 671, 3023, 1009, 754, 1640, 1178, 111, 112, 4397, 50118, 24303, 2].

3.4 Data Sets

We prepared our data sets[16] to perform three tasks: classifying programs on different programmer groups, transforming non-compliant code into compliant code and recommending suggestions to improve maintainability and structural quality of non-compliant code samples. After extracting data from GitHub and Codeforces, we calculated and incorporated the software metric features into the data sets to prepare initial data sets. Then, for classification, we prepared the balanced data sets using gender, region, and expertise classes from both GitHub and Codeforces. We prepared a data set for the code transformation task using Codeforces submissions, where each non-compliant code sample was paired with a corresponding compliant version. To support our code recommendation model, we derived another data set from the same Codeforces repository, where each non-compliant code sample was compared against its compliant counterpart to extract feature deviations and train the model to generate targeted improvement suggestions.

[16]The prepared data sets have been made publicly available at `https://doi.org/10.5683/SP3/RINCO1`.

3.4.1 GitHub Data Sets

Following the approach discussed in Section 3.1.1, we prepared an initial data set from the GitHub repository. We then incorporated software metric features into the data set following the approach discussed in Section 3.2.

3.4.1.1 Initial Data Set

In the initial data set from GitHub, we prepared 583,673 instances with the following information: program, gender, country code, LOC, blank line, CC, vocabulary, program length, LLOC, comments, volume, difficulty, effort, MI, and quality.

3.4.1.2 Balanced Data Sets

We prepared three balanced data sets from GitHub: one for ternary gender classification, one for binary gender classification, and one for binary region classification. Balancing data over class labels is important to ensure that the training does not bias the model towards one class. For example, observing Table 3.2, there are 477,507 instances from male programmers, 27,176 from female programmers, and 78,990 instances from other programmers. To prepare a balanced data set for ternary labels of the gender data set, instances from all the labels (male, female, and other) must be the same size, usually the size of the smallest class, in this case 27,176.

3.4.1.2.1 Balanced Ternary Gender (GitHub) In the balanced gender data set, we had three class labels: 'male', 'female', and 'other'. As shown in Table 3.2, the number of instances of 'male' and 'other' programmers were higher than 'female' programmers, so we under-sampled [23] instances of 'male' and 'other' programmers to create a balanced data set.

As discussed in Appendix C.1, for the instances labelled as 'male' and 'other', we selected an equal distribution across region and quality classes, ensuring that the number of instances did not exceed that of 'female' programmers. We included all 27,176 instances of 'female' programmers. This resulted in a total of 81,528 instances where each class label – 'male', 'female' and 'other' – equally had 27,176 instances for the balanced gender data set from GitHub.

We prepared both ternary and binary gender data sets to address different analytical require-

ments. The ternary set (‘male’, ‘female’, ‘other’) preserves the full diversity of the data, where it includes an ‘other’ category for instances where gender could not be identified from the user’s first name or where no usable information was provided. While ‘other’ class captures all unidentified gender labels, it may contain programming styles similar to those of either male or female programmers. The binary set (‘male’, ‘female’) contains only instances where genderize.io could confidently assign a gender label, enabling more focused comparisons. However, classification often becomes harder as the number of target classes increases, which can lead to lower performance in ternary setups versus binary [29], and our results reflected this trend.

3.4.1.2.2 Balanced Binary Gender (GitHub)

To classify the gender of programmers into two primary categories – ‘male’ and ‘female’ – we prepared a balanced binary gender data set. We removed the instances labelled as ‘other’ to create a data set with an equal number of male (27,176) and female (27,176) instances, totalling 54,352 instances.

3.4.1.2.3 Balanced Binary Region (GitHub)

From Table 3.3, we can see that in the region data set there are more instances of ‘high GDP’ countries than ‘steady GDP’ countries. We incorporated all instances from the ‘steady GDP’ countries and applied the under-sampling approach on ‘high GDP’ countries as discussed in Appendix C.2. This resulted in a balanced region data set of 8,474 instances in both ‘high GDP’ and ‘steady GDP’ class labels, totalling 16,948.

3.4.2 Codeforces Data sets

Following the approach discussed in Section 3.1.2, we prepared an initial data set from Codeforces, then we added software metric features into the data set.

3.4.2.1 Initial Data Set

In the initial data set, we prepared 74,397 instances with the following information: program, gender, country, maxRating, maxRank, LOC, blank line, CC, vocabulary, program length, LLOC, comments, volume, difficulty, effort, MI, and quality.

3.4.2.2 Balanced Data Sets

We prepared five balanced data sets from Codeforces: ternary and binary gender classification sets, a binary region classification set, and ternary and binary expertise classification sets.

3.4.2.2.1 Balanced Ternary Gender (Codeforces) The balanced gender data set has three class labels: 'male', 'female', and 'other'. As shown in Table 3.2, the number of instances for male and other programmers is significantly higher than those for female programmers. Therefore, we set specific criteria to balance the number of instances for male and other as described in Appendix C.3.

Since the number of programs written by females was fewer than those written by males, we included all of the instances from female programmers. Using the same under-sampling approach as for the GitHub data, for the instances labelled as 'male' and 'other' we selected an equal distribution across region, expertise, and quality classes, ensuring that the number of instances did not exceed that of female programmers. Following this approach, we chose an equal number of $3,448$ instances for each class label ('male', 'female' and 'other'), which resulted in a total of $10,344$ instances for the balanced gender data set.

3.4.2.2.2 Balanced Binary Gender (Codeforces) To classify the gender of programmers into two primary categories 'male' and 'female', we prepared a balanced binary gender data set. We removed the instances labelled 'other' to create a data set with an equal number of male ($3,448$) and female ($3,448$) instances, resulting in a total of $6,896$ instances.

3.4.2.2.3 Balanced Binary Region (Codeforces) As shown in Table 3.3, for the region data set, we observed that there were significantly more instances from 'high GDP' countries than the 'steady GDP' countries. Therefore, we selected all of the instances from the 'steady GDP' countries and applied the under-sampling approach to the 'high GDP' countries as discussed in Appendix C.4. This resulted in a total of $12,548$ instances for the balanced region data set, where $6,274$ instances were from 'high GDP' and $6,274$ instances were from 'steady GDP' regions.

3.4.2.2.4 Balanced Ternary Expertise (Codeforces) Table 3.4 shows that there are fewer instances with the class labels 'Expert' and 'Beginner' than with the 'Intermediate'. The 'Beginner'

class has almost the same number of instances as the 'Expert' label. Within the 'Beginner' class, the maxRank distribution is: 'Newbie' (2,160 instances), 'Pupil' (1,971 instances), 'Apprentice' (0 instance), and 'Specialist' (5,261 instances). Since 'Specialist' accounts for the largest share of the 'Beginner' class, we first removed 610 'Specialist' instances to bring the 'Beginner' and 'Expert' classes to the same size (8,782 instances each). As described in Appendix C.5, we then applied the under-sampling approach on the 'Intermediate' class label which resulted in 26,346 instances for the balanced expertise data set, where each class label ('Beginner', 'Intermediate' and 'Expert') had 8,782 instances.

3.4.2.2.5 Balanced Binary Expertise (Codeforces) There were 28,884 instances in the 'Newbie' class label, while the 'Skilled' class label had 45,513 instances as shown in Table 3.5. We included all 28,884 instances from the 'Newbie' class label and applied the under-sampling approach on the 'Skilled' class label as discussed in Appendix C.6 to create a balanced data set. This gave us a balanced binary expertise data set of 28,884 instances in both 'Newbie' and 'Skilled' class labels, totalling 57,768 instances.

3.4.3 Data set for code transformation

In the code transformation task, we fine-tuned a ML model by providing {non-compliant code, compliant code} pairs of data as input and reference programs. The goal was to train a model to transform a non-compliant code sample into its corresponding compliant code sample.

Codeforces contains unique problem sets, each containing submissions from different programmers. In Codeforces, programmers may solve a common problem differently, but the test cases and expected goals are fixed. Therefore, we generated {non-compliant code, compliant code} pairs from each problem set within Codeforces. Because each project in GitHub is unique there is little opportunity for creating {non-compliant code, compliant code} pairs for ML purposes using GitHub data.

We collected the programs from 50 different Codeforces problem sets and calculated the probability value to measure their code quality as described in Section 2.2.1.5. We labelled a code sample as non-compliant code if the probability value was between 0.0 and 0.44 and compliant code if the

probability ranged from 0.45 to 1.0. Specifically, from each problem set, we chose two compliant samples with the highest code quality score and paired them with non-compliant samples from the same problem set, ensuring a clear quality gap between the paired code samples. Because the compliant samples were both above the 0.45 threshold and selected from the very top of the quality score range, their scores were substantially higher than those of the paired non-compliant samples, making the distinction between the two categories more pronounced. As shown in Table 3.9, among 74,397 programs, 59,534 instances were labeled as non-compliant code and 14,863 instances as compliant code.

Table 3.9: Number of instances for each code quality class.

Quality	**Range [0∼1]**	**Number of instances (Codeforces)**
Non-compliant	0.0 to 0.44	59534
Compliant	0.45 to 1.0	14863
Total		**74397**

After collecting the programs, we found that the number of intermediate programmers was higher than those of both beginner and expert programmers. The 'compliant' label was still assigned solely based on the predicted quality score from our equation (Section 2.2.1.5). However, when selecting a representative compliant sample from each problem set for pairing with non-compliant samples, we chose compliant code written by intermediate programmers. This decision was motivated by our findings that programming style varies with programmer expertise, and that the intermediate style lies between beginner and expert groups, making it more broadly understandable. Beginners can progress to the intermediate style by following standard programming practices, while experts can also follow it with ease. Because the number of female programmers in our data set was significantly lower than that of male or other genders, gender was not considered in this pairing step. For regional variation, we selected separate compliant samples for high GDP and steady GDP countries, as both groups had sufficient participation in the data set.

From each of the 50 problem sets, we selected two compliant programs to serve as *target programs*—the versions that the model should learn to transform non-compliant code into. One target program came from a high GDP country, and the other from a steady GDP country. This resulted

in a total of 100 compliant code instances. To ensure the transformed code would reflect a style accessible to the broadest audience, each target program was chosen from the intermediate-level programmer group. Although our ternary classification results indicated that this group can be heterogeneous—with some programmers displaying beginner-like styles and others resembling experts—we considered this diversity beneficial for producing transformed code that a wider range of programmers could understand. During pairing, a non-compliant code sample from a high GDP country was matched with the high GDP target program and similarly a sample from a steady GDP country was matched with the steady GDP target program. For non-compliant code from countries outside these two groups, we paired them with the high GDP target program. This choice was based on our regional analysis, which showed that high GDP countries had greater participation and higher overall contribution to global programming activity, making their style a practical default when no direct regional match was available. Figures 3.4 and 3.5 illustrate our approach for selecting compliant target programs and pairing them with non-compliant samples.

From 50 different problem sets, we had $59,534$ instances of non-compliant code that could be paired with their corresponding compliant code. However, this could result in model overfitting with a data set where there are only 100 possible target programs. Overfitting occurs when a model achieves nearly zero error on the training set but performs poorly when applied to new, unseen data [126]. In addition, in a $\{$non-compliant code, compliant code$\}$ pair from the same problem set, these programs might follow different approaches to solve the same problem, as two different persons solved them. To address these two issues, we calculated the similarity value between the non-compliant and compliant code of each $\{$non-compliant code, compliant code$\}$ pair using cosine similarity. We chose only pairs with similarity values greater than or equal to 0.6, resulting in $5,106$ instances of $\{$non-compliant code, compliant code$\}$ pairs. To avoid the model overfitting issue, we added $\{$compliant code, compliant code$\}$ pairs where we set the input code and the target code with the same compliant code whose 'quality' value was greater than 0.52, which gave us $5,132$ instances of $\{$compliant code, compliant code$\}$ pairs. We merged both $\{$non-compliant code, compliant code$\}$ and $\{$compliant code, compliant code$\}$ pairs so that the model can use $5132 + 100 = 5232$ possible target programs for fine-tuning, instead of only 100 programs. Following this approach, our data set contained $10,238$ pairs of instances.

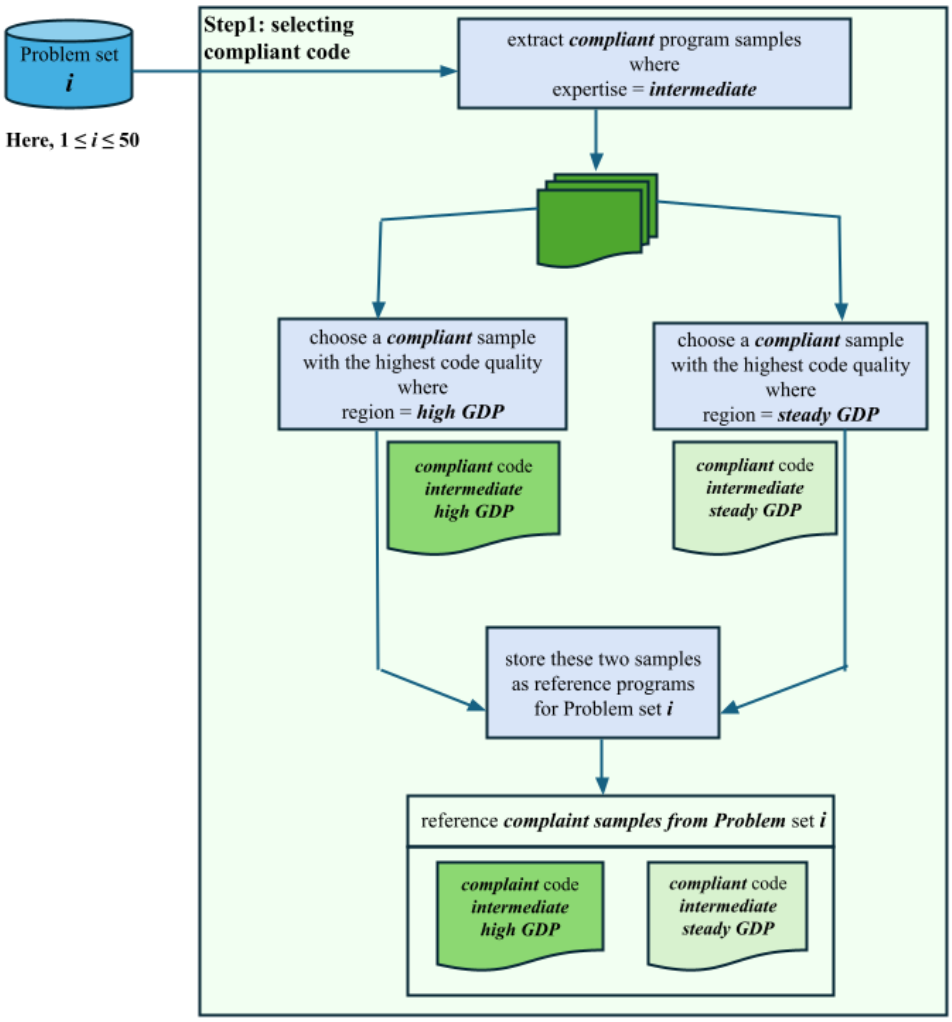

Figure 3.4: Choosing compliant code samples for Problem set i, $1 \leq i \leq 50$.

As we prepared this data set to fine-tune an LLM, we split it into training, validation, and testing sets. To achieve an even distribution for each of the 50 problem sets, we split the data in each set—allocating 80% to the training set, 14% to the validation set, and 6% to the test set. This ensured that each subset included representative samples from every problem set. This resulted in $8,227$ instances for training, $1,472$ for validation, and 539 instances for testing the model. The model is hence tested on new data to avoid data leakage and overfitting.

3.4.4 Data set for code recommendation

While both the code transformation and the recommendation tasks are aimed at improving code quality, they have different purposes and outputs. For the transformation task, the aim is to generate

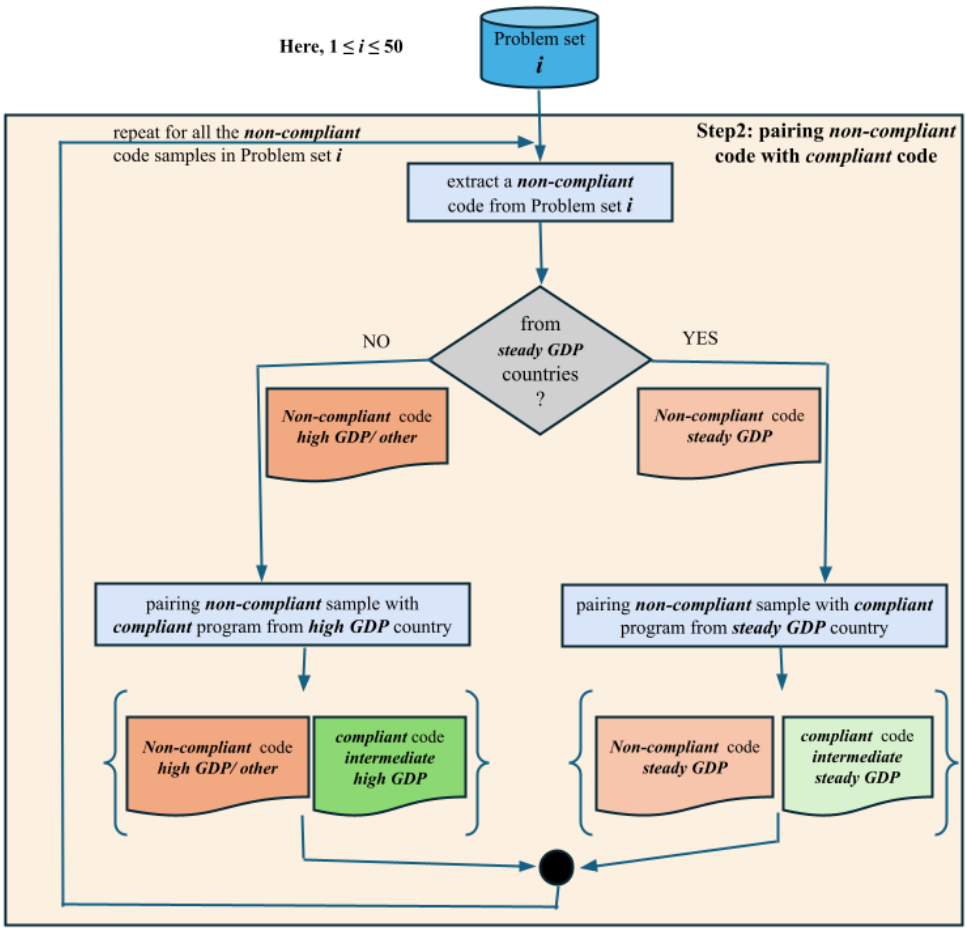

Figure 3.5: Pairing non-compliant and compliant code approach for Problem set i, $1 \leq i \leq 50$.

maintainable code samples from given non-compliant samples. In the recommendation task, the model does not generate programs. Instead, it compares software metrics between non-compliant and compliant samples and generates targeted, metric-based suggestions for improvement. This approach avoids the challenges of generating compilable code while still providing actionable guidance for enhancing code quality.

To best train our recommendation model, we needed a data set that not only captured software metrics variation between non-compliant and compliant code samples but also provided diverse coverage of code quality levels. The following section outlines how we preprocessed, balanced, and split the data set used for model training.

3.4.4.1 Constructing Metric-Based Label Vectors

We collected the programs from 50 different Codeforces problem sets and again calculated the probability value to measure their code quality. As described in Section 2.2.1.5, instead of using a threshold of 0.45, which we chose for the transformation data set, we chose 0.5 for the code recommendation data set. We labelled a code sample as non-compliant code if the probability value was between 0.0 and 0.49 or compliant code if the probability ranged from 0.5 to 1.0. As shown in Table 3.10, of the 74,397 programs, 55,799 instances were labeled as non-compliant code and 18,598 instances as compliant code.

Table 3.10: Number of instances for each code quality class.

Quality	**Range [0∼1]**	**Number of instances (Codeforces)**
Non-compliant	0.0 to 0.49	55799
Compliant	0.5 to 1.0	18598
Total		**74397**

For each problem set, we selected the compliant code sample with the highest code quality value and paired it with all associated non-compliant code samples from the same problem set. These {non-compliant, compliant} pairs enabled us to compare structurally equivalent code and identify key areas of deviation. Each pair allowed us to contrast the measurements of equivalent programs and generate 5-bit binary label vectors indicating where the non-compliant code diverges negatively from the compliant standard. This design enabled the model to learn how to recommend targeted improvements based on observed deviations in metrics.

Preparation involved three steps of significance as shown in Figure 3.6:

- **Metric extraction:** We computed the five key software metrics—Halstead Volume (HV), Lines of Code (LOC), Cyclomatic Complexity (CC), Comments, and Difficulty—for both the non-compliant and compliant code samples.
- **Binary label generation:** For each metric, we compared the value from the non-compliant code with its compliant counterpart. If the non-compliant value was higher, we set the corresponding bit in the label vector to 1; otherwise, 0.

- **Final data set format:** This process yielded a data set composed of pairs in the form: {Non-compliant code sample, [HV, LOC, CC, Comments, Difficulty] binary labels}

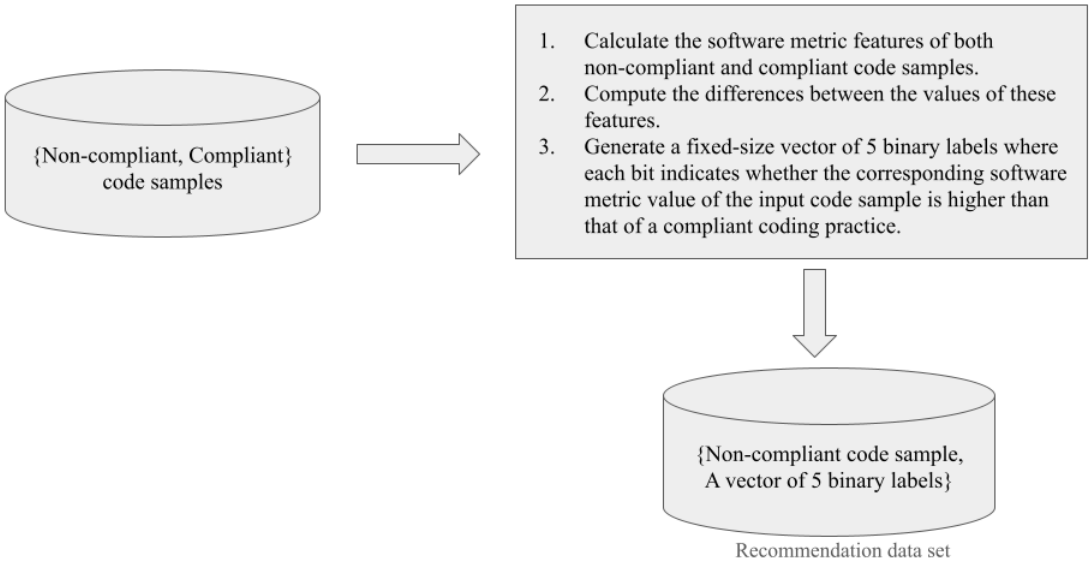

Figure 3.6: Recommendation data set preparation.

These labels allowed the model to learn which metrics typically deviate between non-compliant and compliant code and to make recommendations accordingly.

For example, we consider a pair of code samples—compliant (Listing 2.1) and non-compliant (Listing 2.2)—as introduced in Section 2.2.1.6. In the first step, we compute five software metric features—Halstead volume (HV), lines of code (LOC), cyclomatic complexity (CC), comment, and difficulty—following the methodology described in Section 2.2.1. The computed values are summarized in Table 3.11. Next, we compare the metric feature values between the compliant and non-compliant samples. *HV*, *LOC*, *CC*, and *difficulty* are all higher in the non-compliant code and thus receive a binary label of 1. The *comment* metric is lower in the non-compliant sample and thus receives a label of 0. From this comparison, we construct the following labelled instance for the recommendation data set:

{non-compliant code (Listing 2.2), [1, 1, 1, 0, 1]}.

Table 3.11: Constructing the vector of binary labels for the recommendation data set.

	HV	LOC	CC	Comment	Difficulty
Compliant (Listing 2.1)	60.23	12	2	1	5.2
Non-compliant (Listing 2.2)	100.32	19	4	0	12
Binary labels	1	1	1	0	1

3.4.4.2 Balancing Non-Compliant Code Samples by Quality Distribution

To ensure that the model could generalize over a large enough set of levels of code quality, we looked at the distribution of non-compliant code samples as a function of the quality score computed using our code quality equation from Section 2.2.1.5. By definition, any sample with less than 0.5 in the quality score is non-compliant. We split the non-compliant samples into five bins based on their quality scores as shown in Table 3.12.

Table 3.12: Quality distribution of non-compliant code samples.

Quality Range	Number of Samples
$0.0 \leq x < 0.1$	1086
$0.1 \leq x < 0.2$	3904
$0.2 \leq x < 0.3$	22286
$0.3 \leq x < 0.4$	25828
$0.4 \leq x < 0.5$	2695

To maintain balance and prevent the model from being skewed towards any range, we chose a sample size of $2,695$ from each bin. The bin with the lowest quality range (0.0–0.1) contained only $1,086$ samples, and hence we used all of them.

Thus, the final data set contained a total of (4 bins $\times$ $2,695$) + $1,086$ = $11,866$ samples. This sampling strategy provided the wide range of quality degradation levels that the data set captured while still maintaining balance among the categories.

3.4.4.3 Train-Test-Validation Split

We divided our data set into three portions: 70% for training ($8,306$ samples), 15% for validation ($1,780$ samples), and 15% for testing ($1,780$ samples) to reduce the risk of overfitting. We used

the train_test_split() function of the sklearn.model_selection module with random_state = 42, which produces randomized but repeatable splits, thereby avoiding sampling bias. For example, with 100 samples of data and *test_size* of 0.2, train_test_split() function would provide us with 80 samples for training and 20 samples for testing. This division enabled us to perform hyperparameter tuning on the validation set and evaluate the model's generalization capacity on an unseen test set.

Chapter 4

Experiments and Results

This chapter describes how we performed the experiments and what results we achieved. It also discusses those results.

4.1 Overview of Experiments

In the first phase of this research we prepared eight balanced data sets for classifying computer programs (based on gender, region, and expertise) from GitHub and Codeforces, as discussed in Section 3.4. Using the Random Forest algorithm with software metrics as features, we trained four machine-learning models for gender classification, two for region classification and two for expertise classification. This resulted in a total of eight experiments in Phase 1, as shown in Table 4.1. Our goal was to determine whether software metrics could be used by machine-learning algorithms (such as Random Forest) to identify the gender, region, and expertise of a programmer.

We selected the Random Forest (RF) algorithm for the metrics-based classification task because of its stable performance and interpretability in software engineering research [71, 100]. Compared to single decision trees, RF aggregates numerous trees by bagging, which reduces variance and overfitting, and thus is particularly suitable for high-dimensional data sets [17]. Moreover, in our earlier work [2], RF performed better than all other baseline machine-learning algorithms such as Bayes Net (BN), Sequential Minimal Optimization (SMO), Classification Via Regression (CVR), and Decision Table (DT) for similar tasks. These results confirm RF as an effective method for program classification based on software measures. RF has also been widely used as a baseline model for empirical software engineering problems like defect prediction and maintainability evaluation [71, 100], which reinforces its suitability as the foundation for our Phase 1 experiments.

The second phase moved to the use of Large Language Models (LLMs) called Bidirectional

Table 4.1: Phase 1 – Experiments and data sets for traditional machine learning approach.

Data set	Number of instances	Experiment No.
Balanced ternary gender (Codeforces)	10344	1
Balanced ternary gender (GitHub)	81528	2
Balanced binary gender (Codeforces)	6896	3
Balanced binary gender (GitHub)	54352	4
Balanced binary region (Codeforces)	12548	5
Balanced binary region (GitHub)	16948	6
Balanced ternary expertise (Codeforces)	26346	7
Balanced binary expertise (Codeforces)	57768	8

Encoder Representations from Transformers (BERT) [31] and compared the classification results. We fine-tuned BERT to determine whether the tokens in a source code show any context-based relationships among them and how these relationships help identify the sociolinguistic variables of gender, region, and expertise for each programmer. We developed seven independent models by fine-tuning BERT, including three for gender, three for region, and one for expertise. This gave us a total of seven experiments in phase 2, as shown in Table 4.2.

Table 4.2: Phase 2 – Experiments and data sets for LLM approach.

Data set	Number of instances	Experiment No.
Balanced binary gender (Codeforces)	6896	9, 11
Balanced binary gender (GitHub)	54352	10, 11
Balanced binary region (Codeforces)	12548	12, 14
Balanced binary region (GitHub)	16948	13, 14
Balanced binary expertise (Codeforces)	57768	15

In the third phase, we fine-tuned the Text-to-Text Transfer Transformer (T5) [102] model to transform non-compliant code into compliant code, as shown in Table 4.3.

Phase 4 consisted of fine-tuning CodeBERT [33] model to recommend suggestions for improving maintainability and structural quality of non-compliant code samples Table 4.4.

Table 4.3: Phase 3 – Experiment and data set for code transformation.

Data set	Number of instances	Experiment No.
Code transformation (Codeforces)	10238	16

Table 4.4: Phase 4 – Experiment and data set for code recommendation.

Data set	Number of instances	Experiment No.
Code recommendation (Codeforces)	11866	17

4.2 Programming Environment

We used a combination of tools and environments to perform the code classification, transformation and recommendation tasks.

4.2.1 Visual Studio Code (VS Code)

Visual Studio Code (VS Code, version 1.96)[17] is an integrated development environment developed by Microsoft© for developers. We used VS Code as our editor and wrote Python programs for various coding tasks, including data collection, data pre-processing (cleaning), training, and testing using the Random Forest model.

4.2.2 Google Colab Pro

Google Colab Pro[18] is a paid, hosted Jupyter Notebook[19] service that provides a web-based interactive computing platform. We used it to fine-tune the BERT model for classification tasks, the T5 model for the transformation task, and the CodeBERT model for the recommendation task.

[17] https://code.visualstudio.com/.
[18] https://colab.research.google.com/.
[19] https://jupyter.org/.

4.3 Experiment parameters and operational details

For this research we performed the experiments in two different environments. We performed the experiments in Phase 1 using the Python (version 3.13.3) programming language on a DELL-G3-15 laptop with an *Intel* core i5 processor, 8 GB RAM, 1 TB hard disk, and *Windows*© 11. We conducted the experiments in Phase 2 using Google Colab Pro, where the runtime type was Python 3 and the hardware accelerator was T4 GPU. We stored the data sets in *Google Drive*[20], a cloud-based storage service from which we accessed our prepared data sets while performing experiments. Traditional machine-learning algorithms such as Random Forest, can be effectively trained on local CPU machines due to their relatively low computational requirements. In contrast, large language models (LLMs) such as T5 and CodeBERT, require significantly more computational power and are typically trained on GPU machines to handle the extensive parallel processing demands. As a result, we used two distinct environments—-one for training the Random Forest model on a CPU machine and another for training the BERT, T5 and CodeBERT models on a GPU machine—-for our experiments.

4.3.1 Phase 1: Classification using Random Forest and software metrics

We experimented with our prepared balanced data sets to classify based on ternary (male, female, and other) and binary (male and female) labels representing gender. Additionally, we classified the programmers based on their expertise using ternary (beginner, intermediate, and expert) and binary (newbie and skilled) labels. We considered only binary (high GDP and steady GDP) class labels for region classification.

We used Pandas (version 2.3.0), a Python library, to read the CSV (Comma Separated Values) file, and another library, sklearn (version 1.7.0), an open-source machine-learning and data-modelling library, to train and test the model. From Sklearn, we chose the *RandomForestClassifier* model for the training and different scores (accuracy_score, precision_score, recall_score, and f1_score) for evaluating the trained model. To evaluate the predictive performance of the model on unseen data, we split the data sets into training and testing sets. We chose a test size of 20% to provide a reasonable balance between training and testing data. Additionally, to ensure the results

[20] `http://drive.google.com/`.

are reproducible, we performed the split with a fixed *random_state* value. This ensured that the data was split into training and testing subsets in the same way, without random shuffling, across different executions of a model. We used the twelve software metrics features shown in Table 4.5 for our experiments.

Table 4.5: Software metrics features.

Software metrics features				
1	LOC		7	Comments
2	Blank line		8	Volume
3	LLOC		9	Difficulty
4	Vocabulary		10	Effort
5	Program length		11	Code quality
6	Cyclomatic complexity		12	Maintainability index

4.3.1.1 Gender classification using Random Forest and software metrics

For gender classification using Random Forest and software metrics, we conducted the following experiments:

- Experiment 1: Ternary (male, female, and other) gender classification with Codeforces data,
- Experiment 2: Ternary (male, female, and other) gender classification with GitHub data,
- Experiment 3: Binary (male and female) gender classification with Codeforces data, and
- Experiment 4: Binary (male and female) gender classification with GitHub data

The results are given in Table 4.6 and discussed in Section 4.4.1.

4.3.1.2 Region classification using Random Forest and software metrics

For region classification using Random Forest and software metrics, we conducted the following experiments:

- Experiment 5: Binary (high GDP and steady GDP) region classification with Codeforces data, and

- Experiment 6: Binary (high GDP and steady GDP) region classification with GitHub data.

The results are given in Table 4.8 and discussed in Section 4.4.2.

4.3.1.3 Expertise classification using Random Forest and software metrics

For expertise classification using Random Forest and software metrics, we conducted the following experiments:

- Experiment 7: Ternary (beginner, intermediate, and expert) expertise classification with the Codeforces data, and
- Experiment 8: Binary (newbie and skilled) expertise classification with Codeforces data.

The results are given in Table 4.10 and discussed in Section 4.4.3.

4.3.2 Phase 2: Classification by fine-tuning BERT model with source code

In phase 2 of our work, we fine-tuned the BERT model to classify the programmer's sociolinguistic features based on binary (male and female) gender, binary (high GDP and steady GDP) region, and binary (newbie and skilled) expertise. Initially, we fine-tuned the BERT model with Codeforces data sets. Since source code samples from Codeforces are online contest programs to facilitate competitive programming practice, we additionally used GitHub public repositories to increase the variety of programs. However, unlike Codeforces, GitHub has no rating or ranking, so we used only Codeforces to fine-tune the BERT model for expertise classification.

4.3.2.1 Configuration of the BERT Model for fine-tuning

Training a large language model from scratch requires a large data set and high computational power [1]. Fine-tuning adapts a specific task by training a pre-trained (previously trained with one or more extensive data sets) machine-learning model with a smaller, domain-specific data set, which saves computing power [55]. After selecting a pre-trained model as the base model, we follow three basic steps for fine-tuning: process data using the base model's tokenizer, set appropriate hyperparameters to train the model and use different metrics to evaluate the model's performance after training.

We used the BERT model (*bert-base-uncased* from the Hugging Face[21], with 110 *M* parameters) as our language model and fine-tuned it to classify the gender, region, and expertise of programmers based on their source code. We selected balanced data sets prepared for binary classification and set *num_labels* to 2. The input parameter *num_labels* refers to how many labels the model will predict for the classification task.

We installed the necessary libraries for the experimental setup, including *Transformers* and *Scikit-learn* (version 1.6.1), where the *Transformers* (version 4.53.2) library facilitated model training. The *Scikit-learn* library enabled us to process our data set using *train_test_split* and *LabelEncoder* and evaluate performance using such metrics as accuracy, precision, recall, and F1-scores.

We used *BertTokenizer* to tokenize the input data and managed variable-length sequences by applying truncation and padding, with a maximum length of 512 tokens. We set the batch size to 32 for both training and evaluation. At the beginning of training, large gradients could cause instability. We set a warmup step count of 500, which means the learning rate increased for the first 500 steps to ensure stable updates. This process prevented instability at the start of training by gradually increasing the learning rate. After the warmup phase, the learning rate followed a decay schedule to prevent overfitting by adding a penalty to the loss function. We used a weight decay value of 0.01. We set the evaluation strategy to 'epoch' and evaluated the model performance at the end of each training epoch. Finally, we configured the *Trainer* class to encapsulate these settings, integrating the training and evaluation data sets to create the training process.

4.3.2.2 Gender classification by fine-tuning BERT with source code

In the gender classification we conducted the following experiments:

- Experiment 9: Fine-tuning BERT with Codeforces data (GenCF),
- Experiment 10: Fine-tuning BERT with GitHub data (GenGH), and
- Experiment 11: Fine-tuning BERT with GitHub and Codeforces data (GenGH-CF).

Results are given in Table 4.7 and discussed in Section 4.4.1.

[21] `https://huggingface.co/`.

4.3.2.3 Region classification by fine-tuning BERT with source code

In the region classification we conducted the following experiments:

- Experiment 12: Fine tuning BERT with Codeforces data (RegCF),
- Experiment 13: Fine tuning BERT with GitHub data (RegGH), and
- Experiment 14: Fine tuning BERT with GitHub and Codeforces data (RegGH-CF).

Results are given in Table 4.9 and discussed in Section 4.4.2.

4.3.2.4 Expertise classification by fine-tuning BERT with source code

In the expertise classification, by fine-tuning the BERT model with source code, we conducted the following experiment:

- Experiment 15: Fine-tuning BERT with Codeforces data (ExpCF).

Table 4.11 presents the result, which is discussed in Section 4.4.3.

4.3.3 Phase 3: Fine-tuning T5 model to transform non-compliant code into compliant code

Throughout our research we used various software metrics, including code quality, to analyze code to determine whether it was well-written or needs improvement. In phase 3 of our work we aimed to take a code sample that was classified as non-compliant and generate an improved solution for the programmer. We prepared a code transformation data set, trained a language model, and suggested an improved version of the code sample so the programmer could decide how to focus on enhancing their coding skills.

We chose a transformer-based encoder-decoder model, T5, to transform non-compliant code into compliant code. T5 was pre-trained on a large corpus for various NLP tasks. We fine-tuned the model so that the encoder could understand the input (non-compliant code) and the decoder could generate the output (compliant code).

4.3.3.1 Fine-Tuning Configuration of the T5 Model

We used the pre-trained T5 tokenizer (T5Tokenizer) from the Hugging Face transformers library, specifically the T5-base model. This tokenizer is designed for the T5 model's text-to-text framework, where both input and output sequences are tokenized using the same vocabulary and preprocessing.

The data set consisted of pairs of non-compliant and compliant code samples. To prepare the data for training, we defined a function that tokenizes both the inputs (non-compliant code) and outputs (compliant code). The function truncates and pads sequences to a length of 512 tokens, the maximum sequence length the T5 model can handle. The labels (i.e., compliant code) are tokenized using the *as_target_tokenizer()* context manager to ensure the target code is processed appropriately.

We loaded the data sets (train, validation, and test) from Pandas's *DataFrames* and converted them into *Dataset* objects using the *from_pandas()* method. We then tokenized these data sets using the *map()* function, applying the *tokenize_function* in a batched manner.

For the fine-tuning, we used the *T5ForConditionalGeneration* model, initialized with the pre-trained weights from the *t5-base*. The model was transferred to the GPU machine for training.

We managed the training using the *Trainer* class from the *Transformers* library of Hugging Face. The configuration for training was defined in the *TrainingArguments* object, which specifies several important hyperparameters, including:

- Learning rate: We chose a learning rate of $2e-5$ as a reasonable value for fine-tuning.
- Batch size: We set the batch size to 4 for both training and evaluation.
- Epochs: We trained the model for 3 epochs to ensure adequate fine-tuning.
- Warmup steps: We included 500 warmup steps to gradually increase the learning rate, helping the model stabilize early in training.
- Weight decay: We applied a weight decay of 0.01 to regularize the model and prevent overfitting.

We instantiated the *Trainer* with the following components:

- The model: The T5 model to be fine-tuned.

- The training arguments: The configuration specifies how the training will proceed.
- The train and evaluation data sets: The tokenized training and validation data sets.

4.3.3.2 Code transformation by fine-tuning T5 model

For the non-compliant code to compliant transformation task, we conducted the following experiment:

- Experiment 16: Fine-tuning T5 with Codeforces data (Code transformation model).

4.3.4 Phase 4: Code recommendation by fine-tuning CodeBERT model

In contrast to the classification and code transformation experiments presented in this chapter, Phase 4 focuses on a different objective: generating targeted, metric-based recommendations for improving non-compliant C++ code, rather than predicting programmer attributes or generating compliant code. This phase builds directly on our analyses of software metrics, using them as the basis for actionable improvement suggestions.

Section 5.1 outlines the design, training, and evaluation of our recommendation model, which was fine-tuned from the CodeBERT architecture to identify software metrics deviations in non-compliant code and recommend changes that would bring the code closer to a compliant standard. The model was trained on the recommendation data set described in Section 3.4.4, which was constructed specifically for this purpose and differs in structure from the data sets used in phases 1-3.

The experiment conducted in this phase is as follows:

- Experiment 17: Fine-tuning CodeBERT with Codeforces data (Recommendation model).

Section 5.1.5 presents the results of this experiment, highlighting the recommendation model's ability to provide improvement guidance without generating compilable code. By presenting this phase in a separate chapter, we aim to clearly distinguish it from the classification and transformation experiments, both in methodology and in research objectives.

4.4 Results

During this research, we experimented with classifications of different programmer groups in several phases. We also showed a baseline result for the code transformation task to transform a non-compliant code sample into compliant code.

4.4.1 Gender classification results

Experiments $1-4$ and $9-11$ focused on gender classification. The first phase $(1-4)$ used traditional machine learning (Random Forest), while the second phase $(9-11)$ used the LLM (BERT).

4.4.1.1 Experiment 1

In Experiment 1, the Random Forest model showed an accuracy of 40.99%, as indicated in Table 4.6. The Random Forest model's precision, recall, and F1-score are 40.9%, 41%, and 40.8%, respectively. In this experiment, we used the Codeforces ternary gender data set, and trained the model to classify three class labels: male, female, and other.

As discussed in Section 3.4.1.2.1, the 'other' class may contain programming styles similar to those of either male or female programmers, which can make ternary classification more challenging and lead to lower performance compared to binary classification. Other work has suggested that the binary classification approach performs better than the ternary or multi-class classification models [16]. Therefore, in experiments 3 and 4 we trained the model to perform gender classification with only binary class labels (male and female).

Table 4.6: Gender classification using Random Forest and software metrics.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 1	40.99	40.9	41	40.8
Experiment 2	50.94	50.9	50.9	50.9
Experiment 3	61.3	61.29	61.3	61.27
Experiment 4	66.36	66.36	66.36	66.36

4.4.1.2 Experiment 2

Since Codeforces consists of competitive programs, we used GitHub to increase the variety of programs for ternary (male, female, and other) gender classification in Experiment 2. As shown in Table 4.6, the Random Forest model showed an accuracy of 50.94% in Experiment 2, slightly better than in Experiment 1. This level of accuracy is relatively low for a three-class classification task, where a random guess would yield approximately 33.33%. One possible explanation is that the 'other' group lacks a well-defined gender category, and its members may have programming styles similar to those of either male or female programmers. This overlap can reduce the model's ability to distinguish between classes. Thus for the following experiments we reduced the classification to binary class labels: male and female.

4.4.1.3 Experiment 3

Experiment 3 uses the binary gender labels (male and female) and the balanced Codeforces data. Table 4.6 shows that in Experiment 3, the Random Forest model with 12 software metrics features achieved an accuracy of 61.3% with precision, recall, and F1-score values of 61.29%, 61.3%, and 61.27% respectively. This is an improvement of approximately 20% over the ternary classification approach. To evaluate the binary classification performance using data other than the competitive programs, we conducted binary gender classification using GitHub, which provided a much larger sample set for the next experiment.

4.4.1.4 Experiment 4

In Experiment 4, the Random Forest model achieved an accuracy of 66.36% for GitHub data with binary class labels: male and female. Table 4.6 shows that the Random Forest model's precision, recall, and F1-score values are 66.36%, 66.36%, and 66.36%, respectively. Although this accuracy is higher than in the ternary classification results from Experiments 1 and 2, it still indicates limited discriminative power for large-scale gender classification. Given the global nature of software development, programmers often have access to similar resources, programming languages, and libraries, and may adopt comparable solution structures for the same problem. When classification relies solely on software metrics such as LOC, CC, and Halstead measures, these structural

similarities can reduce the visibility of group-specific differences, especially in large and diverse data sets [30].

This finding links directly to our broader research question (Section 1.1): whether programmers—despite potentially using similar logic or problem-solving approaches [15]—still exhibit distinct stylistic signatures in how they structure, comment, and organize their code. For example, two programmers might solve the same sorting problem using identical library functions, but differ in variable naming conventions, comment placement, and overall code organization. These stylistic patterns are complex for metrics-based models to detect, but can be captured by models that consider the full semantic and contextual flow of the code.

This motivated our hypothesis that context-based analysis using large language models (LLMs) could improve classification performance. Therefore, we performed experiments 9, 10, and 11 using the BERT large language model which was fine-tuned for the specific classification task.

4.4.1.5 Experiment 9

In Experiment 9, we fine-tuned the BERT model using Codeforces data with two gender class labels: male and female. This resulted in an accuracy of 76.88% with precision, recall, and F1-score values of 77%, 76.88%, and 76.88%, respectively. As shown in Table 4.7, this performance is significantly better than the highest performance of the Random Forest model.

Table 4.7: Gender classification by BERT with source code.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 9 (Codeforces)	76.88	77	76.88	76.88
Experiment 10 (GitHub)	**82.2**	**82.2**	**82.2**	**82.2**
Experiment 11 (Codeforces)	76.01	76.28	76.01	75.91
Experiment 11 (GitHub)	76.94	77.1	76.94	76.9

4.4.1.6 Experiment 10

In Experiment 10, we fine-tuned the BERT model using GitHub data with two gender class labels: male and female. This resulted in our best results for gender classification. As seen in Table 4.7, the GenGH model achieved an accuracy of 82.2%. This seems to support our hypothesis that context-based analysis of programs using large-language models may achieve higher performance results. One contributing factor is the substantially larger size of the GitHub data set compared to Codeforces. In general, fine-tuning large language models over a bigger data set gives more thorough coverage of contextual patterns, less overfitting, and improved generalizability to unseen examples [31]. In our case, the increased diversity of programming styles, problem domains, and coding contexts in GitHub samples likely increased the model's ability to capture stylistic signals relevant to gender classification.

4.4.1.7 Experiment 11

Experiments 9 and 10 evaluated the BERT model when trained exclusively on Codeforces and GitHub data, respectively. In these experiments, we did not evaluate the Codeforces-trained model on GitHub data, nor the GitHub-trained model on Codeforces data; however, given the substantial domain differences between structured contest-style problems in Codeforces and the diverse problem types in GitHub, it was reasonable to expect a performance drop if such cross-domain testing were conducted [93]. To address this potential limitation, in Experiment 11 we combined the Codeforces and GitHub training sets to fine-tune a single model (GenGH-CF) to improve its ability to generalize across both domains. Table 4.7 shows that the GenGH-CF model achieved an accuracy of 76.01% (precision: 76.28%, recall: 76.01%, F1-score: 75.91%) on the Codeforces testing set and 76.94% (precision: 77.1%, recall: 76.94%, F1-score: 76.9%) on the GitHub testing set. Interestingly, these values are lower than those in Experiment 10, suggesting that while mixing data sets from different domains improves cross-domain robustness, it can undermine domain-specific features learned from a single source and result in a modest reduction in peak performance. In contrast, the GenGH-CF model offers a more versatile solution by guaranteeing balanced accuracy on both domains and being more consistent when dealing with unseen data from various programming environments.

4.4.2 Region classification results

Observing statistical data from [42, 88], we reasoned that based on GDP, we could categorize countries into two regions (countries having high GDP and steady rise in GPD countries), where programming practices and styles between programmers from those groups of countries may differ. Experiments $5-6$ and $12-14$ focused on classifying binary (high GDP and steady GDP) regions. In the first phase of these experiments (experiments $5-6$), we used a traditional machine-learning method (Random Forest), and we used the large language model (BERT) in the second phase (experiments $12-14$).

4.4.2.1 Experiment 5

In Experiment 5, we classified balanced binary (high and steady GDP) Codeforces data using the Random Forest model. As shown in Table 4.8, the model resulted in an accuracy of 65.9% with precision, recall, and F1-Score values of 66%, 65.9%, and 65.9%, respectively. Next, we performed the classification task on open-source GitHub data and conducted Experiment 6.

Table 4.8: Region classification using Random Forest and software metrics.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 5	65.9	66	65.9	65.9
Experiment 6	76.84	76.9	76.8	76.8

4.4.2.2 Experiment 6

In Experiment 6, the Random Forest model achieved an accuracy of 76.84% for GitHub data. Table 4.8 shows the model's precision, recall, and F1-scores: 76.9%, 76.8%, and 76.8%, respectively. While this represents an improvement over some earlier classification results, it is still not as high as we would like for large-scale region classification. One reason is that programmers from different regions may share similar programming languages, libraries, and problem-solving approaches—especially when working on widely used developer platforms—resulting in structurally similar code. When classification relies solely on software metrics such as LOC, CC, and Halstead measures, these similarities can obscure regional stylistic patterns, particularly in large and diverse

data sets [30].

This observation relates to our broader research question (Section 1.1) of whether programmers—despite potentially using similar logic or problem-solving approaches [15]—still exhibit distinct regional signatures in how they structure, comment, and organize their code. For example, two developers from different regions might implement the same e-commerce checkout workflow using the same framework, yet differ in their choice of default currency formats, date-time conventions, or error-message phrasing to align with regional user expectations. These subtle, region-shaped coding decisions are difficult for metrics-based models to capture, but can be more effectively recognized by models that analyze the complete contextual structure of the code.

Our earlier experiments with gender classification had shown that replacing a metrics-based Random Forest model by a context-based large language model (LLM) such as BERT led to a notable improvement in accuracy (Section 4.4.1). This motivated us to attempt the same approach here, hypothesizing that context-based analysis would improve region classification as well. Therefore, we conducted Experiments 12, 13, and 14 using the BERT large language model, which was fine-tuned for this specific classification task.

4.4.2.3 Experiment 12

In Experiment 12, we fine-tuned the BERT model using Codeforces data with two region class labels: high GDP and steady GDP. The fine-tuned BERT model (RegCF) showed an accuracy of 85.22% with precision, recall, and F1-score values of 85.24%, 85.22%, and 85.22%, respectively. As shown in Table 4.9, this performance is better than the highest performance of the Random Forest model.

4.4.2.4 Experiment 13

We again fine-tuned the BERT model with GitHub data (with two region class labels: high GDP and steady GDP) in Experiment 13 and developed the RegGH model. As seen in Table 4.9, the RegGH model showed the highest performance for region classification with two class labels: high GDP and steady GDP. The model showed an accuracy of 89.44% with the same precision, recall, and F1-score values: 89.44%. As with the gender classification, the RegGH model successfully

Table 4.9: Region classification by BERT with source code.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 12 (Codeforces)	85.22	85.24	85.22	85.22
Experiment 13 (GitHub)	**89.44**	**89.44**	**89.44**	**89.44**
Experiment 14 (Codeforces)	83.9	84.01	83.9	83.89
Experiment 14 (GitHub)	78.79	80.12	78.79	78.47

classified the source code samples using context-based source code analysis with a more extensive data set.

4.4.2.5 Experiment 14

Experiments 12 and 13 evaluated the BERT model for region classification when trained exclusively on Codeforces and GitHub data, respectively. In these experiments, we did not evaluate the Codeforces-trained model on GitHub data, nor the GitHub-trained model on Codeforces data; however, given the substantial domain differences between structured contest-style problems in Codeforces and the diverse problem types in GitHub, it was reasonable to expect a performance drop if such cross-domain testing were conducted [93]. To address this potential limitation, in Experiment 14 we combined the Codeforces and GitHub training sets to fine-tune a single model (RegGH-CF) to improve its ability to generalize across both domains. Table 4.9 shows that the RegGH-CF model achieved an accuracy of 83.9% for the Codeforces testing set and 78.79% for the GitHub testing set. Although these values are lower than those of RegGH in Experiment 13, the decrease is likely due to the dilution of domain-specific features when blending data from different domains, a trade-off that often occurs in multi-domain training. Overall, RegGH-CF remains a more versatile solution, offering balanced performance across both domains and greater reliability when encountering unseen data from varied programming contexts.

4.4.3 Expertise classification results

Experiments 7, 8, and 15 focused on expertise classification. The first phase ($7-8$) used traditional machine learning (Random Forest), while the second phase (15) used the LLM (BERT).

4.4.3.1 Experiment 7

In Experiment 7, the Random Forest model used software metrics and achieved an accuracy of 49.12% for Codeforces data, including three class labels: beginner, intermediate, and expert. As indicated in Table 4.10, the model showed precision, recall, and F1-score values of 49.3%, 49.1%, and 49.2%, respectively. As explained in Chapter 3 (see Table 3.4), the intermediate class included programmers from four maxRank levels (Expert, Candidate Master, Master, and International Master). However, the first two maxRanks are near the beginner levels, while the other two maxRanks are closer to the expert level. This likely contributed to the model achieving only 49.12% accuracy—slightly above the random-chance level for three classes (approximately 33%)—indicating that the Random Forest approach struggled to discriminate effectively between the expertise levels. Therefore, we conducted Experiment 8 to classify binary (newbie and skilled) expertise using Codeforces data.

Table 4.10: Expertise classification using Random Forest and software metrics.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 7	49.12	49.3	49.1	49.2
Experiment 8	58.64	58.6	58.6	58.6

4.4.3.2 Experiment 8

In Experiment 8, we used Codeforces data with two expertise class labels: newbie and skilled. The Random Forest model's performance improved by only 9 percentage points over the three-class setting in Experiment 7, reaching an accuracy of 58.64%. While this was an improvement, the model still showed limited ability to separate the two expertise groups. As observed in our gender and region classification experiments (Sections 4.4.1.5 and 4.4.2.3), metrics-based models such as Random Forest can miss subtle stylistic and semantic patterns that distinguish programmer groups.

In contrast, large language models (LLMs) like BERT can leverage the full contextual and structural information within source code, enabling them to capture expertise-related cues—such as problem-solving strategies, abstraction choices, or code organization—that aggregate software metrics cannot easily represent. Based on this reasoning, we proceeded to fine-tune the BERT model for context-based classification in Experiment 15.

4.4.3.3 Experiment 15

In Experiment 15, we fine-tuned the BERT model using the Codeforces expertise data set with two class labels (newbie and skilled) and developed the ExpCF model. As shown in Table 4.11, the model achieved an accuracy of 74.54% with precision, recall, and F1-score values of 74.55%, 74.54%, and 74.53%, respectively. This suggests that our hypothesis—that context-based analysis using BERT would improve binary expertise classification—was correct, as the model outperformed the metrics-based Random Forest result from Experiment 8. However, the accuracy remained around 74%—well above the random baseline of fifty percent for binary classification, but still leaving a quarter of the instances misclassified—indicating that the task remains challenging.

Table 4.11: Expertise classification by BERT with source code.

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Experiment 15	74.54	74.55	74.54	74.53

As described in Section 3.2.2, we defined programmer expertise using their highest recorded Codeforces rank and rating (maxRank and maxRating), which reflect peak historical performance rather than the skill level demonstrated in any single program instance. While this avoids short-term fluctuations caused by problem set difficulty, time constraints, or external distractions [66, 104], it also means that an expert programmer could occasionally produce code resembling a beginner's style. A beginner could, in rare cases, produce code resembling that of an expert. This label–instance mismatch likely introduced noise into the classification process. Furthermore, programmers may have peak expertise in a different programming language than the one used in the given instance, reducing the visibility of style cues linked to their primary skill. Finally, the wide variation in Codeforces problem domains and required solution strategies can influence code structure and style

independently of programmer expertise, further limiting the model's ability to separate the two groups with higher accuracy.

4.4.4 Phase 3: Code transformation results (Experiment 16)

Following the classification experiments in Phases 1 and 2, which established that context-based large language models (LLMs) could detect differences in programming style, we proceeded in Phase 3 to investigate whether such models could be used to actively improve code style and quality. Our research goal is to facilitate programmers in enhancing the maintainability and reducing the difficulty of their programs—qualities we refer to collectively as code quality. Unlike existing online resources that typically provide complete solutions, our goal was to develop a tool that could take a programmer's non-compliant code and produce a compliant version. This transformation would then allow us to analyze the generated code and recommend targeted improvements.

In Experiment 16, we fine-tuned the T5 model using the Codeforces data set, consisting of paired non-compliant and compliant code samples. The model's performance was evaluated using Rouge-1, Rouge-2, and Rouge-L metrics—each reported in terms of recall, precision, and F1-score—and the BLEU score. Unlike accuracy in classification, which directly reflects the proportion of correct predictions, these metrics measure *textual similarity* between the generated code and the compliant target code. Rouge scores capture the degree of overlap in unigrams (Rouge-1), bigrams (Rouge-2), and the longest common subsequence (Rouge-L). BLEU evaluates the precision of n-gram matches and includes a length penalty to discourage outputs that are much shorter than the target. The definitions and calculation details for these metrics are provided in Section 2.4.8.2. Higher scores indicate greater lexical similarity, but do not necessarily guarantee functional correctness or compilability.

As shown in Table 4.12, the T5 model achieved F1-scores (Rouge-1 and Rouge-L) of approximately 41–42%, with a BLEU score of 0.237. In the context of code generation, these values reflect that the transformed programs shared a measurable amount of lexical overlap with the target compliant code, but also exhibited substantial differences—an expected outcome given that many distinct implementations can satisfy the same specification [78]. However, despite producing outputs with comparable lexical similarity to those reported in prior code-to-code transformation benchmarks, the outputs often failed to compile.

Table 4.12: Performance of Code Transformation model.

		Our model
Rouge-1 (%)	Recall	37.83
	Precision	50.78
	F1-score	42.13
Rouge-2 (%)	Recall	25.07
	Precision	37.79
	F1-score	29.43
Rouge-L (%)	Recall	36.71
	Precision	49.51
	F1-score	41.01
BLEU score [0∼1]		0.2371

This lack of compilable output meant we could not apply software metric–based analysis directly to the transformed code, which in turn motivated the redesign in Phase 4 (Section 5.1) toward a recommendation-based approach, where targeted software metric–based diversity could be achieved without requiring compilable generated code.

4.5 Overview and next Phase

We hypothesize that sociolinguistic variables such as programmers' gender, region, and programming expertise help in identifying their program writing style. Experiments 1–4 and 9–11 demonstrated that gender-related differences can be detected in source code, while Experiments 5, 6, and 12–14 determined stylistic variations associated with programmers' regional backgrounds. Moreover, Experiments 7, 8, and 15 confirmed that expertise levels also leave measurable stylistic imprints in code. These findings validate our first hypothesis.

Previous research found that software metrics helped in identifying different groups of programmers [132]. In our research, the results of Experiments 1–8 demonstrated that metrics such as LOC, CC, HV, comments, difficulty, and MI can be used with traditional models to classify different groups of programmers, validating Hypothesis 3. However, based on the model's performance, these metrics fall short in representing the stylistic differences that more strongly distinguish sociolinguistic groups of programmers. This could be because different groups of programmers may

share the same resources to generate ideas while solving problems. As well, compared to the recent past, many online forums and tools now provide source code solutions for various problem requirements [15].

To address this limitation, Experiments 9–15 focused on context-based models. A comparison of the experimental results from Experiments 1–8 and Experiments 9–15 is discussed in Section 6.2. We observed that context-based LLMs are more effective than metric-based models for identifying different sociolinguistic backgrounds of programmers, supporting Hypothesis 4.

In Section 4.3.3 we further conduct a transformation task by fine-tuning an encoder–decoder LLM in Experiment 16. A detailed discussion including limitations of encoder–decoder models for transforming non-compliant code into compliant code is presented in Section 6.3. The ROUGE and BLEU scores (discussed in Section 4.4.4) demonstrate that the model was able to capture and reproduce such stylistic patterns. However, because the generated outputs were not executable—owing to architectural and tokenization constraints—we could not conclusively verify whether the transformations precisely produced compliant code. Although the model demonstrated the ability to perform stylistic adjustments (which partially supports Hypothesis 5), its practical significance in generating compliant code remains unconfirmed. These challenges highlight the importance of future research on code-aware tokenization and hybrid methods that integrate LLMs with program analysis techniques, as discussed in detail in Section 6.3.

In Section 4.3.4 we introduced our recommendation model as Experiment 17, which fine-tunes CodeBERT to detect code quality issues and generate targeted recommendations. The experiment and results are discussed in Section 5.1.

Based on these experimental results and recognizing their educational potential, we integrated the models into a prototype web-based tool called *Code Insight*, introduced in Chapter 5. While *Code Insight* has not yet been evaluated through user studies, it builds on and extends prior research on intelligent tutoring systems and code feedback platforms such as AutoStyle [24] and OverCode [43], which aim to provide personalized feedback to users. The significance of our tool lies in the combination of classification and recommendation models, guided by sociolinguistic and software quality analysis, which together provide a new approach to support code comprehension and self-improvement. Details of this tool and its development are given in Section 5.2.

Chapter 5

Recommendation Model and Tool

This chapter provides an in-depth look at our recommendation model and the prototype tool designed to help students and learners enhance their programming practices. First, we present a recommendation model that examines code and gives specific suggestions for improvement. Unlike focusing on code generation, our approach enables developers to follow best practices without needing to create code that is both syntactically correct and executable. While syntactic correctness and executability remain important, in educational or learning contexts, the emphasis often shifts toward providing suggestions that improve style and maintainability [131], helping programmers develop stronger habits beyond simply producing working code.

The chapter concludes with a presentation of *Code Insights*, our prototype tool. Using visual aids and concrete examples, we demonstrate how the tool is designed to provide helpful, metric-based feedback and gradually changes non-compliant code into more maintainable and compliant versions. This tool showcases our research vision for utilizing metrics and machine learning to assist developers in writing better, more maintainable code.

5.1 Recommendation Model

In this section, we describe the design, data preparation, training, and testing of a recommendation model that can suggest actionable improvements to non-compliant C++ code based on software quality metrics. This model can help programmers by identifying areas where the style of the code can be improved and suggesting specific recommendations that can improve code quality according to standard software engineering practices.

5.1.1 Model Overview

We fine-tuned an encoder-based large language model, CodeBERT, for multi-label classification to develop our recommendation model. As illustrated in Figure 5.1, the model accepts a non-compliant code sample as input and outputs a vector of five binary labels. Each bit in this vector corresponds to a specific software metric:

- Halstead Volume (HV)
- Lines of Code (LOC)
- Cyclomatic Complexity (CC)
- Comments
- Difficulty

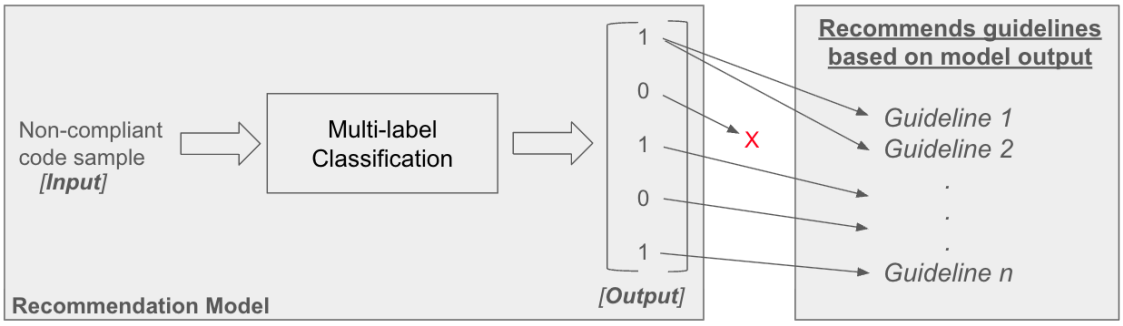

Figure 5.1: Recommendation approach.

A value of 1 for a given metric indicates that the non-compliant code has a higher value than its compliant counterpart for that metric, suggesting a potential area of concern. For example, an output of [1, 0, 1, 0, 1] would indicate that the code has higher HV, CC, and difficulty, while its LOC and comments are lower or equal compared to the compliant coding practice.

For an unseen source code sample we can compute the five software metrics as described in Chapter 2 using the respective formulas. However, determining whether a particular value for a specific metric—such as Halstead Volume (HV), Lines of Code (LOC) or Cyclomatic Complexity (CC)—is appropriately low or unsuitably high is not straightforward. This is because the values of

metrics depend on context, problem complexity, implementation style, and programming style. For example, consider the code sample shown in Listing 2.2, where the computed LOC is 19 (Table 2.1). By itself, this number alone does not reveal whether the code is too long or acceptable. Comparing it with one possible solution to the same problem, presented in Listing 2.1, shows an alternative solution provides the same functionality in only 12 lines. This implies that the first version could be more verbose or less maintainable. However in practice, such comparisons are not always possible as a known compliant counterpart may not be available.

To address this, we trained a model capable of evaluating source code by comparing its metric values against those of corresponding compliant code samples. For instance, the model may predict whether the Halstead Volume of a given code sample exceeds that of a compliant coding practice recommended in the literature. If so, the tool can flag a potential issue such as "Halstead volume (HV) seems high" and provide actionable suggestions like "Eliminating any redundant operators can reduce HV" or "Removing any redundant operands can decrease HV."

5.1.2 Software Metrics and Guideline Derivation

5.1.2.1 Selection of Software Metrics

To drive our recommendations for improving code quality, we selected five key software metrics: Halstead volume (HV), lines of code (LOC), cyclomatic complexity (CC), comments, and difficulty. These metrics represent well-established dimensions of software quality and have wide acceptance in both academic literature and industry best practices [26, 47, 52, 81, 113].

We selected these measures based on three overall considerations:

- **Theoretical Foundation:** Each measure has a direct mapping to the most important factors of code readability, maintainability, or cognitive complexity [26, 47, 52, 81].
- **Empirical Validation:** The relationship of changes in these measures with maintainability and defect proneness has been verified by earlier research [47, 52, 81, 86].
- **Practical Measurability:** It is possible to automatically compute all five metrics from source code using static analysis. So they are feasible for large-scale automatic evaluation [92].

Moreover, incorporating these metrics into a composite code quality equation, we quantitatively assessed whether a code sample is compliant (quality ≥ 0.5) or non-compliant (quality < 0.5). In Chapter 2 we introduced a formula to assess code quality. This formula integrates the Difficulty metric and the Maintainability Index (MI)—a widely adopted, standardized measure in software engineering that is based on Halstead volume (HV), cyclomatic complexity (CC), lines of code (LOC), and comments [26, 47, 81, 52]:

$$\text{QUALITY} = \frac{4}{5} \times \frac{\text{MI}}{100} + \frac{1}{5} \times \frac{1}{\text{Difficulty}} = \frac{1}{125} \times \text{MI} + \frac{1}{5} \times \frac{1}{\text{Difficulty}}$$

Since the composite quality score is calculated directly from these five measures, the same set was utilized while generating recommendations. This synchronization allows consistency between measuring and providing feedback. For example, suppose the overall score is reduced due to a high number of LOC or a high cyclomatic complexity value. In that case, the model can directly mark these metrics as being too high relative to compliant practice. In this way, the model's output not only explains why a particular code sample is assessed as non-compliant but also points to concrete, measurable properties that a programmer can address. This design ensures that recommendations are interpretable and actionable, directly linking quality evaluation to improvement guidance.

5.1.2.2 Recommendation Development

While absolute thresholds (e.g., "LOC must be < 50") might be fine for specific contexts, they will not take into account the nuances of coding practice in a particular problem space. We therefore used a relative comparison rather than using hard thresholds when preparing the recommendation data set. For each non-compliant sample, we provided a comparison to a compliant equivalent that solved a similar problem. Where the non-compliant sample had a greater value for a particular metric, we marked it as potentially problematic. However, the interpretation of the comments metric differs. In this case, a lower number of comments in non-compliant code is considered as a concern, as comments contribute positively to the Maintainability Index and overall code understandability.

We treated comments differently: a value of 1 indicates that the non-compliant code has at least as many or more comments as the compliant example, suggesting adequate documentation. A

value of 0 indicates that the number of comments is lower in the input (non-compliant) code sample compared to compliant practices, suggesting that programmers should improve their documentation.

This relative evaluation strategy offers two advantages:

- It uses realistic, context-aware baselines to measure, as the compliant samples presumably demonstrate good coding practices on similar problems.
- It is in support of metric-specific advice that is empirically guided and coincides with what has proven effective in practice [86].

In order to turn metric deviations into actionable recommendations, we devised guidelines for issuing recommendations following software engineering principles. The guidelines define potential issues that arise when a metric is greater in non-compliant code and offer concrete advice for fixing it.

All of these suggestions are based on the fundamental definitions and intended purposes of the metrics, as described in Section 2.2.1:

- **Halstead Volume (HV):** High HV means large and potentially complicated code structures. Reducing unnecessary operations or operands will decrease this value, and the code becomes more readable and maintainable.
- **Lines of Code (LOC):** Extremely long code may be a sign of bloat or lack of abstraction. Shortening code by removing unused sections improves readability and maintainability.
- **Cyclomatic Complexity (CC):** High CC makes code harder to test and understand as it means more decision points. Decomposing functions and reducing logic significantly reduces this complexity.
- **Comments:** Fewer comments are a sign of a lack of documentation, which would hinder understanding, especially in team environments. Increasing inline and functional comments solves this issue.
- **Difficulty:** This metric reflects the effort to understand or modify code. Simplifying logic, introducing abstractions, or removing duplication can simplify code.

Table 5.1: Guidelines to improve code quality.

Software Metrics	Value is **higher** in **Non-compliant** code than in **compliant** code	**Guidelines**
Halstead Volume (HV)	Yes	**Potential issue:** Halstead volume (HV) seems high **Suggestion:** 1. Eliminating any redundant operators can reduce HV 2. Removing any redundant operands can decrease HV
Lines of Code (LOC)	Yes	**Potential issue:** Lines of code (LOC) seems high **Suggestion:** 1. Eliminating unused functions, variables, or debug code can reduce LOC 2. Minimizing unnecessary blank lines can decrease LOC
Cyclomatic Complexity (CC)	Yes	**Potential issue:** Cyclomatic complexity (CC) seems high **Suggestion:** 1. Splitting large functions into several simple functions can reduce CC 2. Minimizing branching logic can decrease CC
Comments	No	**Potential issue:** Number of comments seems low **Suggestion:** 1. Describe purpose of each function as comments 2. Describe input pattern as comments
Difficulty	Yes	**Potential issue:** Program seems difficult **Suggestion:** 1. Breaking down complex math or logic expressions can reduce program difficulty 2. To reduce program difficulty, use loops or functions to eliminate repetition in tasks or similar code blocks

The complete set of guidelines appears in Table 5.1. Applying these rules, our system not only identifies which measures are suboptimal but also suggests concrete refactoring that developers can use to improve their code's compliance.

5.1.3 Summary of Data Preparation Pipeline

The following is a short overview of the data preparation process for the recommendation model. We presented a detailed description of each step in Chapter 3, Section 3.4.4. Here, we provide a short overview to keep the continuity from data preparation to model training.

1. **Identifying compliant and non-compliant code samples:** Using the code quality equation (Section 2.2.1.5), we labeled code with quality < 0.5 as *non-compliant* and code with quality ≥ 0.5 as *compliant*.

2. **Pairing solutions:** For each Codeforces problem set, the compliant solution with the highest quality score was paired with all corresponding non-compliant solutions. These pairs ensured that stylistic deviations were compared against functionally equivalent programs.

3. **Constructing binary label vectors:** For each $\{$non-compliant, compliant$\}$ pair, we compared five metrics—Halstead volume (HV), lines of code (LOC), cyclomatic complexity (CC), comments, and difficulty. A binary vector was generated such that a value of 1 was assigned if the non-compliant sample had a higher metric value than its compliant counterpart (except for the comments metric, where fewer comments resulted in 1).

4. **Balancing by quality distribution:** After constructing the metric vectors for non-compliant code samples, we stratified them into quality bins and applied uniform under-sampling, ensuring that the final data set maintained balanced representation across all levels of quality degradation.

5. **Train/validation/test split:** The balanced data set, where each instance consisted of a non-compliant code sample paired with its 5-bit label vector, was divided into 70% training, 15% validation, and 15% testing sets using randomized but reproducible splits. The training set was used to fit model parameters, the validation set to tune hyperparameters and monitor overfitting, and the test set to provide an unbiased estimate of final model performance.

These steps are summarized schematically in Figure 5.2.

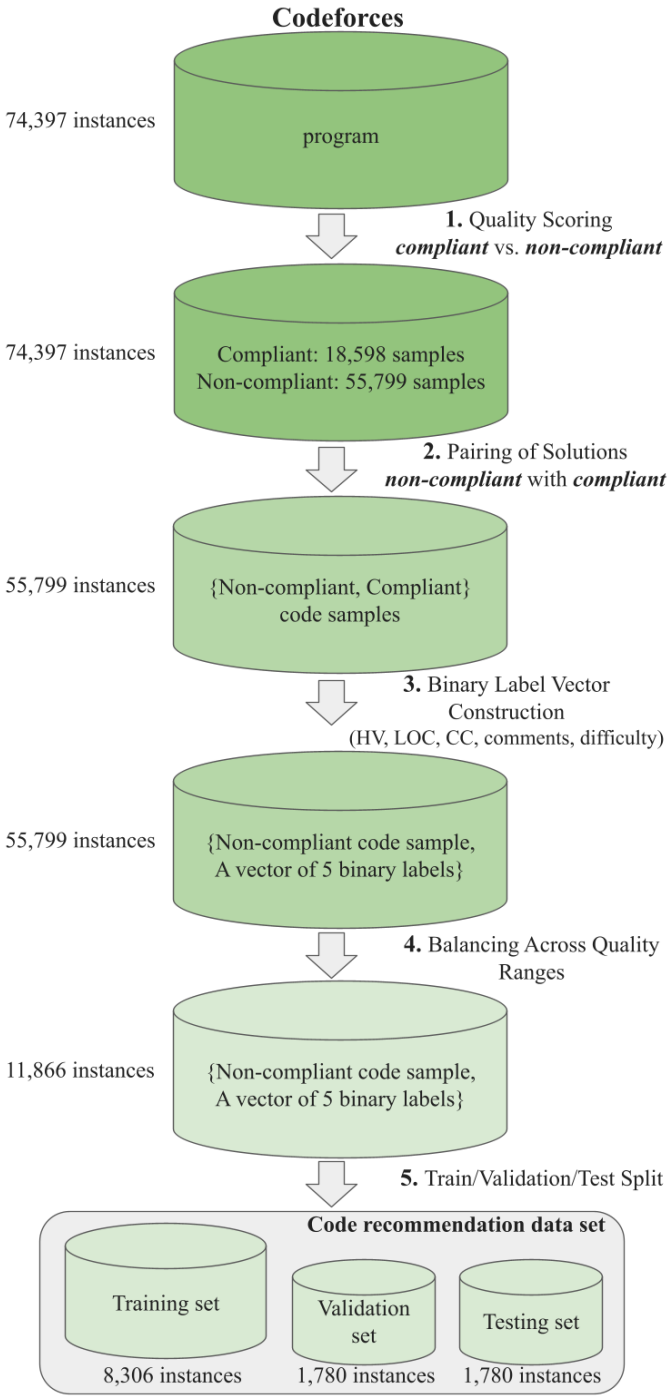

Figure 5.2: Summary of the data preparation pipeline for the recommendation model.

5.1.4 Model Training

To train our recommendation model, we fine-tuned a pre-trained language model—CodeBERT (microsoft/codebert-base)—on our custom data set (discussed in Section 3.4.4) of non-compliant

C++ code samples and their associated metric-based label vectors. Here, non-compliant code refers to programs with a quality score below 0.5—indicating code that lacks maintainability or imposes undue cognitive burden on readers and maintainers—as defined earlier in Sections 2.2.1.5 and 2.2.1.6. CodeBERT is a transformer-based encoder model pre-trained on source code and natural language, making it a strong foundation for learning structural and semantic patterns in code. We defined this task as a multi-label classification problem, where the input to the model is a non-compliant code sample, and the output is a binary vector of five labels, one for each software metric for whether it is a violation of compliant coding standards.

We trained with the Hugging Face[22] Transformers library. The important configuration details are as follows:

- **Training Loss and Stability:** During training, the model optimizes a cross-entropy loss function, which measures how far its predicted token probabilities deviate from the true tokens in the training data [33]. Lower cross-entropy values indicate that the model is learning to represent and predict code patterns more accurately. The term *training stability* refers to how smoothly this loss decreases over time without large spikes or oscillations [77]. Stable training typically indicates that the model is improving well and not diverging or oscillating due to significant updates or poorly configured hyperparameters.

- **Learning Rate** ($2e-5$)**:** We use the learning rate to control how fast the model updates its weights while training. A too high value may result in the model overshooting optimal solutions, while a very low value will make training slow or even remain stuck in suboptimal minima.

  - We employed the conservative learning rate of $2e-5$ (0.00002), which has been widely recommended as a starting point for fine-tuning transformer models like CodeBERT.

  - This low learning rate allows stable and incremental adaptation of the pre-trained model to our domain-specific task without overfitting and catastrophic forgetting of the pre-trained knowledge.

[22] `https://huggingface.co/`.

- **Batch Size** (16)**:** Using batch size, we determine the number of samples processed in parallel before a weight update during the training process.
  - A smaller batch size allows for better generalization and helps prevent overfitting, especially on moderate-sized data sets like ours (11,866 samples).
  - It also ensures that GPU memory constraints are respected, which is often necessary when working with large transformer models.
- **Epochs** (4)**:** An epoch is one complete pass through the entire training data set.
  - We trained for four epochs, which was sufficient to allow the model to converge without overfitting.
  - This value was determined empirically—after observing the training loss and validation performance over time, we found that the model continued to improve for the first few epochs and then began to level off beyond that.
  - Training longer than four epochs offered diminishing returns while increasing computational cost.
- **Warmup Steps** (500)**:** Warmup steps are used to increase the learning rate at the beginning of training gradually. This helps stabilize the model early on and avoid irregular updates when weights are still random or unadapted.
  - We used 500 warmup steps, which is typical for fine-tuning transformer models on small to mid-sized data sets. This allows the optimizer to "warm up" and find a stable direction before applying full-scale updates.
- **Weight Decay** (0.01)**:** Weight decay is a regularization technique that avoids overfitting by penalizing large weights.
  - Having the decay parameter at 0.01 invokes moderate regularization, and it encourages the model to capture simpler and more generalizable patterns.
  - It is particularly effective in multi-label classification tasks, where label imbalance and noisy signals can lead to overfitting specific patterns in the training data.

All code samples were tokenized using the CodeBERT tokenizer, with a max_length of 512 tokens. This limit is typical for transformer models and ensures that most code samples are fully captured without truncation, preserving both syntax and structure. During training, the model's loss steadily decreased, indicating consistent learning and convergence. By step 2050, the training loss dropped to 0.1478, down from an initial value of 0.6944.

5.1.5 Evaluation and Performance

After training, we evaluated our multi-label classification model on a held-out test set consisting of $1,780$ non-compliant code samples. The goal of this evaluation was to assess how accurately the model could predict which software metrics in a given code sample deviate from compliant coding practices based on learned patterns during training. Since this is a multi-label classification task, we assessed the model's performance separately for each of the five binary labels: Halstead Volume (HV), Lines of Code (LOC), Cyclomatic Complexity (CC), Comments, and Difficulty. To evaluate classification results, we analyzed both per-label confusion matrices and standard classification metrics such as precision, recall, and F1-score.

5.1.5.1 Confusion Matrices

The confusion matrix provides insight into how well the model identifies deviations in each software metric between non-compliant and compliant code. In this setup, every code sample is non-compliant overall, but its associated 5-bit label vector specifies which individual metrics deviate from the paired compliant program. Here, a **True Positive (TP)** indicates that the model correctly recognized a metric in the non-compliant code that exceeded its paired compliant program (e.g., higher HV, LOC, CC, or Difficulty). A **True Negative (TN)** indicates that the model correctly recognized when no such deviation existed. A **False Positive (FP)** is when the model signals a deviation even though the metric is aligned with the compliant program, and a **False Negative (FN)** is when the model fails to detect a deviation that exists. For the *comment* metric, the interpretation differs: since a positive label indicates adequate commenting, a TP corresponds to correctly identifying that a non-compliant sample nonetheless has sufficient comments, whereas an FN indicates that adequate commenting was misclassified as insufficient. Conversely, TN and FP represent cor-

rect and incorrect identification of under-commented code, respectively. Table 5.2 summarizes the meaning of the positive and negative classes for each metric.

Table 5.2: Definition of positive/negative classes and their interpretation in confusion matrices.

Metric	Positive class (1)	Negative class (0)
Halstead Volume (HV)	Non-compliant code has higher HV than its paired compliant program	HV aligned with its paired compliant program
Lines of Code (LOC)	Non-compliant code has more LOC than its paired compliant program	LOC aligned with its paired compliant program
Cyclomatic Complexity (CC)	Non-compliant code has higher CC than its paired compliant program	CC aligned with its paired compliant program
Comments	Non-compliant code has adequate comments, aligned with its paired compliant program	Non-compliant code has fewer comments than its paired compliant program
Difficulty	Non-compliant code has higher Difficulty than its paired compliant program	Difficulty aligned with its paired compliant program

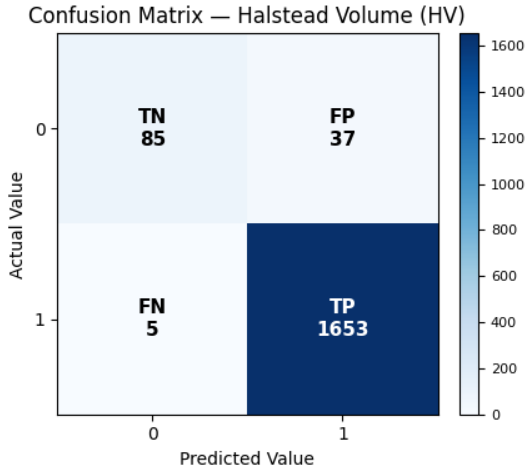

Figure 5.3: Confusion Matrix for Halstead Volume (HV) prediction.

Figures 5.3, 5.4, 5.5, 5.6, and 5.7 show the confusion matrices for the five evaluated metrics. In the case of HV, the model correctly identified 1653 positive instances and only a few errors, which

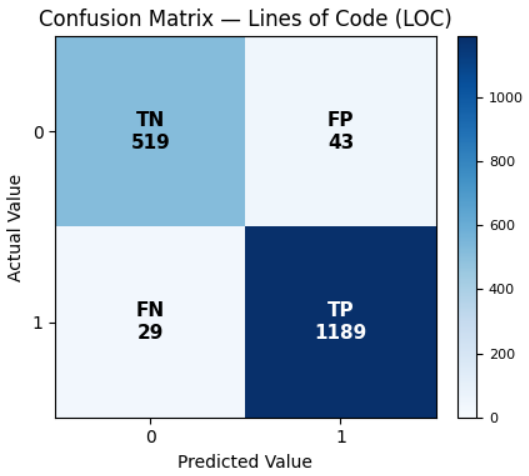

Figure 5.4: Confusion Matrix for Lines of Code (LOC) prediction.

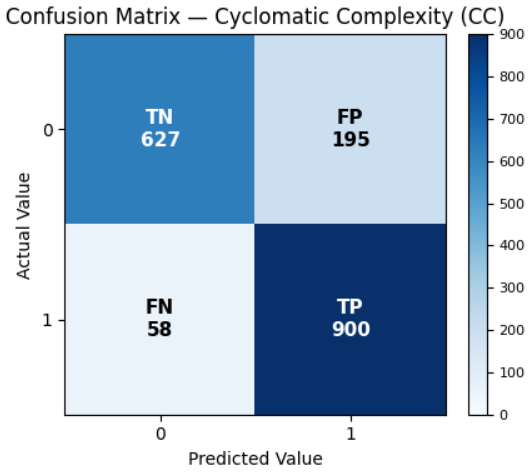

Figure 5.5: Confusion Matrix for Cyclomatic Complexity (CC) prediction.

points to precision and recall that are nearly ideal. Difficulty showed a similar outcome, with 1574 of 1583 positive cases identified correctly and nine missed. LOC had a few misclassifications in either direction. For CC, the matrix shows 195 false positives and 58 false negatives, indicating that mislabeling was more frequent than for other metrics. For comments, the matrix records 80 false negatives and 120 false positives. While most adequately commented samples were classified correctly, misclassifications tended to occur in programs whose commenting was near the dividing line between adequate and inadequate.

Taken together, the confusion matrices show variation in classification performance across the

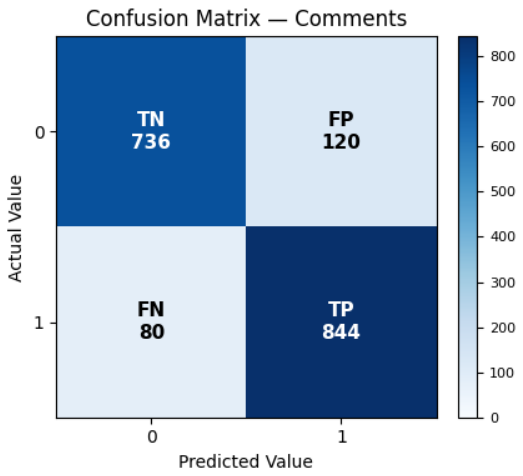

Figure 5.6: Confusion Matrix for Comments prediction.

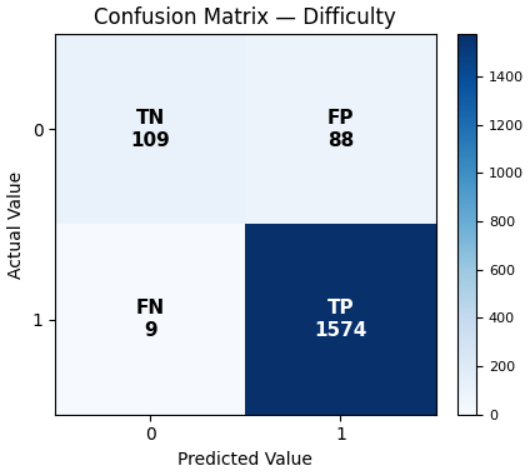

Figure 5.7: Confusion Matrix for Difficulty prediction.

five software metrics. HV, Difficulty, and LOC produced very few errors, whereas CC and comments contained higher counts of false cases.

5.1.5.2 Per-Class Performance Metrics

Table 5.3 summarizes the precision, recall, and F1-scores for each metric. HV, LOC, and Difficulty all show F1-scores above 0.97, with precision and recall consistently high across these metrics. Cyclomatic Complexity (CC) yielded lower precision (0.82) but relatively high recall (0.94), meaning the model frequently identified instances of higher complexity but also produced more false

positives. For the comment metric, the model obtained balanced values (precision 0.88, recall 0.91, F1-score 0.89), indicating that both under-commented and adequately commented samples were classified with similar reliability.

Table 5.3: Recommendation model performance.

Software Metric	**Precision**	**Recall**	**F1-Score**
HV	0.98	1.00	0.99
LOC	0.97	0.98	0.97
CC	0.82	0.94	0.88
Comment	0.88	0.91	0.89
Difficulty	0.95	0.99	0.97

These results validate that the model is effective in identifying patterns within non-compliant code and can accurately predict where software metric values deviate from compliant standards. In particular:

- HV and Difficulty were predicted with very high accuracy.
- The model demonstrated generalization for all five metrics.
- The tendency to overpredict certain labels (e.g., CC) suggests an area for refinement such as providing a wider variety of training examples or incorporating additional semantic context.

The recommendation model identifies whether a software metric deviates from its compliant standard, with accuracies shown in Table 5.3. This predictive capability is used in the recommendation process, where detected deviations are mapped to targeted suggestions (shown in Table 5.1) for improving the code sample, for example, reducing CC by splitting large functions into several simple functions. Outcomes of this recommendation process confirm that an encoder-based recommendation model can generate meaningful and context-aware guidance for improving non-compliant code, which validates Hypothesis 6.

The overall structure of our research, including the key tasks and their interdependencies, is presented in Figure 5.8.

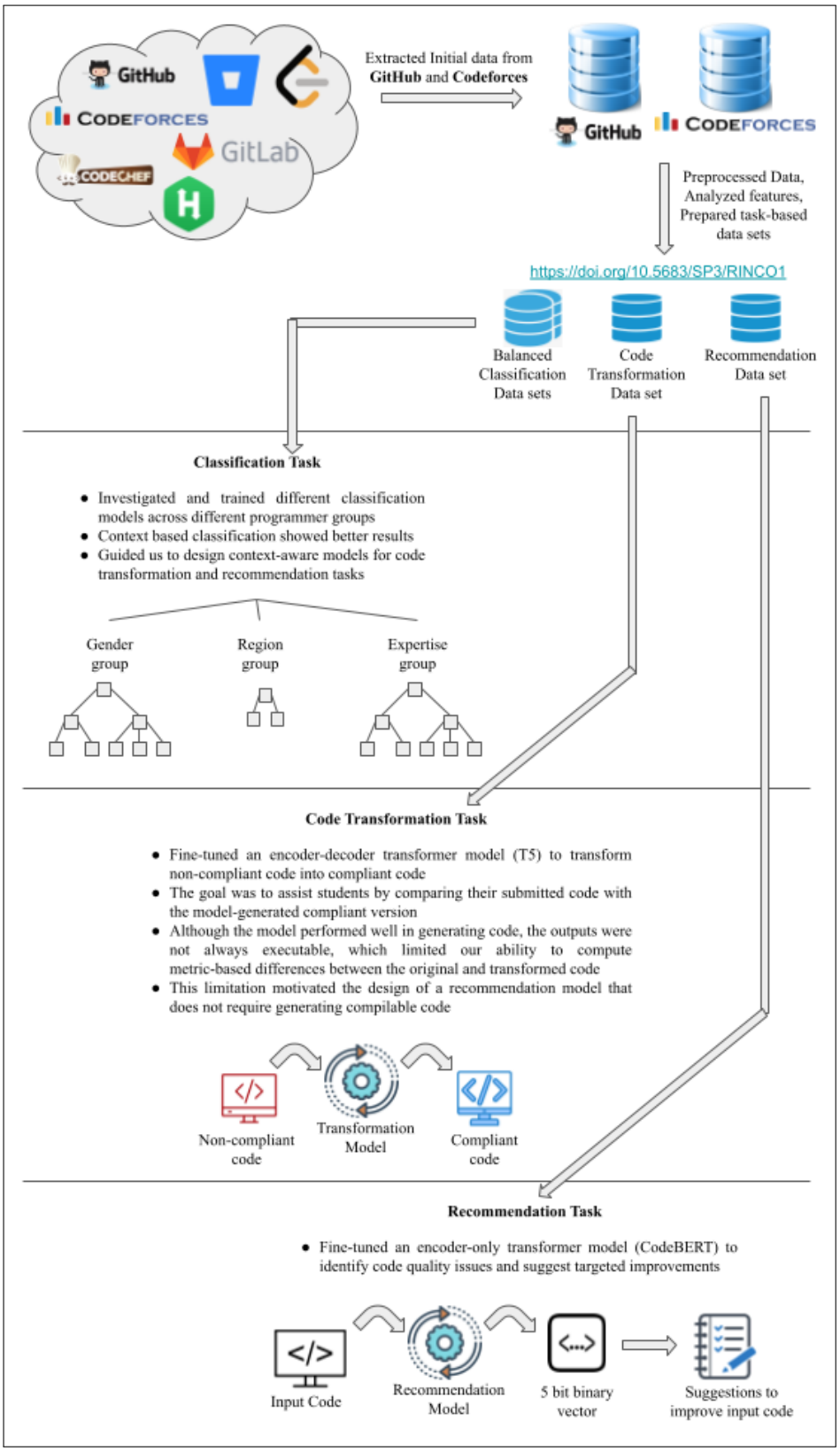

Figure 5.8: Overview of our research workflow.

5.2 Code Insights: A Web-Based Code Recommendation Tool

To make the functionality of our trained recommendation and classification models accessible to end users—particularly students and instructors—we developed a proof-of-concept tool that we named Code Insights. Code Insights is a Flask-based web application designed to analyze and provide feedback on C++ source code. This interactive tool can evaluate code quality using well-established software metrics and offers intelligent suggestions for improvement based on empirical patterns observed in compliant code samples.

5.2.1 Functionality Overview

Code Insights allows the user to input a C++ code snippet using a web interface. Once uploaded, the tool performs markdown several functions:

- **Software Metric Calculation:** The server computes a set of meaningful software metrics, including:
  - Halstead measures: n1 (number of unique operators), n2 (number of unique operands), N1 (total number of operators), N2 (total number of operands), Program vocabulary, Program length, Volume (HV), Difficulty, Effort
  - Structural and stylistic parameters: Lines of code (LOC), Logical lines of code (LLOC), Blank lines, Cyclomatic complexity (CC), and Comment
  - Composite scores: Maintainability Index (MI) and Code Quality (based on the composite equations 2.9 and 2.7)

  These estimated values are then displayed in a table. To support visualization, we chose to represent compliant code (quality score ≥ 0.5) with a green background, indicating good quality. Conversely, non-compliant code (quality score < 0.5) is highlighted with a red background, indicating potential issues. This colour scheme was a design choice in the user interface to make the results more interpretable for learners.

- **Writing Style Classification:** Regardless of the quality score, the tool also returns data regarding the style used in the code sample following two other classification models:

  - Geographic Style Classification: Tells whether the user code exhibits stylistic characteristics associated with high-GDP or steady-GDP countries, based on coding patterns learned from the manually curated balanced binary region data sets (GitHub in Section 3.4.1.2.3, Codeforces in Section 3.4.2.2.3). The model was fine-tuned on these data sources and further evaluated through cross-domain generalization (Section 4.4.2.5).
  - Skill-Level Classification: Indicates whether the programming style is more similar to that of a newbie or a skilled programmer.

  These stylistic notes better place the feedback into context, indicating to users precisely where their code compares to best practices among the various developer communities.

- **Recommendations for Non-Compliant Code:** When a code segment is flagged as non-compliant, Code Insights provides metric-driven feedback by hypothesizing probable reasons for low quality and recommending corrective actions. According to the model's recommendation, the tool will generate a list of potential concerns such as:

  - Halstead volume (HV) seems high.
  - Cyclomatic complexity (CC) seems high.
  - The number of comments seems low.

  Each identified concern is accompanied by actionable suggestions such as:

  - Removing any redundant operands can decrease HV.
  - Splitting large functions into several simple functions can reduce CC.
  - Describe the purpose of each function in the comments.

  These recommendations aim to guide the users towards the improvement of their code in small increments in a methodical and comprehensible manner.

5.2.2 Scope and Expectations

We trained the recommendation model using our prepared data set collected from the Codeforces repository (described in Section 3.4.4). While this offers a strong base for student-written or contest-

style code, the model is not ideal in generalizing to unseen or domain-varying codebases due to the restricted size and scope of training data.

Code Insights is not meant to replace human judgment or perform complete static analysis, but instead as a pedagogical aid for learners. Based on observation of common patterns and metric deviations in real contest solutions, the tool gives constructive, directed feedback to help the user improve their code iteratively.

It is important to note that some algorithms, by their nature, must be of greater complexity or lower maintainability regardless of their implementation quality. Graph algorithms, dynamic programming, and heavy combinatorial logic are examples of algorithms that yield inherently higher complexity ratings on the code but can still be considered compliant if written simply and efficiently. Previous studies [20, 81, 92] on software maintainability and complexity also validate this viewpoint, noting that certain types of programs consistently receive poor ratings on maintainability metrics even with good coding.

Thus, our recommendation model is designed to show cases where compliant solutions are known for non-compliant versions of code. By relative learning from such cases, the tool can guide students away from suboptimal programming habits towards improved solutions that visibly enhance the code quality score. We include a real-world example in the next section and demonstrate how Code Insights identifies quality issues and suggests the appropriate transformations, starting from intentionally degraded code and guiding users to verified compliant solutions.

5.2.3 Interactive Code Evaluation and Refactoring with Code Insights

Our web-based tool, Code Insights, aims to aid learning and code quality enhancement by providing interactive feedback and recommending personalized improvements. This section demonstrates how Code Insights can help programmers, particularly students, improve their C++ code using data-driven suggestions produced by our trained model.

The initial interface of Code Insights is shown in Figure 5.9. It provides a clean and intuitive dashboard in which one may paste C++ code into a text box and receive instant feedback on different software metrics. Upon submission, the tool calculates respective metrics (such as Halstead Volume, Lines of Code (LOC), Cyclomatic Complexity (CC), etc.), including the quality score. The tool

suggests improvements based on specific metric deviations from compliant code practice if the code is flagged as non-compliant (i.e., quality score < 0.5).

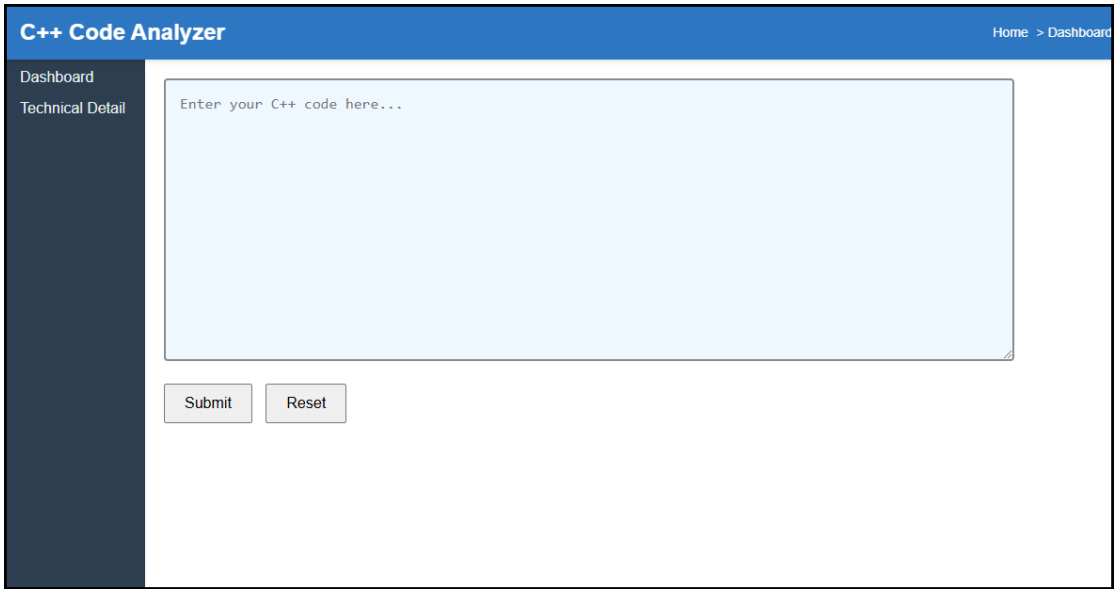

Figure 5.9: Interface of *Code Insights* prototype.

Our first example uses a non-compliant C++ function to calculate the factorial of a number, as shown in Listing 5.1.

After submitting this code to Code Insights, the tool classified it as non-compliant with a quality score of 0.43, as illustrated in Figure 5.10. The tool identified LOC (32 lines) and Cyclomatic Complexity (CC = 5) as exceeding the compliant coding standard. All these results were presented clearly, along with likely causes and corresponding improvement recommendations.

We first chose to reduce LOC. We removed unnecessary print statements (the debugging print statements on lines 5, 6, 9, 15, 28, and 30 of Listing 5.1) to come up with the refactored code in Listing 5.2.

Upon reviewing this version in Code Insights, the quality score improved to 0.46, as shown in Figure 5.11. Although LOC decreased, it remained above the LOC values that our model associates with compliant code. We can continue further enhancements to meet the compliant code standards.

Next, we followed the tool's recommendation to reduce the branching logic by simplifying the conditional structures. Specifically, we eliminated the unnecessary *else* block from line 9 and the unnecessary conditional *return* statement from line 15 of Listing 5.2. The rewritten code is given in

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    cout << "Calculating factorial of " << n << endl;

    int result = 1;
    if (n < 0) {
        cout << "Invalid input." << endl;
        return -1;
    }
    else {
        for (int i = 1; i <= n; i++) {
            result *= i;
            cout << "Intermediate result at i=" << i << ": " << result
   << endl;
        }
    }

    if (result < 0) {
        cout << "Overflow occurred." << endl;
    }

    return result;
}

int main() {
    int num = 5;
    cout << "Starting factorial calculation..." << endl;
    cout << "Factorial: " << factorial(num) << endl;
    cout << "Calculation complete." << endl;
    return 0;
}
```

Listing 5.1: Initial version of *factorial* code sample (Non-compliant).

Listing 5.3.

Executing the final version (Listing 5.3) in Code Insights, the tool classified the code as compliant with a new quality score of 0.50, as seen in Figure 5.12. The CC was reduced to 4 successfully, and the LOC also decreased marginally due to structure simplification. Although the score was just at the threshold, the code successfully met the compliance requirement based on our quality equation.

In addition to reporting quality scores, the tool also compares a user's code with patterns learned from our region and expertise-based classification models (Section 4.3.2). In this example, the classifier suggested that the submission reflected the style of a novice programmer from a high GDP

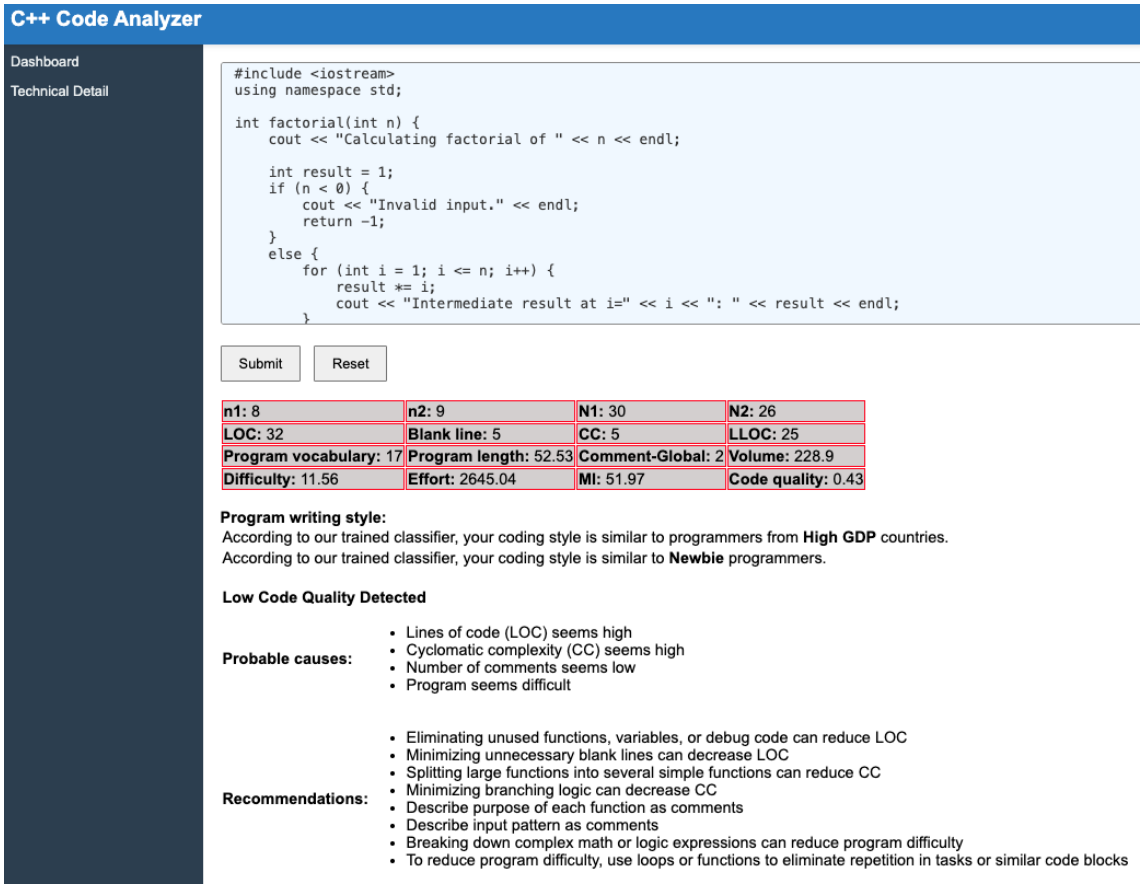

Figure 5.10: Analysis result for the non-compliant factorial code (Listing 5.1).

country. This is not a judgment of correctness, but rather a context that shows how a learner's code relates to broader programming communities. Such feedback can be helpful in two ways: it gives learners a sense of whether their style is closer to that of beginners or more advanced programmers, and it situates their work within regional practices, making them aware of stylistic differences shaped by economic and educational factors. Together with the metric-based recommendations, these stylistic insights encourage learners to consider both the technical quality of their code and how their style aligns with broader programming practices.

This example shows how Code Insights moves beyond the traditional static code analysis to produce a dynamic learning experience. By the process of incrementally analyzing and refining their code, users not only get higher quality scores but also a deeper understanding of software metrics and maintainability. For students and instructors, the tool may offer valuable resource for developing better programming habits through data-driven, contextualized feedback.

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    int result = 1;
    if (n < 0) {
        return -1;
    }
    else {
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
    }

    if (result < 0) {
        return -1;
    }

    return result;
}

int main() {
    int num = 5;
    cout << "Factorial: " << factorial(num) << endl;
    return 0;
}
```

Listing 5.2: Reduced LOC version.

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    int result = 1;
    if (n < 0) return -1;
    for (int i = 1; i <= n; i++) {
            result *= i;
    }

    return result;
}

int main() {
    int num = 5;
    cout << "Factorial: " << factorial(num) << endl;
    return 0;
}
```

Listing 5.3: Optimized version with reduced CC.

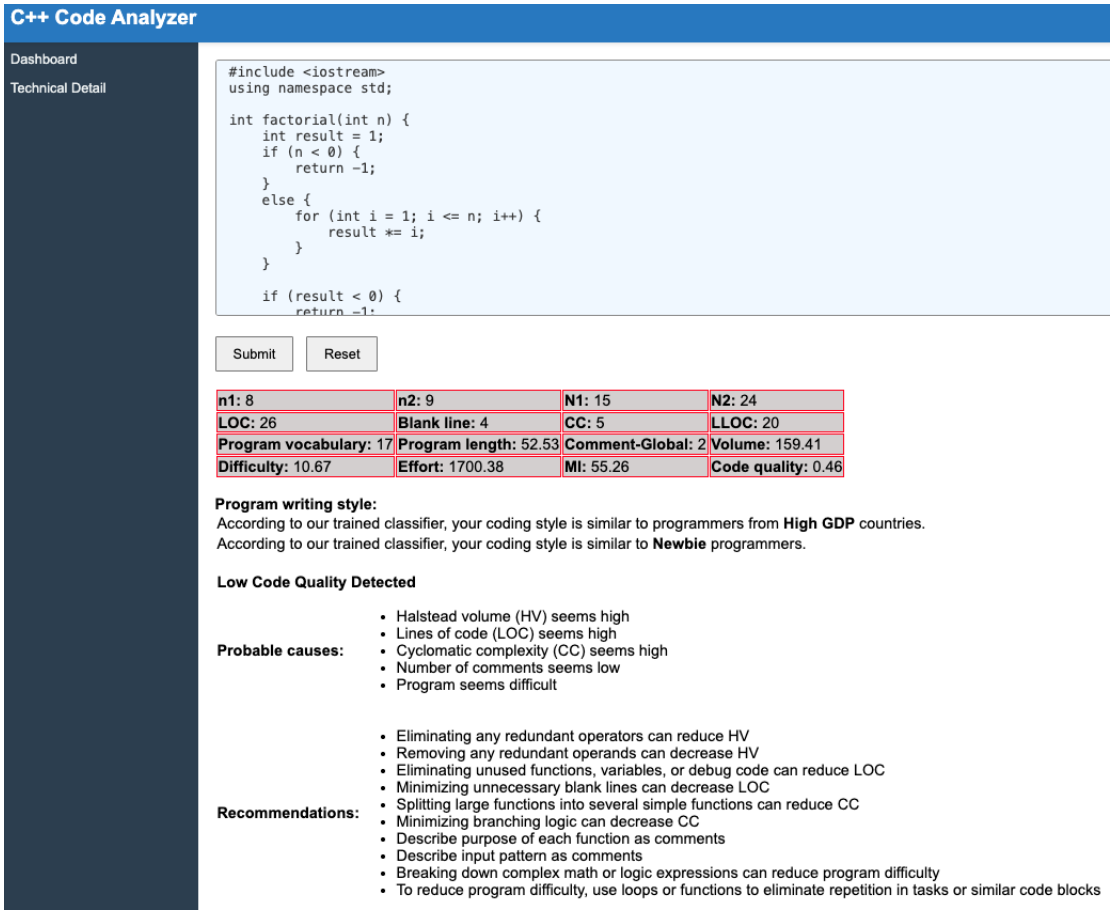

Figure 5.11: Updated result after reducing LOC in the factorial code (Listing 5.2).

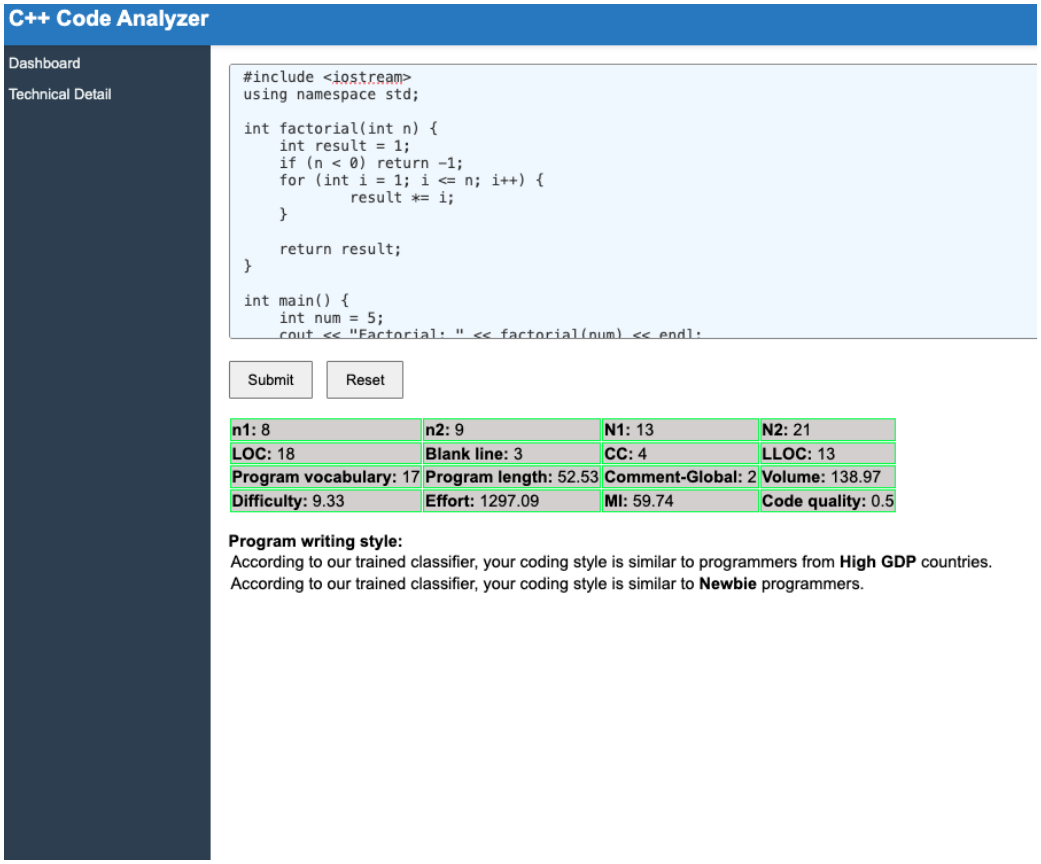

Figure 5.12: Final analysis after reducing CC in the factorial code (Listing 5.3).

Chapter 6

Analysis

This chapter presents the core analyses conducted to investigate how programming styles and code characteristics relate to code quality. The first part examines the connection between software metrics and code quality. We analyze the relationships between code quality and key metrics such as lines of code (LOC), Halstead volume (HV), cyclomatic complexity (CC), comments, and difficulty. These analyses give insight into how different metrics reflect code quality.

Next, we compare the performance of two classification methods—a Random Forest model using software metrics and a BERT-based large language model (LLM) that learns from source code. This comparison evaluates how effectively LLMs comprehend and categorize code quality.

Finally, we discuss the expectations and evaluation of our code transformation model. We look at training and validation performance, highlighting how early stopping helps prevent overfitting. We also address the limitations of encoder-decoder models such as T5, particularly their challenges in generating executable code, and explain our reasons for not choosing alternative methods, including decoder-only LLMs or traditional tools.

6.1 Correlation Between Software Metrics and Code Quality

In this section, we examine how individual software metrics relate to the computed code quality score. As described in Section 2.2.1, code quality in this study is operationalized as a probability-based score derived from a formula that combines fundamental software metrics—lines of code (LOC), Halstead volume (HV), cyclomatic complexity (CC), comments, and difficulty. This formula extends the Maintainability Index (MI) by including difficulty, following widely accepted practices in software engineering research [25].

The purpose of this analysis is twofold. First, by calculating correlation values, we quantify

the extent to which each metric influences the overall quality score. Second, by visualizing these relationships, we reveal how these metrics vary with code quality, providing insights into whether associations are strong, scattered, or cluster-based. Since quality in this context is a computed value rather than a human judgment, graphical inspection is important for showing which factors dominate and how they interact within the data set.

We begin with a numerical analysis using Pearson's correlation coefficient (Table 6.1) and then present scatter plots (Figures 6.1–6.5) that illustrate the relationships between LOC, HV, CC, comments and difficulty with code quality.

6.1.1 Rationale for Selected Features

We focus on LOC, HV, CC, comments, and difficulty because they provide measurable insights into how easy it is to understand, modify, and extend a given code sample—key indicators of maintainability and overall quality [32, 130]. For example, a program with high cyclomatic complexity and high Halstead volume is more difficult to comprehend and maintain, which corresponds to a lower quality score. In contrast, concise code with fewer branching paths is typically easier to maintain and is reflected in higher quality scores. Thus our correlation analysis aims to clarify how each of these metrics contributes to the maintainability profile of code and to inform how they might be used in targeted feedback and recommendations.

6.1.2 Numerical Correlation Analysis

We first computed Pearson's correlation coefficient (Equation 6.1) to measure the strength and direction of the relationship between code quality and software metrics [98]. This coefficient ranges from -1 to 1, where 0 means no correlation, values between 0 and 1 indicate a positive correlation and values between 0 and -1 represent a negative correlation. If x and y are two numeric variables, then their correlation coefficient r is calculated as follows:

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}} \tag{6.1}$$

where x_i = values of the x-variable in a given sample, $\bar{x}$ = mean of the values of the x-variable,

y_i = values of the y-variable in a given sample, and $\bar{y}$ = mean of the values of the y-variable.

The correlation analysis in Table 6.1 demonstrates the relationships between code quality and software metrics. Higher LOC, HV, difficulty and CC values are generally associated with lower code quality as they indicate more complex and harder-to-maintain code. Table 6.1 shows that LOC, HV, difficulty, and CC negatively correlate with the code quality in the Codeforces data set. From the table, we observe that comments have a negative correlation with code quality, indicating that non-compliant code has more comments than compliant code in the Codeforces data set.

Table 6.1: Correlation between code quality and software metrics.

Metric	**Correlation with code quality**	
	Values	**Positive / Negative**
Lines of code (LOC)	-0.848	Negative
Halstead volume (HV)	-0.800	Negative
Cyclomatic complexity (CC)	-0.814	Negative
Comments	-0.541	Negative
Difficulty	-0.663	Negative

These correlation values have several implications for understanding the nature of the code samples in the data set:

- The strong negative correlation between LOC and code quality ($r = -0.848$) indicates that longer code is strongly associated with lower quality. This suggests that unnecessarily long or verbose code is more likely to be non-compliant or poorly structured.

- Similarly, HV and CC have strong negative correlations ($r = -0.800$ and $r = -0.814$ respectively), showing that high computational or logical complexity often corresponds to lower code quality. This aligns with the intuition that complex code is harder to maintain and understand.

- The difficulty metric also shows a substantial negative correlation ($r = -0.663$), implying that as a code sample becomes harder to understand or modify, its quality tends to decline.

- Interestingly, the negative correlation between comments and quality ($r = -0.541$) suggests that non-compliant code tends to have more comments. One possible interpretation is that de-

velopers may compensate for poor structure or clarity by adding more explanatory comments, or that compliant code tends to be more self-explanatory.

6.1.3 Graphical Analysis of Relationships

While correlations summarize associations numerically, scatter plots provide a more intuitive view of how each metric varies with code quality. Figures 6.1–6.5 present the LOC, HV, CC, comments, and difficulty plotted against quality. All plots generally slope downward, which confirms that larger, more complex, and more difficult code generally has lower calculated quality. The scatter plots also reveal the spread and density of points, giving a more accurate perspective of how such patterns are formed across the data set.

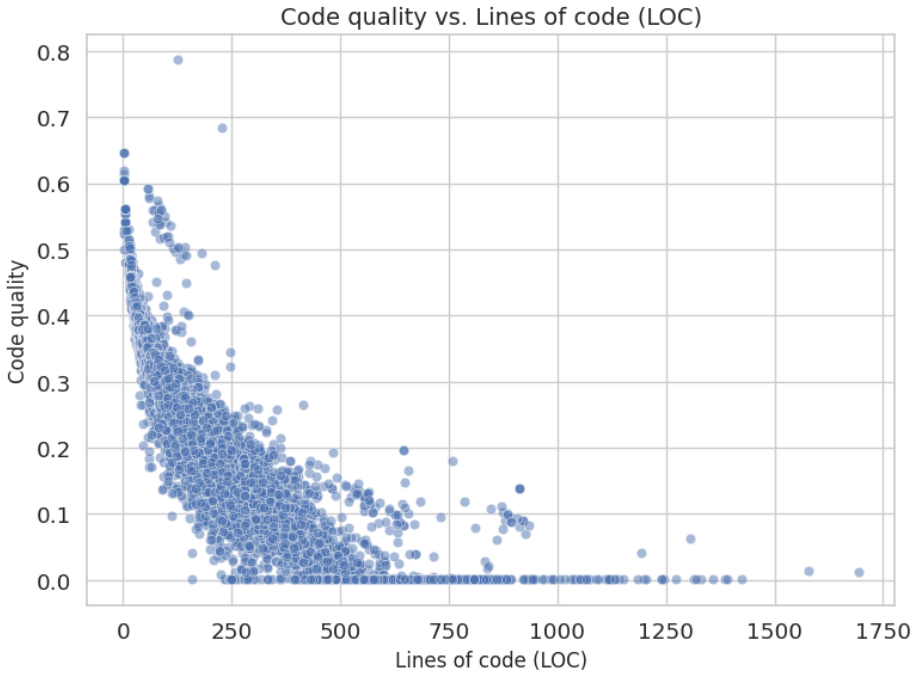

Figure 6.1: Scatter plot of lines of code (LOC) versus code quality.

6.1.4 Interpretation

This analysis helps uncover how individual metrics behave within our data set and what their relative influence is on the composite quality score. Although comments contribute positively to the equation, their negative correlation suggests that their effect may be outweighed by stronger negative factors such as LOC or HV, or may reflect dataset-specific tendencies (e.g., non-compliant code has

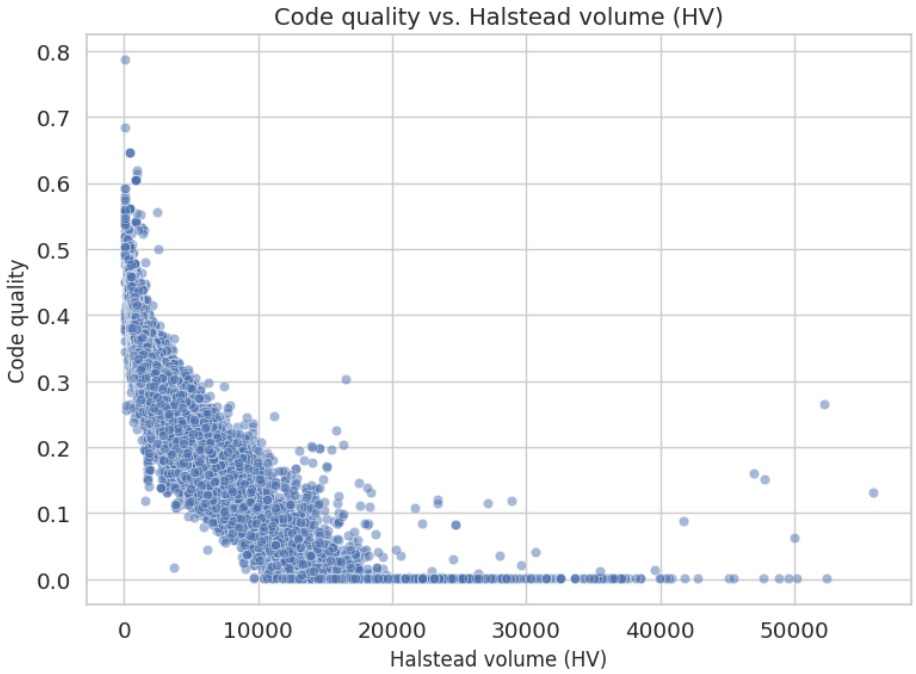

Figure 6.2: Scatter plot of Halstead volume (HV) versus code quality.

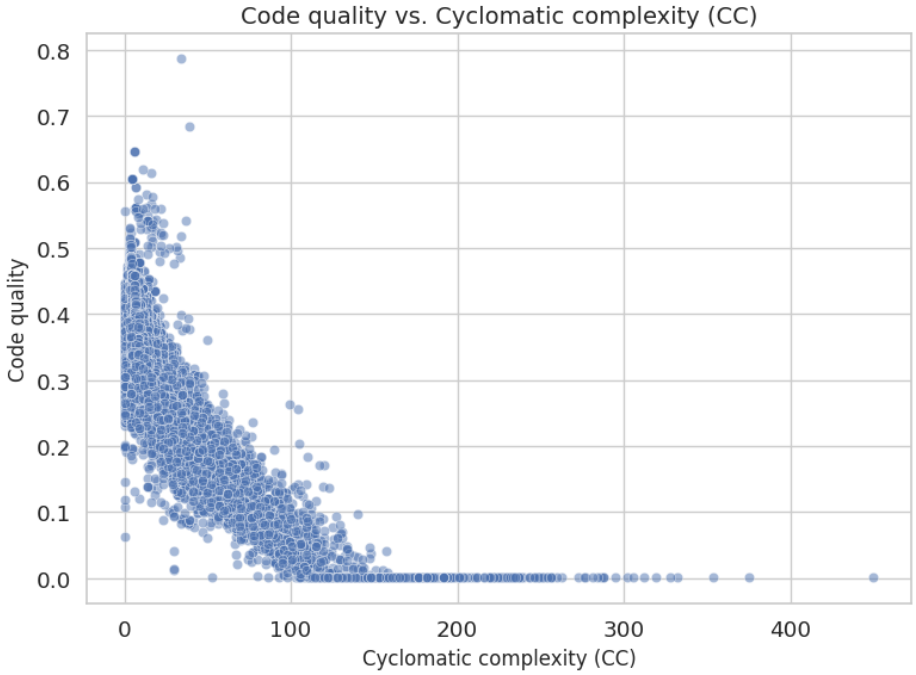

Figure 6.3: Scatter plot of cyclomatic complexity (CC) versus code quality.

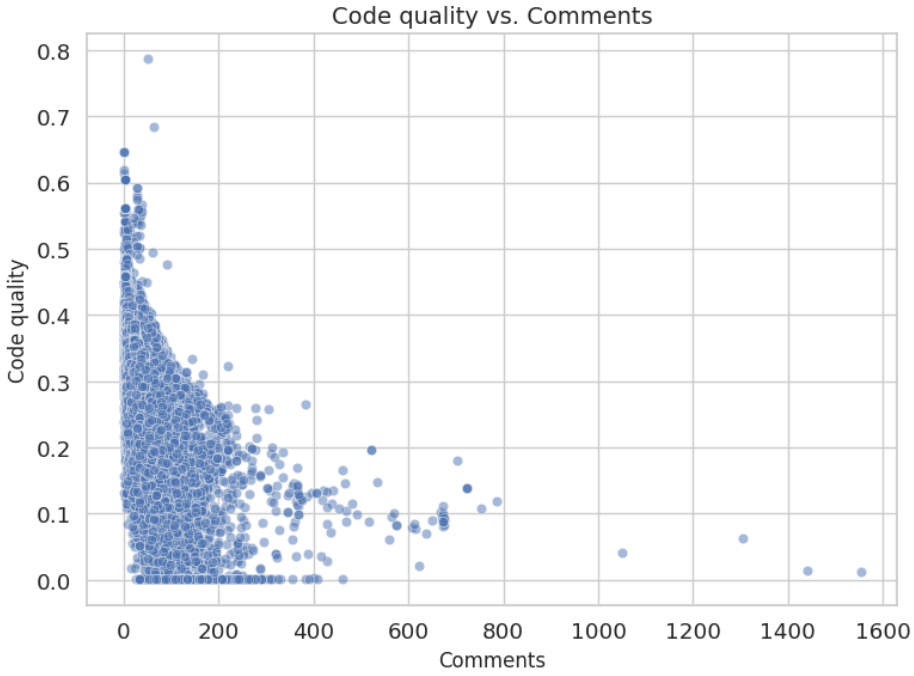

Figure 6.4: Scatter plot of comments versus code quality.

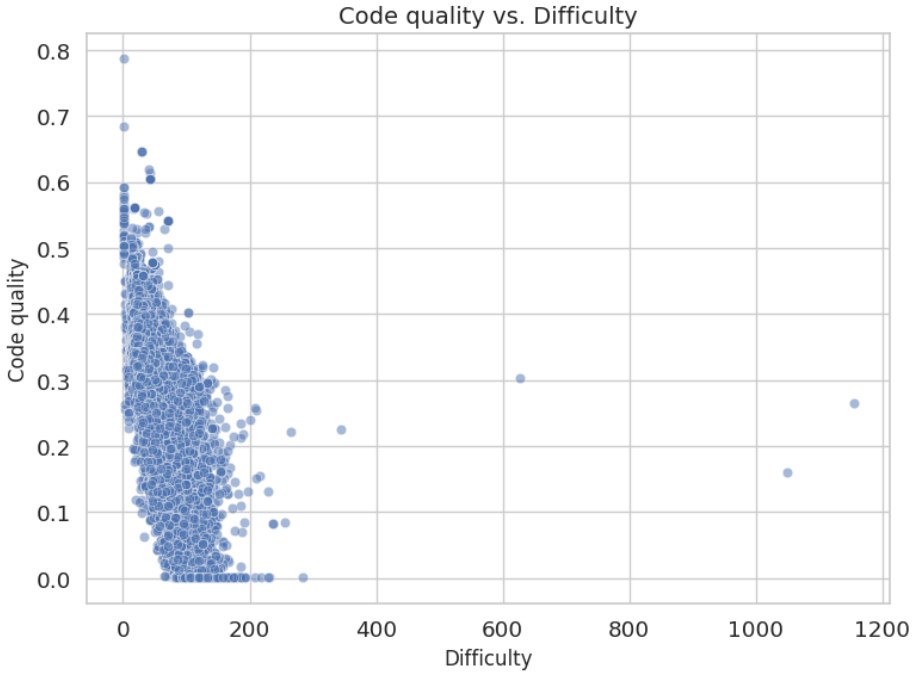

Figure 6.5: Scatter plot of difficulty versus code quality.

more comments than compliant code in the Codeforces data set). LOC, HV, and CC all have negative correlation coefficients with quality, as one would expect, because longer and more complex code is more complicated to maintain. Difficulty also has the predicted negative relationship with code quality.

Based on the correlation coefficients and scatter plots, we observed that the quantitative measures capture the strength of each relationship, while the visualizations illustrate how these relationships are distributed across the data set. This analysis shows how the underlying metrics help in determining the composite quality score. Insights from these findings confirm that a subset of traditional software metrics—including LOC, HV, CC, comments, and difficulty—correlates strongly with code readability, maintainability, and overall quality, which validates Hypothesis 2.

6.2 Performance comparison between BERT and Random Forest

One of our primary research questions aimed to investigate whether large language models (LLMs) could classify groups of programmers from source code more accurately than baseline ML models. This exploration is built on work introduced in [2]. For this, we compared a fine-tuned BERT model with a Random Forest (RF) model on two diverse data sets: Codeforces and GitHub. As shown in Table 6.2, the BERT model consistently outperformed the Random Forest model across all group classifications (gender, region, expertise), but with notable variation in performance gains between data sets and classification tasks.

The improvements are particularly significant for gender classification on both data sets and for all three classification tasks on Codeforces. For example, the F1-score for gender classification on GitHub improved by over 10 percentage points (from 66.36% to 76.9%), and the improvement is even greater on Codeforces (from 61.27% to 75.91%). This suggests that the BERT model is capable of capturing subtle stylistic patterns in code that are not captured by surface-level software metrics.

Interestingly, the BERT model showed relatively small improvements for region classification on the GitHub data set (F1-score increased from 76.8% to 78.47%). However, region classification on the Codeforces data set improved very substantially (from 65.9% to 83.89%). A reason could be the nature of the data. GitHub repositories contain a diverse, wide-ranging set of programming

Table 6.2: Performance comparison between fine-tuned BERT model and Random Forest (RF) classification model.

	Group	**Model**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
GitHub	Gender	RF	66.36	66.36	66.36	66.36
		Fine-tuned BERT	**76.94**	**77.1**	**76.94**	**76.9**
	Region	RF	76.84	76.9	76.8	76.8
		Fine-tuned BERT	**78.79**	**80.12**	**78.79**	**78.47**
Codeforces	Gender	RF	61.3	61.29	61.3	61.27
		Fine-tuned BERT	**76.01**	**76.28**	**76.01**	**75.91**
	Region	RF	65.9	66	65.9	65.9
		Fine-tuned BERT	**83.9**	**84.01**	**83.9**	**83.89**
	Expertise	RF	58.64	58.6	58.6	58.6
		Fine-tuned BERT	**74.54**	**74.55**	**74.54**	**74.53**

problems, application domains (e.g., web development, data analysis, systems programming), and project goals. Such diversity introduces variability that obscures fine-grained regional coding practices [58]. Codeforces competitors, however, all compete on the same set of constant, given competitive programming problems. This shared problem context constrains the functional requirements of the code, meaning that regional stylistic differences—such as preferences in structuring loops, writing conditions, or adding explanatory comments—stand out more clearly against the otherwise uniform task. In other words, the narrower and more standardized context of Codeforces reduces background noise and makes region-specific patterns more salient to the BERT model [43, 132].

In contrast, gender classification shows the opposite trend: it is more precise on the GitHub data set than on Codeforces. One possible explanation is that GitHub's broader set of programming tasks provides richer opportunities for stylistic variation to emerge. Previous research in software engineering and sociolinguistics shows that cognitive or educational expertise are mirrored in stylistic tendencies like naming conventions, commenting strategies, or modularization techniques [15, 30]. These tendencies may align, at least in part, with gendered patterns in problem solving or commu-

nication. On Codeforces the narrow and highly constrained nature of the tasks limits the ways such stylistic preferences can be expressed, which reduces the observable distinction between gender groups. While regional styles benefit from having limited problem contexts where they are easier to detect, gender-related styles may only become visible when programmers work across a wider variety of contexts, as seen in GitHub repositories.

These results support Hypothesis 4 that context-aware models such as BERT are more effective at capturing high-level stylistic and linguistic differences between programmer groups. As discussed in Section 4.5, traditional software metrics may be less indicative of group identity in an era where programmers rely heavily on common online resources and templates [15]. However, despite the growing use of online resources to solve similar problems [15], programmers may exhibit different idioms in naming conventions, code structuring, and logical flow—all of which are better captured by transformer-based models [4, 33].

Although the fine-tuned BERT model tends to perform better than the Random Forest baseline in general, especially for structured data sets like Codeforces, it is crucial to understand what contributes to its performance drivers. These findings emphasize the growing importance of contextual and linguistic features in programmer classification tasks and suggest promising avenues in building more customized, group-sensitive recommendation systems.

6.3 Analysis of Code Transformation Task

In this study we fine-tuned the T5 model to perform code transformation—specifically, converting non-compliant code into compliant code. As defined in Section 2.2.1.5, code is considered 'compliant' if its computed quality score exceeds a given threshold, and 'non-compliant' otherwise. For the transformation data set (Section 3.4.3), we used a stricter threshold of 0.45 to select non-compliant examples. To our knowledge this is the first attempt in the literature to apply a large language model (LLM) for such a transformation task. Our work establishes F1 results as discussed in Section 4.4.4. The model demonstrates moderate performance on code transformation, with F1-scores of 42.13% for Rouge-1, 29.43% for Rouge-2, and 41.01% for Rouge-L, as well as a BLEU score of 0.2371.

6.3.1 Training Performance and Early Stopping

Figure 6.6 shows the training vs. validation loss while fine-tuning the T5 model for the code transformation task. Here, training loss measures how well the model fits the training data, while validation loss reflects how well the model generalizes to unseen data [33]. Monitoring both helps identify overfitting and determine the appropriate point for early stopping. This distinction also highlights why the data set must be divided into separate training, validation, and test sets: the training set is used for model fitting, the validation set for tuning and early stopping, and the test set for unbiased final evaluation.

We used three epochs for the fine-tuning to reduce the chance of overfitting. Both training and validation losses decrease as the number of epochs increases. The training and validation losses were approximately 0.5 and 0.39 in the first epoch. In the last epoch, both losses decreased to 0.26 and 0.22. The gradual decrease in training and validation loss ensures no sign of overfitting in our training process and the model's stability. The small gap between training and validation losses suggests that the model is learning and can perform effectively to unseen validation data.

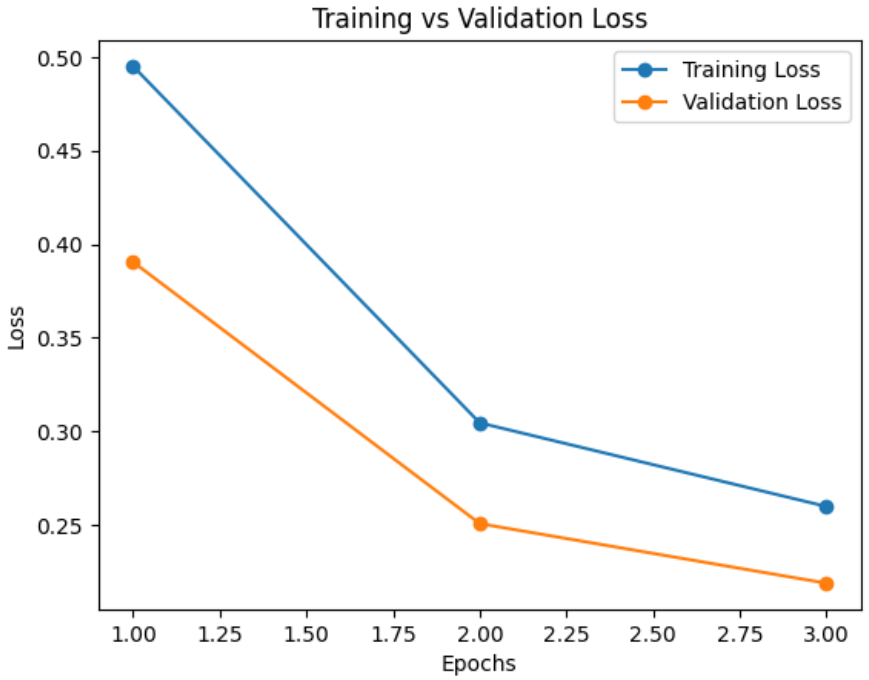

Figure 6.6: Training and Validation Loss of Code Transformation Model.

However, we applied early stopping after only three epochs to reduce the risk of overfitting, as our data set lacked diverse compliant code samples. By 'diverse', we refer to the limited variation in the compliant code used as transformation targets. As described in Section 3.4.3, although we began with 59,534 non-compliant samples from 50 different problem sets, we only had 14,863 compliant programs available. To form transformation pairs, we selected two representative compliant programs per problem set—one from a high GDP country and one from a steady GDP country, yielding just 100 compliant reference programs across the 50 problems. This constrained set resulted in many non-compliant samples being mapped to a small pool of target programs, thereby increasing the risk of overfitting.

Since different programmers wrote compliant and non-compliant programs, the approaches used to solve the same problem often differed significantly. To mitigate noise from structural differences between independently written code samples, we computed the cosine similarity between source–target embeddings and retained only those with a similarity of 0.6 or higher. Cosine similarity is a widely used way of comparing text or code because it focuses on the meaning rather than the length of the sequences [106], and it has proven effective in applications such as filtering parallel corpora in machine translation [63]. Researchers in software engineering have long used cosine similarity to weed out code snippets that do not really belong together [83]. Recently, the same idea has been applied in code-to-code translation, where it helps match pairs of programs that are close enough in structure to be useful for training [129]. Building on these earlier approaches, our filtering step ensures that the training pairs are similar enough in meaning to allow for a realistic transformation, which reduces noise and lowers the risk of overfitting. Again, the limited range of target programs meant the model still had only a narrow view of different coding styles across groups. In future work incorporating a larger and more varied pool of compliant samples—capturing differences across gender, region, and expertise—would enable richer pairing criteria and support continued training with less concern about overfitting.

6.3.2 Pros and Cons of Encoder-Decoder LLMs (T5)

Encoder-decoder language models such as T5, work effectively mostly for the natural language processing tasks; however, they have some limitations when applied to code generation

tasks—particularly when the objective is to generate executable and functional programs. One primary issue is that models like T5 were originally pre-trained on natural language corpora such as the C4 data set [102], and are thus optimized for generating fluent, syntactically well-formed natural text. However, this does not guarantee the semantic correctness or executability of generated code, particularly without extensive fine-tuning on code-specific data [21, 76].

We observed in our experiments that the code generated by the T5 model could not be compiled. This is consistent with outcomes in previous work, which show that even state-of-the-art large language models (LLMs) trained on code can not generate compilable or semantically correct code without further processing or constraint policies [4, 21, 76]. The source of the problem lies in the fact that such models may be optimized for predicting the next token rather than ensuring the correctness of programs. Thus, even when the generated code appears compliant in terms of surface structure or aesthetics, it can be non-compliant when it comes to compilation rules or functional expectations [65].

These limitations pose significant challenges in attempting to compute software metrics — especially Halstead metrics — which require the presence of at least one valid function in a syntactically correct and compilable source file [48]. Since the generated code is frequently non-compilable, we are unable to apply these metrics to quantitatively assess differences between non-compliant input code and the generated output. As a result, we relied on traditional text-based evaluation metrics such as BLEU and ROUGE, following prevailing conventions [95]. These measures may report surface similarity between the generated and reference code, but cannot quantify the executable or behavioural accuracy of the output. Therefore, such measures are insufficient for determining whether the produced code applies to actual use [95].

Still, our research aim was not to produce code that is executable or functional. Instead, we intended to assist learners in enhancing code readability, maintainability, and adherence to best practices—goals that do not necessarily demand compilable code [131]. Providing advice or partial solutions at times may be more pedagogically appropriate than including a complete full solution, especially in educational settings where learning the rationale behind best practices matters more than being provided with working code [131].

We found that encoder-decoder models like T5 are not very good at generating executable or

semantically correct code, which restricts their usefulness in functional correctness tasks. These restrictions made us shift towards a suggestion-based approach through an encoder-only model, something that is better aligned with our aims of conveying stylistic and structural guidance rather than fully executable code.

This redirect opens up an area of future research: building systems that are more like code tutors than code generators, issuing interpretive feedback in terms of coding standards and best practices rather than attempting to replace students' work with generated code. Functional testing—such as compilation validation or test-case verification—is still applicable, especially if the model is eventually built for general use in software engineering. However, for this work our scope is limited to providing guidance around stylistic conformity as part of our proof of concept tool.

6.3.3 Pros and Cons of Other Approaches

In designing our recommendation model, we selected an encoder-based architecture, CodeBERT [33], because of its high potential for deriving contextual code representations. Although alternative approaches to code generation exist such as decoder-only large language models and traditional compilers, they were not selected because they are not compatible with our goal of providing recommendations that focus on education, style, and maintainability. The rationale behind these selections is explained in the subsequent subsections.

6.3.3.1 Decoder-Only LLMs (e.g., ChatGPT)

Decoder-only models such as ChatGPT, Codex, and StarCoder have been demonstrated to work well on code and natural language generation tasks [76]. Such models learn to produce output by finding patterns within large volumes of training data. They read a prompt and produce likely completions based on statistical patterns in the data on which they were trained. This makes them highly effective at generating meaningful text or executable code for standard tasks, particularly when the prompt specifies the desired output.

For example, if a user asks, 'Write a C++ function to compute the factorial of a number', a decoder-only model like ChatGPT will typically return a syntactically correct and executable response as shown in Figure 6.7.

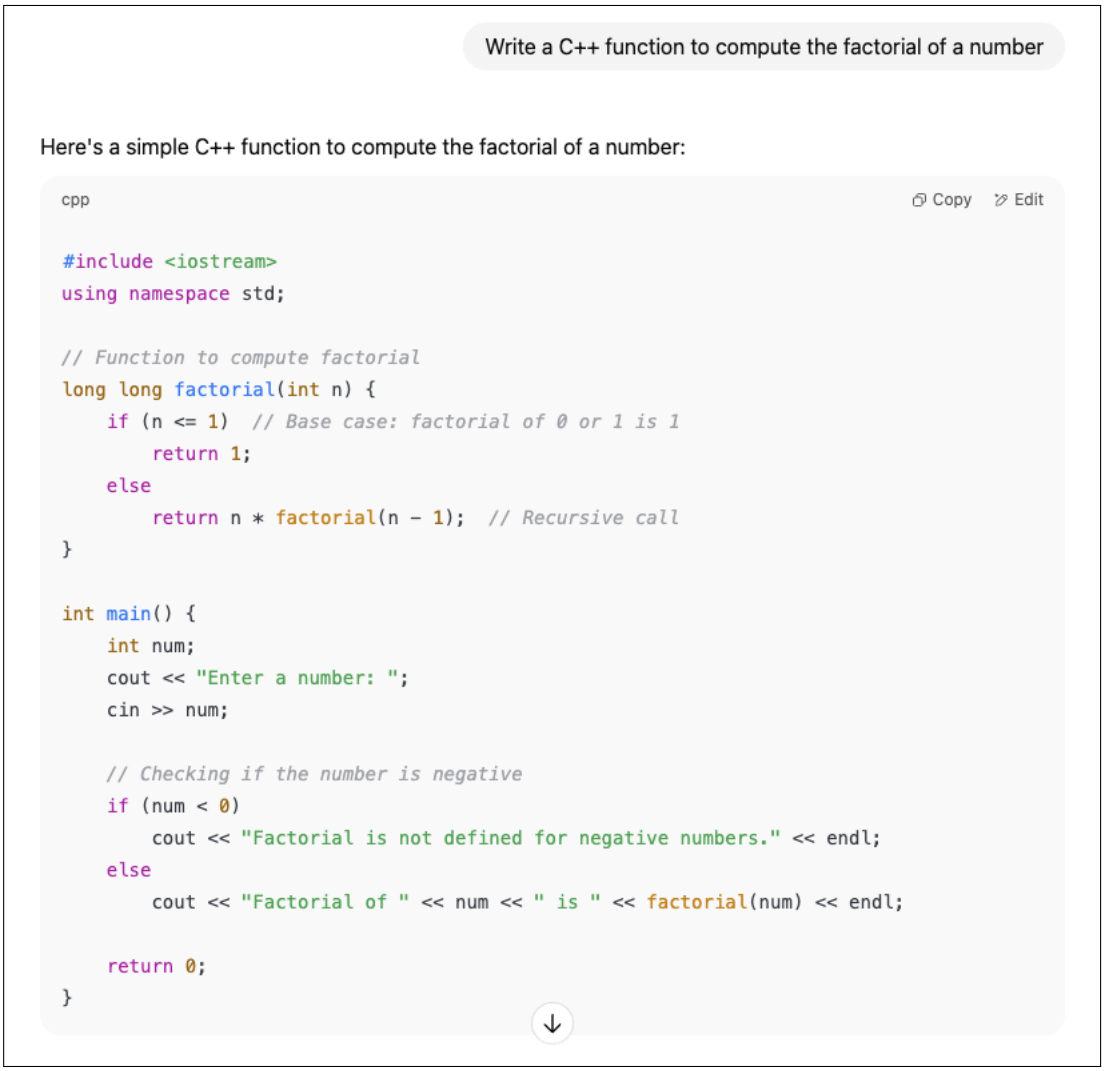

Figure 6.7: Response from ChatGPT.

This response appears accurate and functional because it aligns with patterns the model has seen during pretraining. Such tasks — where the input describes what to generate — play to the strengths of decoder-only models.

However, these models are limited when it comes to tasks that require understanding and reasoning over the structure and context of the input itself. Decoder-only models do not include an explicit encoder component that processes the input to create a contextual representation. As a result, they are generally unsuitable for tasks like classification, translation, or transformation, where interpreting and analyzing the input is essential before producing the output [99].

In contrast, tasks like code transformation or recommendation — which require identifying

non-compliant constructs and producing targeted, explainable suggestions — depend heavily on the model's ability to understand the input structure and semantics. This is where encoder-based models (like CodeBERT) or encoder-decoder models (like T5) are more appropriate. These models process the entire input to generate context representations, using that information to make informed decisions about how to transform or classify the input. For example, our encoder-based model generates structured outputs (e.g., binary labels indicating metric deviations such as line length or nesting depth) that reflect an understanding of the code's properties — something that decoder-only models cannot easily be fine-tuned to perform.

While decoder-only models are effective at generating fluent and reasonable outputs based on past data, they are not well suited for tasks that require deep contextual understanding, structured transformation, or interpretable recommendations [99]. These limitations guided our decision to adopt an encoder-based approach in this research, where the goal was to detect and correct non-compliant coding patterns in a way that is both explainable and pedagogically useful.

6.3.3.2 Traditional Tools

The second possible solution to writing more maintainable code can be the use of tools such as compilers, linters, and static analyzers. These traditional tools are central to software development because they validate code against syntactic and structural rules, report potential bugs, enforce style guidelines, and ensure conformity to language-specific constraints [85].

However, these tools are not designed for maintainability-driven code recommendation or stylistic adaptation [85], which is the focus of our work. Compilers, for instance, are strictly rule-based and are only interested in syntax checking, type checking, and binary code generation. While they can detect issues like uninitialized variables or mismatched braces, they are not designed to suggest improved naming conventions, promote modular design, or enhance code readability [14]. Similarly, linters and style checkers—while capable of flagging formatting or naming issues—operate on predefined static rules and do not generalize or learn from patterns in real-world codebases [85].

Another concern with using such tools in our context is that they are not data-driven and therefore cannot adapt to evolving notions of code readability, maintainability, or community-driven style [14, 85]. For example, while a linter may identify that a function name violates a naming

convention, it cannot learn from examples in a large data set to recommend contextually relevant, semantically meaningful alternatives. These tools lack the ability to generalize or infer intent, making them unsuitable for personalized or context-sensitive feedback.

Data-driven models such as CodeBERT, on the other hand, are trained on labelled examples and are capable of learning abstract patterns of quality, structure, and modularity from real code. By this learning-based strategy, they can provide informed, context-sensitive suggestions that evolve with ongoing coding trends. Although such models may not guarantee executability, they are better suited to offer meaningful feedback on stylistic and maintainability concerns, particularly in educational settings where readability and structure often take precedence over functional completeness [95, 131].

While traditional tools are invaluable for enforcing low-level correctness, they fall short as a means of assisting code recommendation tasks requiring learned knowledge, flexibility, and stylistic nuance. Our model addresses this gap by offering contextually grounded feedback founded on data-driven learning, enabling more flexible and sustainable code improvement strategies.

6.3.4 Potential for Future Applications

The analysis of our code transformation task indicates both limitations and opportunities for future research in code understanding and educational tooling. The transformation results indicate moderate overlap between the generated outputs and the target compliant programs, which provides partial support to Hypothesis 5. However, many of the generated samples failed to compile—a common issue when applying encoder–decoder models such as T5 to code generation. Because many of these outputs failed to compile, we were unable to verify whether the transformations actually reached compliant quality under our metric-based code quality definition.

Since most tokenizers are built for natural language rather than programming syntax, it is not surprising that our model captured stylistic and structural patterns but fell short on functional correctness [33, 105, 122]. Such an ability offers a few directions for future applications.

First, our findings suggest that LLMs can effectively learn and generalize stylistic transformation patterns in code. These patterns help inform future work in code-aware tokenization, where models are not simply fine-tuned on raw code but on structurally highlighted, indented, systematically

named, and modularly enriched forms. Evidence from recent work on identifier-aware and syntax-sensitive tokenization [105, 122] supports this direction, showing that focusing on code structure improves syntactic validity and strengthens models' ability to represent code more effectively.

Second, our approach provides pedagogical tools to guide students in improving their programming style. Syntactic correctness and human-readability tend to be given higher priority than functional equivalence in educational settings. Recent research [131] emphasizes that recommendation systems, which provide improvements rather than presenting direct, executable answers, can have a greater impact on learning engagement and skill development. Our transformation model captures this vision by returning semantically correct recommendations that conform to coding conventions, even when they require more information to be used in practice.

Third, although the current model itself does not generate executable code, its predicted transformations can serve as skeletons or loose outlines of the more advanced post-processing chain. Future researchers can explore techniques like symbolic reasoning, syntax checking, or constraint-based decoding [21, 76] to improve functional correctness in generated code. Our study can form the basis of hybrid systems where program analysis tools and large language models are combined.

Finally, our model's ability to distinguish between compliant and non-compliant patterns can also be used in automated code linting and checking tools where complete execution is not feasible or is not required. These types of tools can alert developers to problems and provide real-time suggestions, allowing them to write cleaner code from the beginning.

Though the responses of the model are not directly actionable, the reflections they provide on stylistic transformation, modelling code, and pedagogical feedback point to opportunities for future exploration. This proof-of-concept highlights the necessity for building models that understand how code is written, as opposed to merely what it writes, an area of growing applicability to education [15] and maintainability-focused software engineering research [26, 86].

Chapter 7

Conclusion

This research began with six research questions and their corresponding hypotheses, first introduced in Chapter 1. Each was motivated by the broader inquiry into whether programming styles—often assumed to be neutral or uniform—actually reflect sociolinguistic influences, and whether software metrics and large language models (LLMs) can be employed to analyze, transform, and improve code. In this final chapter, we return to those questions and explain how the work evolved, what was discovered, and how the hypotheses were ultimately confirmed or challenged.

The first research question inquired whether sociolinguistic variables such as gender, region, and expertise influence the way programmers write, structure, and maintain their code. Analyses from both Phase 1 (software metrics with Random Forest classifiers) and Phase 2 (context-based BERT models) showed that these factors influence the stylistic pattern in program writing. The ability to distinguish programmer groups across different approaches confirmed Hypothesis 1, which supports our claim that sociolinguistic markers are embedded in programming style.

The second and third research questions focused on which software metrics correlate most strongly with code quality, and whether these metrics could effectively classify programmer groups. The analysis in Chapter 6 confirmed that metrics such as cyclomatic complexity (CC), lines of code (LOC), Halstead volume (HV), comments, and Difficulty consistently correlated with maintainability and readability. These metrics also provided the foundation for measurable group classification using a traditional machine learning model such as Random Forest. These results validated Hypotheses 2 and 3: software metrics are meaningful indicators of code quality and can be used to classify programmer groups, although their effectiveness decreases when stylistic differences become more nuanced.

The fourth research question asked whether context-based LLMs outperform metric-based mod-

els in capturing programmer style. In Phase 2, results showed that BERT outperformed Random Forests demonstrating that LLMs can identify better stylistic signatures from raw source code. The comparison in Section 6.2 supports Hypothesis 4 that context-based LLMs outperform metric-based classification approaches.

The fifth research question explored whether stylistic patterns and LLMs could be used for code transformation. The encoder–decoder T5 model trained in Phase 3 was able to produce outputs lexically similar to target compliant code, supporting that it learned patterns of maintainability. However, the generated code frequently failed to compile, which highlighted the validity of this approach. While Hypothesis 5 was partially supported—the model demonstrated measurable stylistic transformation—it also revealed practical limitations of encoder–decoder LLMs for producing executable code.

The sixth and final research question asked whether metric-guided models could generate recommendations that help learners improve their code. In Phase 4 a CodeBERT-based multi-label classifier is fine-tuned that flags deviations in software metrics and provides targeted feedback. This approach overcame the limitations of executable code generation, instead delivering actionable, educational suggestions. Results and analysis of this approach (discussed in Chapter 5) support Hypothesis 6, confirming that encoder-based LLMs can generate precise, metric-based recommendations to support reflective learning and improve code maintainability.

All the experimental results and analyses including the data collection and preparation process of this research show a progression: from confirming that sociolinguistic factors influence programming styles, to demonstrating that software metrics and LLMs capture those differences in complementary ways, to exploring transformation and finally developing a recommendation tool as a pedagogically robust solution. The Code Insight tool represents this journey by bringing together classification, quality measurement, and recommendation into a prototype that learners can use to improve their coding practices.

7.1 Future work

Our research showed that LLMs performed better than the traditional classification model. We also used the LLM to transform non-compliant code into compliant code. Additionally, we fine-

tuned an encoder-based LLM that detects quality flaws in source code and provides specific recommendations for fixing. There are two possible ways to improve our work in future when we use LLMs. We can choose different LLMs to improve performance or incorporate diverse data for the targeted task.

Some of the suggestions for future research are given below:

- As discussed in Section 6.3.2, the T5 model has significant limitations when applied to code generation tasks—particularly when the goal is to produce executable and functional programs. A key reason is that T5 was originally pre-trained on natural language corpora, rather than on programming-specific data. For instance, Table 3.8 illustrates that the T5 model failed to recognize the tokens '{' and '}', which are essential in C++ programming syntax.

- In gender-based data sets, the participation of female programmers is still significantly less than that of male programmers. It is important to address this imbalance because, as discussed in Chapter 1, prior research has shown that gender can influence programming style, confidence, and error-handling strategies. For instance, Sharafi *et al.* [110] observed that female programmers often focus on writing accurate programs and minimizing errors, while male programmers are more likely to prioritize speed. Expanding future data sets with stronger representation of female programmers would make it possible to study these gender-related patterns more reliably.

- In this research we considered 10 countries to classify the group into high or steady GDP regions. In future research, we can incorporate more countries to increase the geographical coverage along with their GDPs.

- We considered 'maxRank' to identify programmers' skill levels. In the future, we can incorporate other parameters such as the employment of advanced libraries, design patterns, or solution techniques, to measure the programmer's skill level.

- In future, we can incorporate a more diverse set of compliant code samples when pairing them with non-compliant code in the transformation data set. As discussed in Section 6.3, our current filtering approach used cosine similarity to ensure that source–target pairs were

close enough in structure to reduce noise and overfitting. While this step improved alignment, it also limited the range of target programs and gave the model only a narrow view of different programming styles. Expanding this pool to include compliant code that spans all programmer group labels (e.g., gender, region, expertise) and a wider range of code quality ranges would ensure broader coverage. Such diversity would strengthen the transformation data set by exposing the model to multiple valid pathways toward compliance and by better reflecting the variation present in real-world programming practices.

- Another important direction for future work is user testing of our prototype tool *Code Insight*. While our current evaluation focused on model performance and technical metrics, the effectiveness of the tool in real learning environments has not yet been assessed. Conducting user studies with both learners and instructors would offer valuable feedback on how the recommendations are understood, how well they encourage reflective improvement, and how the interface could be adjusted to make the tool easier and more effective to use.

This research focused specifically on the C++ programming language. While our study did not directly address broader educational applications, our findings suggest that understanding a learner's sociolinguistic background could be valuable in designing personalized training approaches in programming, mathematics, or other online subjects. Since we showed that sociolinguistic patterns can influence programming styles similarly to natural language, future work could explore how these insights might support more adaptive, learner-centred educational tools.

Bibliography

[1] Deen Mohammad Abdullah. Query focused abstractive summarization using BERTSUM model. Master's thesis, University of Lethbridge (Canada), 2020.

[2] Deen Mohammad Abdullah, Sara Binte Zinnat, and Jacqueline E. Rice. Do sociolinguistic factors influence program writing styles? In *2025 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA)*, pages 1–7. IEEE, 2025.

[3] Mohammed Abuhamad, Tamer Abuhmed, David Mohaisen, and Daehun Nyang. Large-scale and robust code authorship identification with deep feature learning. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–35, 2021.

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668. Association for Computational Linguistics, June 2021.

[5] Sowkat Alam. Computer program complexity and its correlation with program features and sociolinguistics. Master's thesis, University of Lethbridge (Canada), 2021.

[6] Shlomo Argamon, Jean-Baptiste Goulain, Russell Horton, and Mark Olsen. Vive la différence! text mining gender difference in french literature. *Digital Humanities Quarterly*, 3(2), 2009.

[7] Shlomo Argamon, Moshe Koppel, Jonathan Fine, and Anat Rachel Shimoni. Gender, genre, and writing style in formal written texts. *Text & talk*, 23(3):321–346, 2003.

[8] Shlomo Argamon, Moshe Koppel, James W Pennebaker, and Jonathan Schler. Automatically profiling the author of an anonymous text. *Communications of the ACM*, 52(2):119–123, 2009.

[9] Shlomo Argamon, Casey Whitelaw, Paul Chase, Sobhan Raj Hota, Navendu Garg, and Shlomo Levitan. Stylistic text classification using functional lexical features. *Journal of the American Society for Information Science and Technology*, 58(6):802–822, 2007.

[10] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the codeboost transformation system for domain-specific optimisation of C++ programs. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 65–74. IEEE, 2003.

[11] CT Bailey and WL Dingee. A software study using Halstead metrics. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 189–197, 1981.

[12] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. Codeplan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

[13] Priya Bansal and Abdelkader Ouda. Study on integration of FastAPI and machine learning for continuous authentication of behavioral biometrics. In *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE, 2022.

[14] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. How should compilers explain problems to developers? In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 633–643, 2018.

[15] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506, 2023.

[16] Mondher Bouazizi and Tomoaki Ohtsuki. Sentiment analysis: From binary to multi-class classification: A pattern-based approach for multi-class sentiment analysis in twitter. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, 2016.

[17] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[18] Margaret Burnett, Scott D Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. Gender differences and programming environments: across programming populations. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2010.

[19] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX security symposium (USENIX Security 15)*, pages 255–270, 2015.

[20] David N Card and Robert L Glass. *Measuring software design quality*. Prentice-Hall, Inc., 1990.

[21] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, 2020.

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[23] Wuxing Chen, Kaixiang Yang, Zhiwen Yu, Yifan Shi, and CL Philip Chen. A survey on imbalanced learning: latest research, applications and future directions. *Artificial Intelligence Review*, 57(6):137, 2024.

[24] Rohan Roy Choudhury, HeZheng Yin, Joseph Moghadam, and Armando Fox. Autostyle: Toward coding style feedback at scale. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, pages 21–24, 2016.

[25] Shaiful Chowdhury, Reid Holmes, Andy Zaidman, and Rick Kazman. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empirical Software Engineering*, 27(6):158, 2022.

[26] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[27] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, MA Borst, and Tom Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering*, 2:96–104, 1979.

[28] Bogusław Cyganek. Modern c++ in the era of new technologies and challenges—why and how to teach modern c++? In *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 35–40. IEEE, 2022.

[29] Pablo Del Moral, Sławomir Nowaczyk, and Sepideh Pashami. Why is multiclass classification hard? *IEEE Access*, 10:80448–80462, 2022.

[30] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*, pages 1–5, 2007.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[32] Yu Du and Shaoying Liu. A methodology for rating maintainability metrics of sofl formal specifications. In *International Symposium on Software Fault Prevention, Verification, and Validation*, pages 1–14. Springer, 2024.

[33] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[34] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[35] Roy Thomas Fielding and Richard N. Taylor. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[36] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian computing education conference*, pages 10–19, 2022.

[37] UNESCO Institute for Statistics. Global education monitoring report: ICT in education. https://www.unesco.org/gem-report/en/publication/technology-education, 2023. [Online; Last accessed 09-August-2025].

[38] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, pages 893–896, 2006.

[39] GitHub. Acceptable use policies. https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies, 2023. [Online; accessed June 02, 2025].

[40] GitHub. Github docs. https://docs.github.com/en/rest/users, 2023. [Online; accessed October 09, 2023].

[41] GitHub. Privacy statement. https://docs.github.com/en/site-policy/privacy-policies/github-general-privacy-statement, 2023. [Online; accessed June 02, 2025].

[42] GitHub. Innovation Graph. https://github.com/github/innovationgraph/blob/main/data/developers.csv, 2024. [Online; accessed 08-August-2024].

[43] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):1–35, 2015.

[44] Google. Bigquery. https://bigquery.cloud.google.com/, 2023. [Online; accessed October 09, 2023].

[45] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 233–236, May 2013. Best data showcase paper award.

[46] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[47] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[48] T. Hariprasad, G. Vidhyagaran, K. Seenu, and Chandrasegar Thirumalai. Software complexity analysis using Halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113. IEEE, 2017.

[49] Jeanette Heidenberg, Max Weijola, Kirsi Mikkonen, and Ivan Porres. A metrics model to measure the impact of an agile transformation in large software development organizations. In *Agile Processes in Software Engineering and Extreme Programming: 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings 14*, pages 165–179. Springer, 2013.

[50] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[51] James D Herbsleb and Deependra Moitra. Global software development. *IEEE software*, 18(2):16–20, 2002.

[52] Tjaša Heričko and Boštjan Šumak. Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences*, 13(5):2972, 2023.

[53] Tu Honglei, Sun Wei, and Zhang Yanan. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications*, volume 1, pages 131–136. IEEE, 2009.

[54] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, September 2024. Just Accepted.

[55] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[56] Richard A Hudson. *Sociolinguistics*. Cambridge university press, 1996.

[57] Blom Jan-Petter and John J Gumperz. Social meaning in linguistic structure: Code-switching in Norway. In *The bilingualism reader*, pages 75–96. Routledge, 2020.

[58] Parisa Kaghazgaran, Nichola Lubold, and Fred Morstatter. Organizational artifacts of code development. *arXiv preprint arXiv:2105.14637*, 2021.

[59] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *Research into Practice–Reality and Gaps: 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings 6*, pages 151–166. Springer, 2010.

[60] Gilang Heru Kencana, Akuwan Saleh, Haryadi Amran Darwito, R. Rizki Rachmadi, and Elsa Mayang Sari. Comparison of maintainability index measurement from Microsoft Codelens and line of code. In *2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI)*, pages 235–239, 2020.

[61] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[62] Ali Athar Khan, Amjad Mahmood, Sajeda M Amralla, and Tahera H Mirza. Comparison of software complexity metrics. *International Journal of Computing and Network Technology*, 4(01), 2016.

[63] Philipp Koehn, Francisco Guzmán, Vishrav Chaudhary, and Juan Pino. Findings of the wmt 2019 shared task on parallel corpus filtering for low-resource conditions. In *Proceedings of the Fourth Conference on Machine Translation (Volume 3: Shared Task Papers, Day 2)*, pages 54–72, 2019.

[64] Moshe Koppel, Shlomo Argamon, and Anat Rachel Shimoni. Automatically categorizing written texts by author gender. *Literary and linguistic computing*, 17(4):401–412, 2002.

[65] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

[66] Miikka Kuutila, Mika Mäntylä, Umar Farooq, and Maelick Claes. Time pressure in software engineering: A systematic review. *Information and Software Technology*, 121:106257, 2020.

[67] William Labov. *Sociolinguistic patterns*. University of Pennsylvania press, 1973.

[68] Luigi Lavazza, Abedallah Zaid Abualkishik, Geng Liu, and Sandro Morasca. An empirical evaluation of the "cognitive complexity" measure as a predictor of code understandability. *Journal of Systems and Software*, 197:111561, 2023.

[69] Edward E Leamer and Edward E Leamer. Gross domestic product. *Macroeconomic patterns and stories*, pages 19–38, 2009.

[70] Fritz Lekschas, Spyridon Ampanavos, Pao Siangliulue, Hanspeter Pfister, and Krzysztof Z Gajos. Ask me or tell me? enhancing the effectiveness of crowdsourced design feedback. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2021.

[71] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering*, 34(4):485–496, 2008.

[72] Xiaozhou Li, Noman Ahmad, Tomas Cerny, Andrea Janes, Valentina Lenarduzzi, and Davide Taibi. Toward organizational decoupling in microservices through key developer allocation. In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, pages 16–20. IEEE, 2025.

[73] Suryadiputra Liawatimena, Harco Leslie Hendric Spits Warnars, Agung Trisetyarso, Edi Abdurahman, Benfano Soewito, Antoni Wibowo, Ford Lumban Gaol, and Bahtiar Saleh Abbas. Django web framework software metrics measurement using radon and pylint. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, pages 218–222. IEEE, 2018.

[74] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[75] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA'08, page 131–142, New York, NY, USA, 2008. Association for Computing Machinery.

[76] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[77] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, Online, November 2020. Association for Computational Linguistics.

[78] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[79] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 931–937, 2023.

[80] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[81] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 4:308–320, 1976.

[82] Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[83] Micheline Bénédicte Moumoula, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F Bissyandé. The struggles of llms in cross-lingual code clone detection. *Proceedings of the ACM on Software Engineering*, 2(FSE):1023–1045, 2025.

[84] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 1–12, 2020.

[85] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543, 2022.

[86] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461, 2006.

[87] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[88] United Nations. National Accounts - Analysis of Main Aggregates (AMA). `https://unstats.un.org/UNSD/snaama/Downloads`, 2024. [Online; accessed 08-August-2024].

[89] Fariha Naz and Jacqueline E Rice. Sociolinguistics and programming. In *2015 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 74–79. IEEE, 2015.

[90] John Noll, Sarah Beecham, and Ita Richardson. Global software development and collaboration: barriers and solutions. *ACM inroads*, 1(3):66–78, 2011.

[91] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[92] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society, 1992.

[93] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[94] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[95] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Benchmarks and metrics for evaluations of code generation: A critical review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 87–94. IEEE, 2024.

[96] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*, pages 1093–1102. PMLR, 2015.

[97] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[98] Iwan Handovo Putro and Tohari Ahmad. Feature selection using pearson correlation with lasso regression for intrusion detection system. In *2024 12th International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6, 2024.

[99] Muhammad Qorib, Geonsik Moon, and Hwee Tou Ng. Are decoder-only language models better than encoder-only language models in understanding word meaning? In *Findings of the Association for Computational Linguistics ACL 2024*, pages 16339–16347, 2024.

[100] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[101] Md Mahmudul Hasan Rafee. Computer program categorization with machine learning. Master's thesis, University of Lethbridge (Canada), 2017.

[102] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[103] Leonard Richardson. Beautiful soup documentation. *April*, 2007.

[104] Saima Ritonummi, Valtteri Siitonen, Markus Salo, and Henri Pirkkalainen. Flow barriers: What prevents software developers from experiencing flow in their work. In *Socio-Technical Perspective in Information Systems Development*, pages 247–264. CEUR Workshop Proceedings, 2022.

[105] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[106] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[107] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[108] Neela Sawant and Srinivasan H Sengamedu. Code compliance assessment as a learning problem. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 445–454. IEEE, 2023.

[109] Paul Sebo. Performance of gender detection tools: a comparative study of name-to-gender inference services. *Journal of the Medical Library Association: JMLA*, 109(3):414, 2021.

[110] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Women and men—different but equal: On the impact of identifier style on source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 27–36. IEEE, 2012.

[111] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.

[112] Diomidis Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[113] Enrico Almer Tahara, Abdullah Faqih Septiyanto, Riyanarto Sarno, and Shoffi Izza Sabilla. Software quality enhancement through code refactoring: An empirical evaluation of roblox game development. In *2025 International Conference on Computer Sciences, Engineering, and Technology Innovation (ICoCSETI)*, pages 739–744. IEEE, 2025.

[114] Nazia Tasnim. *Machine learning in the classification of computer code*. University of Lethbridge (Canada), 2020.

[115] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th international conference on software engineering*, pages 237–248, 2022.

[116] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[117] Alan M Turing. *Computing machinery and intelligence*. Springer, 2009.

[118] Farhan Ullah, Junfeng Wang, Sohail Jabbar, Fadi Al-Turjman, and Mamoun Alazab. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access*, 7:141987–141999, 2019.

[119] Alexander D VanHelene, Ishaani Khatri, C Beau Hilton, Sanjay Mishra, Ece D Gamsiz Uzun, and Jeremy L Warner. Inferring gender from first names: Comparing the accuracy of genderize, gender api, and the gender r package on authors of diverse nationality. *PLOS digital health*, 3(10):e0000456, 2024.

[120] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[121] Petr Vytovtov and Evgeny Markov. Source code quality classification based on software metrics. In *2017 20th Conference of Open Innovations Association (FRUCT)*, pages 505–511. IEEE, 2017.

[122] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[123] Ronald Wardhaugh and Janet M Fuller. *An introduction to sociolinguistics*. John Wiley & Sons, 2021.

[124] Noreen M Webb. The role of gender in computer programming learning processes. *Journal of Educational Computing Research*, 1(4):441–458, 1985.

[125] Kurt D Welker, Paul W Oman, and Gerald G Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.

[126] Ian H Witten, Eibe Frank, Mark A Hall, Christopher J Pal, and Mining Data. Practical machine learning tools and techniques. In *Data mining*, volume 2, pages 403–413. Elsevier Amsterdam, The Netherlands, 2005.

[127] World Bank. Information and communications for development: Leveraging digital technologies for sustainable growth. `https://docs.un.org/en/A/RES/79/194`, 2023. [Online; Last accessed 09-August-2025].

[128] Jones Yeboah and Saheed Popoola. Efficacy of static analysis tools for software defect detection on open-source projects. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1588–1593. IEEE, 2023.

[129] Longhui Zhang, Bin Wang, Jiahao Wang, Xiaofeng Zhao, Min Zhang, Hao Yang, Meishan Zhang, Yu Li, Jing Li, and Jun Yu. Function-to-style guidance of llms for code translation. In *Forty-second International Conference on Machine Learning*, 2025.

[130] Fang Zhuo, Bruce Lowther, Paul Oman, and Jack Hagemeister. Constructing and testing software maintainability assessment models. In *First International Software Metrics Symposium*, pages 61–70. IEEE, 1993.

[131] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024.

[132] Sara Binte Zinnat. Classification of computer programming contest programs based on gender, region and software metrics. Master's thesis, University of Lethbridge (Canada), 2021.

[133] David Álvarez Fidalgo and Francisco Ortin. Efficient source code authorship attribution using code stylometry embeddings. In *Proceedings of the 20th International Conference on Software Technologies - Volume 1: ICSOFT*, pages 167–177. INSTICC, SciTePress, 2025.

Appendix A

Additional details

A.1 Information gain

Let us consider the values in Table A.1 as sample data on home buyers' credit scores and incomes to calculate the attributes' information gain.

Table A.1: Sample data on credit scores and incomes of home buyers.

Credit score	Income	Buy home
Excellent	Low	No
Low	High	Yes
Average	Medium	Yes
Excellent	High	Yes
Average	Low	No

- **Step-1: Calculate probability for each class of the attribute 'Buy home'**

  Let P_i be the probability that a data point in the training data set D belongs to class C_i. We can calculate the probability of each class of the 'Buy home' attribute using Equation A.1.

$$P_i = \frac{\text{the number of records in } D \text{ that belong to } C_i}{\text{the total number of records in } D} \tag{A.1}$$

  Class Y: buys home = Yes $\implies P_Y = \frac{3}{5}$

  Class N: buys home = No $\implies P_N = \frac{2}{5}$

- **Step-2: Calculate the expected information (Entropy) needed to classify a data record**

  We calculate the entropy of this data set using Equation A.2.

$$\text{Info}(D) = -\sum_{i=1}^{m} P_i \log_2(P_i) \tag{A.2}$$

  Here, m is the number of classes that D contains.

  $\text{Info}(D) = -P_Y \log_2 P_Y - P_N \log_2 P_N = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971$

- **Step-3: For each attribute, check how many values the attribute has**

  We partition the data set D according to each attribute as shown in Tables A.2 and A.3.

Table A.2: Frequency of class for 'Buy home' depending on 'Credit score' attribute.

Records	**Credit score**	**Buy home**		**Total**
		Yes	**No**	
$D_{Excellent}$	Excellent	1	1	2
$D_{Average}$	Average	1	1	2
D_{Low}	Low	1	0	1

Table A.3: Frequency of class for 'Buy home' depending on 'Income' attribute.

Records	**Income**	**Buy home**		**Total**
		Yes	**No**	
D_{High}	High	2	0	2
D_{Medium}	Medium	1	0	1
D_{Low}	Low	0	2	2

- **Step-4: Calculate the expected information (Entropy) to classify D using each partitioned attribute**

  We calculate the expected information for attributes 'Credit score' and 'Income' using Equation A.3.

$$\text{Info}_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \text{Info}(D_j) \tag{A.3}$$

Here, v is the value in attribute A.

$\text{Info}(D_j) = -\sum_{i=1}^{m} P_i \log_2(P_i)$, where m is the number of classes in D_j.

$$\text{Info}_{CreditScore}(D) = \frac{|D_{Excellent}|}{|D|}\text{Info}(D_{Excellent}) + \frac{|D_{Avg}|}{|D|}\text{Info}(D_{Avg}) + \frac{|D_{Low}|}{|D|}\text{Info}(D_{Low})$$
$$= \frac{2}{5}(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}) + \frac{2}{5}(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}) + \frac{1}{5}(-\frac{1}{1}\log_2\frac{1}{1} - \frac{0}{1}\log_2\frac{0}{1}) = 0.8$$

$$\text{Info}_{IncomeRange}(D) = \frac{|D_{High}|}{|D|}\text{Info}(D_{High}) + \frac{|D_{Medium}|}{|D|}\text{Info}(D_{Medium}) + \frac{|D_{Low}|}{|D|}\text{Info}(D_{Low})$$
$$= \frac{2}{5}(-\frac{2}{2}\log_2\frac{2}{2} - \frac{0}{2}\log_2\frac{0}{2}) + \frac{1}{5}(-\frac{1}{1}\log_2\frac{1}{1} - \frac{0}{1}\log_2\frac{0}{1}) + \frac{2}{5}(-\frac{0}{2}\log_2\frac{0}{2} - \frac{2}{2}\log_2\frac{2}{2}) = 0$$

- **Step-5: Calculate information gain for each partitioned attribute**

  We calculate the value of Gain using Equation A.4.

$$\text{Gain}_A(D) = \text{Info}(D) - \text{Info}_A(D) \tag{A.4}$$

$$\text{Gain}_{CreditScore}(D) = \text{Info}(D) - \text{Info}_{CreditScore}(D) = 0.971 - 0.8 = 0.171$$
$$\text{Gain}_{IncomeRange}(D) = \text{Info}(D) - \text{Info}_{IncomeRange}(D) = 0.971 - 0 = 0.971$$

Here, the 'Income' has higher information gain than the 'Credit score' attribute to classify 'Buy home' in the data set.

A.2 Problemsets from Codeforces

Table A.4: List of problem sets from the Codeforces.

SL	Problem set	# of source code	SL	Problem set	# of source code	SL	Problem set	# of source code
1	1948E	2371	18	1801D	1985	35	1650G	2191
2	1942D	2269	19	1799D2	2178	36	1637E	1100
3	1934D1	2281	20	1790F	2936	37	1628D1	2827
4	1933F	2018	21	1787E	2061	38	1606E	2313
5	1924B	2189	22	1778D	2444	39	1605D	2373
6	1920E	2212	23	1775E	2877	40	1598E	2079
7	1919D	3235	24	1771D	2508	41	1593F	2205
8	1876C	2009	25	1767C	2386	42	1553E	2710
9	1884D	2633	26	1762D	2955	43	1547G	2473
10	1864E	2358	27	1761D	997	44	1538E	2162
11	1863E	2511	28	1748D	2475	45	1530E	2450
12	1856D	2859	29	1710B	2217	46	1511E	1973
13	1839D	1895	30	1684E	2861	47	1482E	2115
14	1838D	2868	31	1671E	2407	48	1411D	1870
15	1821E	2024	32	1667B	3366	49	1296F	2160
16	1810E	2842	33	1660F2	2403	50	1216F	1752
17	1808D	2079	34	1654D	1917			
Total # of source code = 116379								

Appendix B

APIs

B.1 APIs for accessing GitHub data

We have designed, developed and implemented the following APIs to facilitate the ETL process for GitHub:

- **git_info_wt_un_sc**:

  This API fetches the project information from the GHTorrent data. This data does not include user names and source code. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_info_wt_un_sc/
  ```

- **git_users_name**

  This is a supporting API to scrape user names from GitHub according to user logins. As mentioned earlier in Section 3.1.1, GitHub API has limited access, so we developed our own API. This API returns user name as a string.

  ```
  http://SERVER/git_users_name?users_login={users_login}
  ```

- **git_info**:

  This API fetches the complete extracted information from GitHub including user id, first name, user country code, project id, project language, gender, gender probability, source code. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_info/
  ```

- **git_features**:

  This API fetches GitHub info and calculated features including user id, first name, user country code, project id, project language, gender, gender probability, source code, lines of code (LOC), Blank line, Cyclomatic complexity (CC), Vocabulary, Program length, Logical executable lines of code (LLOC), Comments, Volume, Difficulty, Effort, Maintainability index (MI) and Quality as described in Section 2.2. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_features/
  ```

B.2 APIs for accessing Codeforces data

We have designed, developed, and implemented the following APIs to facilitate the ETL process of Codeforces:

- **cf_user_info**: This API fetches the user information according to the given 'handle'. User information includes first name, last name, country, rating, maxRating, rank, maxRank, gender, and gender probability. This API returns a response with data in {key:value} pairs.

  `http://SERVER/cf_user_info?handle=<user_handle>`

- **cf_problemset**: This API fetches the information from individual problem sets with source code. Information includes problem set, submission id, handle, language, time, memory, and source code. This API returns a response with data in {key:value} pairs.

  `http://SERVER/cf_problemset?problemset=<problemset>`

- **cf_info**: This API fetches the complete extracted information from Codeforces, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, and source code. This API returns a response with data in {key:value} pairs.

  `http://SERVER/cf_info/`

- **cf_features**: This API fetches Codeforces information and calculated features, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, Cyclomatic complexity, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  `http://SERVER/cf_features/`

B.3 Feature calculation API

generate_feature: This API generates 12 features of a source code, including LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

`http://SERVER/generate_feature?source_code={source_code}`

B.4 APIs for accessing balanced data set

B.4.1 Balanced ternary gender data set (Codeforces)

- **cf_balanced_gender**:

  This API fetches instances of the balanced gender data set of Codeforces information, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  `http://SERVER/cf_balanced_gender/`

B.4.2 Balanced binary gender data set (Codeforces)

- **cf_balanced_binary_gender**:

  This API fetches instances of the balanced binary gender data set of Codeforces information, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/cf_balanced_binary_gender/
  ```

B.4.3 Balanced ternary gender data Set (GitHub)

- **git_balanced_gender**:

  This API fetches instances of the balanced gender data set of GitHub information, including user id, first name, user country code, project id, project language, gender, gender probability, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_balanced_gender/
  ```

B.4.4 Balanced binary gender data set (GitHub)

- **git_balanced_binary_gender**:

  This API fetches instances of the balanced binary gender data set of GitHub information, including user id, first name, user country code, project id, project language, gender, gender probability, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_balanced_binary_gender/
  ```

B.4.5 Balanced binary region data Set (Codeforces)

- **cf_balanced_region**:

  This API fetches instances of the balanced region data set of Codeforces information, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/cf_balanced_region/
  ```

B.4.6 Balanced binary region data Set (GitHub)

- **git_balanced_region**:

  This API fetches instances of the balanced region data set of GitHub information, including user id, first name, user country code, project id, project language, gender, gender probability, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/git_balanced_region/
  ```

B.4.7 Balanced ternary expertise data Set (Codeforces)

- **cf_balanced_expertise**:

  This API fetches instances of the balanced expertise data set of Codeforces information, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/cf_balanced_expertise/
  ```

B.4.8 Balanced binary expertise data Set (Codeforces)

- **cf_balanced_binary_expertise**:

  This API fetches instances of the balanced binary expertise data set of Codeforces information, including problem set, submission id, handle, first name, country, rating, maxRating, rank, maxRank, gender, gender probability, time, memory, source code, LOC, Blank line, CC, Vocabulary, Program length, LLOC, Comments, Volume, Difficulty, Effort, MI and Quality. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/cf_balanced_binary_expertise/
  ```

B.5 API for accessing code transformation data set

- **code_transformation_data**:

  This API fetches instances of code transformation data in bad code and good code pair. This API returns a response with data in {key:value} pairs.

  ```
  http://SERVER/code_transformation_data/
  ```

Appendix C

Undersampling Approach

C.1 Balanced Gender (GitHub)

To construct a balanced gender data set from GitHub, we applied undersampling to address the class imbalance among the three gender labels: *male*, *female*, and *other*. The target was to match the minority class size—*female* programmers, with 27,176 instances—across all three classes.

As shown in Table 3.2, the *male* class initially contained a significantly higher number of instances. To reduce this number, we stratified the sampling process by region and code quality to preserve diversity across socioeconomic and software quality dimensions. From countries classified as having a *steady GDP*, we retained all available instances, since they were comparatively fewer. For countries under the *high GDP* category, we applied a fixed cap of 2,159 instances per country, selected in a balanced manner across the two code quality labels (*compliant* and *non-compliant*). This threshold was determined by identifying the maximum number of valid samples that could be included without exceeding the total limit for the class while maintaining an even distribution. Strike-through values in Table C.1 indicate the original counts before undersampling, while the retained numbers reflect the final selection for the balanced data set. In this manner, equal representation by geographies and quality classes was ensured without sampling bias.

Similarly, to balance the *other* gender class, we employed undersampling to reduce the class size from its original total of 78,990 instances to match the minority class size of 27,176 instances. As shown in Table C.2, we retained a stratified sample based on region and code quality to preserve diversity and avoid sampling bias. For countries in the *steady GDP* group, we preserved all instances, as their counts were already low and close to the target class size. In contrast, the *high GDP* countries such as the U.S. and China, had disproportionately large numbers of samples. We capped the number of retained instances per *high GDP* country to 4,376, ensuring a balanced representation between the *compliant* and *non-compliant* code classes within each country. The retained values reflect this balance, while the strike-through values denote the original counts before undersampling. Countries with already small populations such as India, Russia, and Japan, were left intact.

C.2 Balanced Region (GitHub)

To address class imbalance in the binary region data set, we applied an undersampling strategy to the overrepresented *high GDP* class. As shown in Table 3.3, the number of instances from *high GDP* countries substantially exceeded those from *steady GDP* countries. To construct a balanced data set, we retained all instances available from *steady GDP* countries while undersampling the *high GDP* category on a stratified scale. Specifically, we cut off the *high GDP* class to 8,474 samples, matching the size of the *steady GDP* group and having a final balanced data set of 16,948 samples.

Table C.3 details the distribution of the retained samples across gender and code quality within the *high GDP* class. From the original large pool of *male*, *female*, and *other*-gender program-

Table C.1: Programmers distribution for gender label ‘male’ over Region and Code Quality classes in GitHub.

			Code Quality		**Total**
			Compliant	**Non-compliant**	
Region	**high GDP**	U.S.	~~37373~~ 2159	~~46650~~ 2159	~~140078~~ 21590
		China	~~9394~~ 2159	~~15547~~ 2159	
		India	~~4116~~ 2159	~~9562~~ 2159	
		Russia	~~3293~~ 2159	~~4080~~ 2159	
		Japan	~~4378~~ 2159	~~5685~~ 2159	
	steady GDP	Argentina	643	611	~~5595~~ 5586
		Singapore	448	825	
		Vietnam	146	216	
		Bangladesh	528	~~1448~~ 1439	
		Egypt	218	512	
					~~145673~~ 27176

Table C.2: Programmers distribution for gender label 'other' over Region and Code Quality classes in GitHub.

			Code Quality		Total
			Compliant	**Non-compliant**	
Region	**high GDP**	U.S.	~~5881~~ 4376	~~6696~~ 4376	~~37545~~ 24398
		China	~~7169~~ 4376	~~10905~~ 4376	
		India	1029	2089	
		Russia	889	1112	
		Japan	730	1045	
	steady GDP	Argentina	214	466	~~2782~~ 2778
		Singapore	101	146	
		Vietnam	239	300	
		Bangladesh	~~206~~ 205	~~885~~ 882	
		Egypt	96	129	
					~~40327~~ 27176

mers (original totals are shown with a strikethrough), we retained 1,413 *compliant* and 1,413 *non-compliant* instances for each gender subgroup. This stratified retention gave the exact representation to both code quality labels and gender subgroups in the *high GDP* group.

Table C.3: Programmers distribution for region label 'high GDP' over Gender and Code Quality classes.

		Code Quality		**Total**
		Compliant	**Non-compliant**	
Gender	**male**	~~58554~~ 1413	~~81524~~ 1413	~~186426~~ 8474
	female	~~4318~~ 1409	~~4485~~ 1413	
	other	~~15698~~ 1413	~~21847~~ 1413	

C.3 Balanced Gender (Codeforces)

To address the class imbalance in the gender data set, we applied an undersampling strategy to the overrepresented gender classes, starting with the *male* programmers. Based on the distribution shown in Table C.4, we limited our selection to a maximum of 79 instances per cell (i.e., per unique combination of region, expertise, and code quality) for programmers from *high GDP* countries, and 154 instances per cell for those from *steady GDP* countries. This allowed us to achieve a more balanced representation without overfitting to a particular subgroup. For example, within the *high GDP* group, the U.S. had 11 *compliant* and 47 *non-compliant* beginner-level programs, all of which were retained because they were under the threshold. For intermediate-level programmers from the U.S., we sampled 79 *compliant* and 79 *non-compliant* instances from a larger pool of 148 *compliant* and 866 *non-compliant* programs, respectively. At the expert level, we included 44 *compliant* and 79 *non-compliant* instances out of 214 available *non-compliant* programs. A similar stratified selection was applied to other *high GDP* and *steady GDP* countries. By maintaining uniform quotas across regions, expertise, and quality, we ensured fair representation and diversity within the male subset. This procedure resulted in a total of 3,448 *male* instances: 1,719 from *high GDP* countries and 1,729 from *steady GDP* countries.

Similarly, after observing Table C.5, we decided to choose at most 81 instances per cell (i.e., for each unique combination of region, expertise, and code quality) from programmers belonging to *high GDP* countries, and at most 264 instances per cell from *steady GDP* countries for the *other* gender class label. For instance, from China (*high GDP* country), we sampled 81 instances for each expertise and quality combination where sufficient data were available. The same per-cell quota was applied to other *high GDP* countries such as the U.S., India, Russia, and Japan. For *steady GDP* countries like Vietnam and Bangladesh, we selected up to 264 instances per cell, particularly where high-volume data was present for *intermediate* programs. This sampling ensured a more equitable representation of the *other* gender class across regions and expertise levels. Following this strategy, we retained a total of 3,448 instances under the *other* label: 1,728 from *high GDP* countries and 1,720 from *steady GDP* countries.

Table C.4: Programmers distribution for gender label 'male' over region, expertise and code quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively.

			Expertise						**Total**
			Beginner		**Intermediate**		**Expert**		
Region	**high GDP**	U.S.	11	47	~~148~~ 79	~~866~~ 79	44	~~214~~ 79	~~15369~~ 1719
		China	~~106~~ 79	~~249~~ 79	~~710~~ 79	~~1975~~ 79	~~229~~ 79	~~764~~ 79	
		India	~~109~~ 79	~~667~~ 79	~~976~~ 79	~~5785~~ 79	23	~~181~~ 79	
		Russia	8	28	~~228~~ 79	~~1329~~ 79	49	~~456~~ 79	
		Japan	1	8	8	68	10	~~72~~ 71	
	steady GDP	Argentina	1	7	31	98	4	7	~~3884~~ 1729
		Singapore	0	1	9	~~68~~ 67	10	30	
		Vietnam	17	31	~~183~~ 154	~~625~~ 154	19	~~129~~ 128	
		Bangladesh	27	~~140~~ 139	~~305~~ 154	~~1278~~ 154	2	23	
		Egypt	32	~~105~~ 104	~~171~~ 154	~~513~~ 154	3	15	
			Com	**Ncom**	**Com**	**Ncom**	**Com**	**Ncom**	~~19253~~
			Code Quality						3448

Table C.5: Programmers distribution for gender label 'other' over region, expertise and code Quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively.

			Expertise						**Total**
			Beginner		**Intermediate**		**Expert**		
Region	**high GDP**	U.S.	8	30	81	~~412~~ 81	23	72	~~13306~~ 1728
		China	~~215~~ 81	~~577~~ 81	~~1823~~ 81	~~5170~~ 81	~~292~~ 81	~~858~~ 81	
		India	46	~~199~~ 81	~~245~~ 81	~~1381~~ 81	5	46	
		Russia	6	52	~~134~~ 81	~~597~~ 81	19	~~115~~ 81	
		Japan	4	18	~~54~~ 53	~~384~~ 81	50	~~390~~ 81	
	steady GDP	Argentina	0	2	16	43	9	11	~~2085~~ 1720
		Singapore	0	4	38	108	8	63	
		Vietnam	43	136	174	~~590~~ 264	6	72	
		Bangladesh	15	66	~~67~~ 66	~~302~~ 264	4	55	
		Egypt	16	23	46	168	0	0	
			Com	**Ncom**	**Com**	**Ncom**	**Com**	**Ncom**	~~15391~~
			Code Quality						3448

C.4 Balanced Region (Codeforces)

To alleviate the class imbalance in the region data set from Codeforces, we employed an undersampling procedure on the dominant class, specifically the *high GDP* countries. As one can observe in Table 3.3, samples from *high GDP* countries vastly outnumber those from countries with *steady GDP*. In order to get a balanced data set, we first retained all samples of the *steady GDP* class (6,274 instances). Subsequently, a stratified undersampling method was applied to the *high GDP* class, creating the same number of 6,274 instances. Deliberate undersampling was performed to maintain representative coverage across gender, expertise, and code quality. Specifically, as illustrated in Table C.6, we selected at most 376 samples for each cell (i.e., per unique combination of gender, expertise, and code quality). This sampling limit enabled us to maintain diversity while limiting the total to the desired number.

Table C.6: Programmers distribution for region label 'high GDP' over Gender, Expertise and Code Quality classes in Codeforces. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively.

		Expertise						**Total**
		Beginner		**Intermediate**		**Expert**		
Gender	**Male**	235	~~999~~ 376	~~2070~~ 376	~~10023~~ 376	355	~~1687~~ 376	~~15369~~ 2094
	Female	45	~~106~~ 104	300	~~1096~~ 1095	101	~~378~~ 376	~~2026~~ 2021
	Other	279	~~876~~ 376	~~2337~~ 376	~~7944~~ 376	~~389~~ 376	~~1481~~ 376	~~13306~~ 2159
		Com	**Ncom**	**Com**	**Ncom**	**Com**	**Ncom**	~~**30701**~~
		Code Quality						**6274**

C.5 Balanced Ternary Expertise (Codeforces)

To balance the unevenness of the Codeforces expertise data set, we addressed the excess of the *Intermediate* class. From Table 3.4, the *Beginner* and *Expert* classes had fewer instances compared to the *Intermediate*, though they were approximately equal in size. We began by removing 610 instances with the maxRank *Specialist*, leaving 8,782 instances for both the *Beginner* and *Expert* labels. Then, we did an undersampling strategy to restrict the *Intermediate* class to the same amount. As shown in Table C.7, we used stratified undersampling to restrict the samples of *Intermediate*, ensuring diversity by country, gender, and code quality. Specifically, for *high GDP* countries, we employed an upper bound of 215 instances. In contrast, for *steady GDP* countries, we employed a higher bound of 624 instances. The resulting balanced expertise data set contained 26,346 instances, with 8,782 instances per class.

C.6 Balanced Binary Expertise (Codeforces)

To address the imbalance between the binary expertise class labels in the Codeforces data set, we applied an undersampling strategy. As shown in Table C.8, the *Newbie* class—defined as programmers with ratings ranging from 0 to 1799—had 28,884 instances. In contrast, the *Skilled*

Table C.7: Programmers distribution for expertise label 'Intermediate' over Region, Gender and Code Quality classes. Here, Code Quality *compliant* and *non-compliant* are represented as Com and Ncom, respectively.

			Gender						**Total**
			Male		**Female**		**Other**		
Region	**high GDP**	U.S.	148	~~866~~ 215	14	38	81	~~412~~ 215	~~23770~~ 4395
		China	~~710~~ 215	~~1975~~ 215	210	~~718~~ 215	~~1823~~ 215	~~5170~~ 215	
		India	~~976~~ 215	~~5785~~ 215	38	~~165~~ 164	~~245~~ 215	~~1381~~ 215	
		Russia	~~228~~ 215	~~1329~~ 215	30	151	134	~~597~~ 215	
		Japan	8	68	8	24	54	~~384~~ 215	
	steady GDP	Argentina	31	98	0	0	16	43	~~5043~~ 4387
		Singapore	9	68	0	12	38	108	
		Vietnam	183	~~625~~ 623	36	75	174	590	
		Bangladesh	305	~~1278~~ 624	5	32	67	302	
		Egypt	171	513	14	36	46	168	
			Com	**Ncom**	**Com**	**Ncom**	**Com**	**Ncom**	~~28813~~
			Code Quality						8782

class—comprising programmers with ratings of 1800 and above—originally had 45,513 instances. For the purpose of creating a balanced data set, we retained all 28,884 instances from the *Newbie* class and applied class-wise undersampling to the *Skilled* class. Specifically, we reduced the number of instances in each subcategory (e.g., Candidate Master, Master, Grandmaster) of the *Skilled* group to maintain diversity while matching the *Newbie* count. The final balanced data set contained 28,884 instances in each class, totalling 57,768 instances.

Table C.8: Number of instances of binary expertise class from Codeforces data set.

Binary Expertise (Range of rating)	**maxRank (Codeforces feature)**	**Number of Instances (Codeforces)**	**Total**
Newbie (0∼1799)	Newbie	2160	28884
	Pupil	1971	
	Apprentice	0	
	Specialist	5261	
	Expert	19492	
Skilled (1800∼3000+)	Candidate Master	~~16332~~ 10178	~~45513~~ 28884
	Master	~~16482~~ 10227	
	International Master	~~3917~~ 2487	
	Grandmaster	~~4634~~ 3134	
	International Grandmaster	~~3209~~ 2147	
	Legendary Grandmaster	~~939~~ 711	
Total # of instances			**~~74397~~ 57768**

Appendix D

Sample programs in Data sets

```

// Copyright (c) 2011 The Chromium Authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can
    be
// found in the LICENSE file.
#include "stdafx.h"
#include "lock_impl.h"

namespace base {
namespace internal {

LockImpl::LockImpl() {
  // The second parameter is the spin count, for short-held locks it
    avoid the
  // contending thread from going to sleep which helps performance
    greatly.
  ::InitializeCriticalSectionAndSpinCount(&native_handle_, 2000);
}

LockImpl::~LockImpl() {
  ::DeleteCriticalSection(&native_handle_);
}

bool LockImpl::Try() {
  if (::TryEnterCriticalSection(&native_handle_) != FALSE) {
    return true;
  }
  return false;
}

void LockImpl::Lock() {
  ::EnterCriticalSection(&native_handle_);
}

void LockImpl::Unlock() {
  ::LeaveCriticalSection(&native_handle_);
}

}  // namespace internal
}  // namespace base
```

Listing D.1: GitHub Code Sample.

```
#include <iostream>
#include <fstream>

using namespace std;

void printreverse (ifstream &myfile, string revWord, int count)
{
  if(count >0)
  {
    myfile>>revWord;
    printreverse(myfile, revWord, count-1);
  }
  else return;
  cout<<revWord<<endl;
}

int main(int argc, char* argv[])
{
  ifstream file(argv[1]);
  if (file.fail())
  {
    cout<<"Error opening file"<<endl;
    return 1;
  }
  if (argc == 2)
  {
    int numberwords = 0;
    string word;
    file >> numberwords;
    int count = numberwords;
    printreverse(file, word, count);
  }
  else return 1;
}
```

Listing D.2: GitHub Code Sample.

```cpp
#include <bits/stdc++.h>

using namespace std;

const int DIM = 2e5 + 7;
const int INF = 1e9;

#define int long long

int urm[DIM];
int dp[DIM];

main()
{
  int n, k;
  cin >> n >> k;

  string s;
  cin >> s;

  s = ' ' + s;

  urm[n + 1] = INF;

  for(int i = n; i >= 1; i--)
  {
    if(s[i] == '1')
      urm[i] = i;
    else
      urm[i] = urm[i + 1];

  }

  dp[0] = 0;

  for(int i = 1; i <= n; i++)
  {
    dp[i] = dp[i - 1] + i;

    int c = urm[max(i - k, 1LL)];

    if(c <= i + k)
    {
      dp[i] = min(dp[i], dp[max(1LL, c - k) - 1] + c);
    }
  }

  cout << dp[n];
}
```

Listing D.3: Codeforces Code Sample.