

Data Analysis and Code Assessment Using Machine Learning Techniques for Programming Activities

Md. Mostafizer Rahman

A DISSERTATION
SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY
IN COMPUTER SCIENCE AND ENGINEERING

Graduate Department of Computer and Information Systems
The University of Aizu, Japan

September 2022



© Copyright by Md. Mostafizer Rahman, September 2022

All Rights Reserved.

The thesis titled

*Data Analysis and Code Assessment Using Machine Learning Techniques for
Programming Activities*

by

Md. Mostafizer Rahman

is reviewed and approved by:

Chief Referee


Senior Associate Professor

Yutaka Watanobe

Yutaka Watanobe Aug. 4, 2022 

Professor

Incheon Paik

Incheon Paik Aug. 4, 2022 


Senior Associate Professor

Mohamed Hamada

M. Hamada August 4, 2022 

Associate Professor

Keita Nakamura

Keita Nakamura August 4, 2022 

THE UNIVERSITY OF AIZU

September 2022

Contents

| | |
|---|-------------|
| List of Abbreviations | xi |
| List of Symbols | xiv |
| Acknowledgments | xvi |
| Abstract | xvii |
| Chapter 1 Introduction | 1 |
| 1.1 Overview | 1 |
| 1.1.1 Online Judge System | 2 |
| 1.1.2 Programming Learning with Online Judge System | 3 |
| 1.1.3 Programming Data Analysis | 4 |
| 1.1.4 Machine Learning in Programming | 7 |
| 1.1.4.1 RNN-based Code Assessment Models | 9 |
| 1.1.4.2 Transformer-based Code Assessment Models | 10 |
| 1.2 Scope and Motivation of the Study | 12 |
| 1.3 Dissertation Contributions | 13 |
| 1.3.1 Chapter 2: A Comprehensive Data-driven Analysis to Explore the Impact of Programming in Education | 14 |
| 1.3.2 Chapter 3: Code Assessment and Classification Using Attention-Based LSTM Neural Network | 15 |
| 1.3.3 Chapter 4: Code Assessment and Classification Using Bidirectional LSTM | 16 |
| 1.4 Dissertation Outline | 17 |
| 1.5 Publications | 18 |
| Chapter 2 A Comprehensive Data-driven Analysis to Explore the Impact of Programming in Education | 21 |
| 2.1 Introduction | 21 |
| 2.2 Background and Related Works | 24 |
| 2.2.1 Online Programming Learning Platform | 24 |
| 2.2.2 Supervised and Unsupervised Learning Algorithms | 26 |
| 2.2.3 Association Rule Mining Algorithms | 28 |
| 2.2.4 Rule-based Recommendation Systems | 28 |
| 2.2.5 Educational Data Mining and Learning Analytics | 29 |
| 2.3 Dataset and Preprocessing | 30 |
| 2.3.1 Aizu Online Judge System | 30 |
| 2.3.2 Solution Submission logs | 30 |
| 2.3.3 Class Performance Scores | 32 |
| 2.4 Approach | 35 |
| 2.4.1 Elbow Method | 36 |

| | | |
|---------|---|----|
| 2.4.2 | Modified K-means Clustering Algorithm | 36 |
| 2.4.2.1 | Multidimensional Data Clustering in Euclidean Space | 38 |
| 2.4.3 | FP-growth Algorithm | 40 |
| 2.5 | Experimental Results | 41 |
| 2.5.1 | Clustering the Data | 41 |
| 2.5.2 | Extracting Hidden Features | 42 |
| 2.5.3 | Discovering Frequent Data Patterns for Clusters | 48 |
| 2.5.4 | Association Rule Mining | 52 |
| 2.5.5 | Accumulation of Correlated Features | 54 |
| 2.6 | Discussion | 56 |
| 2.6.1 | Analysis and Recommendations | 57 |
| 2.6.2 | Pattern and Association Rule Mining | 60 |
| 2.6.3 | Learning and Teaching Strategies for Programming | 61 |
| 2.6.4 | Overall Assessments and Practical Applications | 61 |
| 2.6.5 | Limitations | 63 |
| 2.7 | Summary | 63 |

Chapter 3 Code Assessment and Classification Using Attention-Based LSTM Neural Network 65

| | | |
|-------|--|----|
| 3.1 | Introduction | 65 |
| 3.2 | Background and Related Works | 68 |
| 3.3 | Overview of Language Model and Recurrent Neural Networks | 71 |
| 3.3.1 | <i>N</i> -gram Language Model | 71 |
| 3.3.2 | Recurrent Neural Networks | 72 |
| 3.3.3 | Gradient Vanishing and Exploding | 74 |
| 3.3.4 | Long Short-Term Memory Neural Network | 75 |
| 3.4 | Approach | 76 |
| 3.4.1 | Proposed LSTM-AM Neural Network Architecture | 77 |
| 3.5 | Data Collection and Problem Description | 79 |
| 3.5.1 | Data Preprocessing and Training | 80 |
| 3.6 | Evaluation Metrics | 83 |
| 3.7 | Experimental Results | 84 |
| 3.7.1 | Hyperparameters | 84 |
| 3.7.2 | Selection of Hidden Units and Cross-entropy Measurement | 85 |
| 3.7.3 | Error Detection and Correction Word Prediction | 88 |
| 3.7.4 | Classification of Source Codes | 91 |
| 3.8 | Discussion | 92 |
| 3.9 | Summary | 95 |

Chapter 4 Code Assessment and Classification Using Bidirectional LSTM 97

| | | |
|---------|---|-----|
| 4.1 | Introduction | 97 |
| 4.2 | Background and Related Works | 101 |
| 4.3 | Proposed BiLSTM Model for Code Assessment and Classification | 104 |
| 4.3.1 | BiLSTM Model Architecture | 104 |
| 4.3.2 | Dataset and Experimental Setup | 107 |
| 4.3.3 | Evaluation Metrics for Code Assessment | 108 |
| 4.3.4 | Experimental Results | 109 |
| 4.3.4.1 | Selection of the Number of Epochs and Hidden Units | 109 |
| 4.3.4.2 | Erroneous Source Code Assessment | 111 |
| 4.3.4.3 | Suggestions for Code Repair | 112 |
| 4.3.4.4 | Overall Model Performance for Error Detection and Binary Classification | 113 |

| | | |
|---|---|------------|
| 4.4 | Proposed Stacked BiLSTM Model for Multilingual Source Code Classification | 113 |
| 4.4.1 | Stacked BiLSTM Model Architecture | 114 |
| 4.4.2 | Dataset and Preprocessing | 114 |
| 4.4.3 | Hyperparameters | 116 |
| 4.4.4 | Activation Functions | 117 |
| 4.4.5 | Evaluation Metrics | 118 |
| 4.4.6 | Experimental Results | 119 |
| 4.5 | Summary | 121 |
| Chapter 5 Conclusion and Future Research | | 124 |
| 5.1 | Conclusion | 124 |
| 5.2 | Future Research | 126 |

List of Figures

| | | |
|-------------|---|----|
| Figure 1.1 | An overview of unsupervised clustering process | 4 |
| Figure 1.2 | A simple IF-THEN relationship for ARM approach | 6 |
| Figure 1.3 | Evolution of the ANN | 7 |
| Figure 1.4 | The AI landscape and the application of DNN in programming | 8 |
| Figure 1.5 | An example of (a) word embedding and encoding process, (b) LSTM model training, and (c) token prediction by the trained LSTM model | 9 |
| Figure 1.6 | A basic architecture of the transformer model | 10 |
| Figure 1.7 | Outline of the Dissertation | 17 |
| Figure 2.1 | The overall framework of the data-driven approach | 36 |
| Figure 2.2 | Elbow method for optimal k selection | 37 |
| Figure 2.3 | An example of two-dimensional data distribution in Euclidean space | 40 |
| Figure 2.4 | 2D visualization of the clusters | 42 |
| Figure 2.5 | Segmentation of error verdicts received by the students | 43 |
| Figure 2.6 | Comparison of scores in different tests | 45 |
| Figure 2.7 | Tendency to submit assignments within the allotted period (08 days) | 46 |
| Figure 2.8 | Comparison of topic-wise accepted (AC) rate | 47 |
| Figure 2.9 | Tendency to submit assignments on the last day | 48 |
| Figure 2.10 | Overview of the frequency of attributes in different clusters | 50 |
| Figure 2.11 | Overview of attribute ranking using frequencies for each cluster | 51 |
| Figure 2.12 | Overall ranking-trend of the attributes based on frequencies of clusters | 51 |
| Figure 2.13 | A number of generated frequent patterns based on different $minSup$ values | 52 |
| Figure 2.14 | Overview of association rules generated using different $minSup$ (500, 1000, 1500, 2000, 2500, 3000) and $minConf$ (50%, 60%, 70%, 80%, 90%, 100%) values | 55 |
| Figure 2.15 | A summary graph of the main features | 58 |
| Figure 3.1 | A simple RNN structure | 73 |
| Figure 3.2 | Internal structure of an LSTM unit | 75 |
| Figure 3.3 | The main workflow of our model: (a) conversion of source code to token IDs, (b) model training using token IDs, and (c) results produced by the softmax function. | 77 |
| Figure 3.4 | An architecture of the proposed LSTM-AM network | 78 |
| Figure 3.5 | Input and output sample of GCD and PN problems | 80 |
| Figure 3.6 | The flowchart of the training and evaluation process of the proposed model | 80 |
| Figure 3.7 | Source code word embedding and tokenization process | 81 |
| Figure 3.8 | Training process of an LSTM language model | 82 |
| Figure 3.9 | LSTM-AM network prediction process | 83 |
| Figure 3.10 | Epoch-wise cross-entropies of 200-unit model using a) IS, b) PN, and c) GCD source codes | 87 |
| Figure 3.11 | Cross-entropies of the 200-unit model using IS, GCD, and PN problems | 88 |
| Figure 3.12 | Erroneous source code evaluated by the standard LSTM | 88 |
| Figure 3.13 | Erroneous source code evaluated by the LSTM-AM | 89 |

| | | |
|-------------|---|-----|
| Figure 3.14 | Erroneous source code evaluated by the LSTM | 90 |
| Figure 3.15 | Erroneous source code evaluated by the LSTM-AM | 91 |
| Figure 4.1 | Workflow of the proposed BiLSTM model | 104 |
| Figure 4.2 | Architecture of the BiLSTM neural network | 106 |
| Figure 4.3 | Typical input and output prototype of the proposed BiLSTM model | 106 |
| Figure 4.4 | Effect of cross-entropy on selecting epochs and hidden units for BiLSTM. (a) GCD problem set, (b) IS problem set. | 109 |
| Figure 4.5 | Effect of cross-entropy on selecting epochs and hidden units for LSTM. (a) GCD problem set, (b) IS problem set. | 110 |
| Figure 4.6 | Source codes evaluation by the LSTM model. (a) Erroneous solution code of GCD problem, (b) Erroneous solution of IS problem. | 111 |
| Figure 4.7 | Source codes evaluation by the BiLSTM model. (a) Erroneous solution code of GCD problem, (b) Erroneous solution of IS problem. | 112 |
| Figure 4.8 | Architectural overview of the stacked BiLSTM model | 114 |
| Figure 4.9 | Statistical overview of the programming languages based on solution codes | 116 |
| Figure 4.10 | Tokenization and encoding | 116 |
| Figure 4.11 | Tanh and ReLU activation functions | 118 |
| Figure 4.12 | Accuracy and loss per epoch during training and validation for the LSTM model | 119 |
| Figure 4.13 | Accuracy and loss per epoch during training and validation for the BiL- STM model | 120 |
| Figure 4.14 | Accuracy and loss per epoch during training and validation for the stacked BiLSTM model | 120 |
| Figure 4.15 | Confusion matrix for all classes using the stacked BiLSTM model | 122 |

List of Tables

| | | |
|------------|---|----|
| Table 2.1 | Topic-wise problem list of ALDS1 course | 31 |
| Table 2.2 | Sample submission logs generated by the AOJ system | 32 |
| Table 2.3 | Sample data distribution of student evaluations | 33 |
| Table 2.4 | Sample operational data distributions by joining submission logs (Table 2.2) and evaluation scores (Table 2.3) | 35 |
| Table 2.5 | An example of 4-dimensional dataset | 38 |
| Table 2.6 | Preliminary statistical information of each cluster | 42 |
| Table 2.7 | Overview of the judge verdicts of ALDS1 course | 43 |
| Table 2.8 | Overview of the submission statistics for each type of problem | 43 |
| Table 2.9 | Cluster-wise solution accuracy and problem-solving $T&E$ | 44 |
| Table 2.10 | Overview of the average scores and standard deviation (σ) in each cluster | 44 |
| Table 2.11 | Statistics of extra problem solutions | 45 |
| Table 2.12 | Topic-wise average accepted (AC) solution rate | 46 |
| Table 2.13 | Average submission rate on the last day | 47 |
| Table 2.14 | Repetition tendency of accepted problems | 48 |
| Table 2.15 | Dictionary for the set attributes | 49 |
| Table 2.16 | Association rules for the students of cluster P | 52 |
| Table 2.17 | Association rules for the students of cluster Q | 53 |
| Table 2.18 | Association rules for the students of cluster R | 53 |
| Table 2.19 | Association rules for the students of cluster S | 54 |
| Table 2.20 | List of the main features | 57 |
| | | |
| Table 3.1 | A partial list of Ids for characters, special characters, numbers, keywords | 81 |
| Table 3.2 | Number of source codes of each problem | 85 |
| Table 3.3 | Cross-entropy comparison using PN source codes | 86 |
| Table 3.4 | Cross-entropy comparison using GCD source codes | 86 |
| Table 3.5 | Cross-entropy comparison using IS source codes | 86 |
| Table 3.6 | Cross-entropy comparison of different models using all source codes | 87 |
| Table 3.7 | List of detected errors and predicted words in Figure 3.12 by the LSTM model | 89 |
| Table 3.8 | List of detected errors and predicted words in Figure 3.13 by the LSTM-AM model | 90 |
| Table 3.9 | List of detected errors and predicted words in Figure 3.14 by the LSTM model | 90 |
| Table 3.10 | List of detected errors and predicted words in Figure 3.15 by the LSTM-AM model | 91 |
| Table 3.11 | Assessment results of syntax errors (CE) detection for erroneous source code | 91 |
| Table 3.12 | Assessment results of logic errors (WA , TLE , MLE , PrE , and RTE) detection for erroneous source code | 92 |
| Table 3.13 | Classification performance comparison using Insertion Sort (IS) source codes | 93 |
| Table 3.14 | Classification performance comparison using Greatest Common Divisor (GCD) source codes | 93 |

| | | |
|------------|---|-----|
| Table 3.15 | Classification performance comparison using Prime numbers (PN) source codes | 93 |
| Table 3.16 | The evaluation results based on Figure 3.12 using standard LSTM | 94 |
| Table 3.17 | The evaluation results based on Figure 3.13 using LSTM-AM | 94 |
| Table 3.18 | Evaluation results by the standard LSTM and LSTM-AM | 95 |
| Table 3.19 | Overview of the average evaluation statistical results using various erroneous source codes | 95 |
| | | |
| Table 4.1 | Comparative lowest crossentropy of LSTM and BiLSTM models for different hidden units | 110 |
| Table 4.2 | Suggestions for GCD problem evaluated in Figures 4.6(a) and 4.7(a) | 112 |
| Table 4.3 | Suggestions for IS problem evaluated in Figures 4.6(b) and 4.7(b) | 112 |
| Table 4.4 | Performance of the models based on the GCD dataset | 113 |
| Table 4.5 | Performance of the models based on the IS dataset | 113 |
| Table 4.6 | A summary of solution codes with problem title/class name | 115 |
| Table 4.7 | List of the hyperparameters and settings used in the proposed model | 117 |
| Table 4.8 | Average precision, recall, and F1-score of all models | 120 |
| Table 4.9 | Average accuracy of the models using ReLU activation function | 121 |
| Table 4.10 | Average accuracy of the models using Tanh activation function | 121 |

List of Abbreviations

| | |
|-----------|--|
| ICT | Information and Communication Technology |
| OJ | Online Judge |
| FP-growth | Frequent Pattern-growth |
| LSTM | Long Short-Term Memory |
| BiLSTM | Bidirectional Long Short-Term Memory |
| RNNs | Recurrent Neural Networks |
| MPLs | Multi-Programming Languages |
| AOJ | Aizu Online Judge |
| IT | Information Technology |
| TAL | Technology-Assisted Learning |
| SPAs | Smart Personal Assistants |
| AI | Artificial Intelligence |
| CE | Compile Error |
| RTE | Runtime Error |
| TLE | Time Limit Exceeded |
| MLE | Memory Limit Exceeded |
| OLE | Output Limit Exceeded |
| PrE | Presentation Error |
| WA | Wrong Answer |
| AC | Accepted |
| ICPC | International Collegiate Programming Contest |
| UVa | University of Valladolid |
| URI | Universidade RegionalIntegrada |
| SVM | Support Vector Machine |
| KNN | K-Nearest Neighbors |
| PAM | Partitioning Around Medoids |
| CLARA | Clustering Large Applications |
| BIRCH | Balanced Iterative Reducing and Clustering using Hierarchies |
| CURE | Clustering Using REpresentatives |
| FCM | Fuzzy C-Means |
| FCS | Fuzzy C-Shells |
| DBCLASD | Distribution-Based Clustering of LARge Spatial Databases |

| | |
|-----------|---|
| GMM | Gaussian Mixture Model |
| DBSCAN | Density-Based Spatial Clustering of Applications with Noise |
| OPTICS | Ordering Points to Identify the Clustering Structure |
| CLICK | CLuster Identification via Connectivity Kernels |
| STING | STatistical Information Grid |
| CLIQUE | CLustering in QUEst |
| COBWEB | Complexity and Organized Behaviour Within Environmental Bounds |
| ARM | Association Rule Mining |
| RS | Recommending System |
| NN | Neural Network |
| ANN | Artificial Neural Networks |
| FNN | Feed Forward Neural Network |
| DNN | Deep Neural Network |
| Tanh | Hyperbolic Tangent |
| ReLU | Rectified Linear Unit |
| CNNs | Convolutional Neural Networks |
| GPU | Graphics Processing Unit |
| CPU | Central Processing Unit |
| ML | Machine Learning |
| ALDS | Algorithms and Data Structures |
| MK-means | Modified K-means |
| ICSM | Initial Center Selection Module |
| ODM | Outlier Detection Module |
| APA | Automated Program Assessment |
| MOOC | Massive Open Online Course |
| EDM | Educational Data Mining |
| LA | Learning Analytics |
| PAGE | Personalized Adaptive Gamified E-learning |
| AATs | Automated Assessment Tools |
| SL | Supervised Learning |
| USL | Unsupervised Learning |
| CLARANS | Clustering Large Applications based on RANdomized Search |
| CLARA | Clustering Large Applications |
| CFSFDP-HD | Clustering by Fast Search and Finding of Density Peaks via Heat Diffusion |
| Eclat | Equivalence CLASS Transformation |
| GBM | Gradient Boosting Machine |
| LMS | Learning Management System |
| T&E | Trial and Error |
| SSE | Sum of Squared Errors |

| | |
|---------------|---|
| PA | Programming Assignment |
| CoT | Coding Test |
| PT | Paper-based Test |
| PCS | Plagiarism Checking Software |
| PCA | Principal Component Analysis |
| SE | Software Engineering |
| NLP | Natural Language Processing |
| LSTM-AM | Long Short-Term Memory with Attention Mechanism |
| Seq2Seq | Sequence to Sequence |
| NB | Naive Bayes |
| AST | Abstract Syntax Tree |
| AUC | Area Under the Curve |
| ROC | Receiver Operating Characteristics |
| GCD | Greatest Common Divisor |
| IS | Insertion Sort |
| PN | Prime Numbers |
| RF | Random Forest |
| RBM | Restricted Boltzmann Machine |
| DBN | Deep Belief Network |
| OBEnvironment | Object Behavior Environment |
| CASE | Computer Aided Software Engineering |
| PDG | Program Dependency Graph |
| NLTK | Natural Language Toolkit |
| OOV | Out Of Vocabulary |

List of Symbols

| | |
|--------------|---|
| V | Online Judge Verdicts for Solution Code |
| J | Objective Function |
| G, H | Vectors of Attributes |
| G_i | Component of G Vector |
| H_i | Component of H Vector |
| Sup | Support |
| Con | Confidence |
| X | Input of Neural Network (NN) |
| W | Connecting weight of each input |
| σ | Non-linearity functions (i.e., sigmoid, Tanh, ReLU, etc.) of NN |
| Y | Output of NN |
| O_{logs} | A sample output of an Online Judge system |
| $CVal$ | Code Validation Score |
| T_{score} | Test Scores |
| T | Theory Scores |
| $Prac$ | Practical Scores |
| FS | Final Scores |
| \mathbb{N} | Natural Numbers |
| \mathbb{R} | Real Numbers |
| D | Distance |
| K | Number of clusters |
| M_k | Mean of k cluster |
| MMA | Min-Max Average |
| \subseteq | Subset |
| ϕ | Null Set |
| \cap | Intersection |
| W | Set of Attributes |
| \prod | Series of Products |
| \sum | Series of Summation |
| \approx | Approximately Equal |
| M_t | Output of previous hidden state |
| α | Attention Weight |

| | |
|-----------|----------------------------------|
| C_t | Context Vector |
| G_t | Output Vector |
| P | Precision |
| R | Recall |
| h_t^f | Outcome of forward hidden state |
| h_t^b | Outcome of backward hidden state |
| $minSup$ | Minimum Support |
| $minConf$ | Minimum Confidence |

Acknowledgments

At first, I am grateful to the Almighty, who is the source of all strength and wisdom.

I would like to express my sincere gratitude and respect to my supervisor, Professor Yutaka Watanobe, for his thoughtful guidance, patience, encouragement, advice, and continuous support during my Ph.D. studies. I was very fortunate to have a supervisor who cared so much about my daily life and my research, listened to everything from my side and responded to my questions and concerns so promptly and humbly. Over the past few years, I have received a lot of support and guidance from Professor Watanobe, both in my research and daily life. Professor Watanobe helped me a lot in the last few years to become a good researcher and without his support, I would not have been able to finish my research. I am deeply inspired by Professor Watanobe for his broad vision, rigorous academic attitude, and way of working for excellence.

I would also like to express gratitude to rest of my dissertation review committee members, Professor Incheon Paik, Professor Mohamed Hamada, and Professor Keita Nakamura, for their constructive comments, ideas, and suggestions for my dissertation.

Additionally, I would also like to thank the Japan Student Services Organization (JASSO) and the Support Association for International Students of the University of Aizu (SAISUA) for their financial support. Most importantly, I would like to thank all of my respected teachers, friends, and colleagues who have helped me enrich my academic goals and social life over the past three years.

Finally, I would like to express my heartfelt gratitude to my parents, wife, daughter, younger brother, grand mother, parents-in-law, and relatives for their encouragement, understanding, support, trust, prayers, and unconditional love for me over the past few years. I am eternally indebted to all of them. I love you all!

Abstract

In recent years, the development of information and communication technology (ICT)-based tools has facilitated human work and increased productivity in solving complex tasks. Computer programming has become an indispensable skill in ICT development because of its wide range of applications. At the same time, meeting the growing demand of highly skilled programmers in the ICT sector is one of the biggest challenges. However, learning programming is not an easy and trivial task, because programming skills are essentially acquired through repeated practice. Here online judge (OJ) systems provide uninterrupted programming learning and practice opportunities in addition to classroom-based learning. Thus, OJ systems have been adopted by many institutions as an academic tool for programming education, and as a result, a huge number of programming-related resources (source codes, logs, scores, activities, etc.) are regularly accumulated in OJ systems. In this dissertation, we leveraged a large number of real-world source codes, submission logs and scores collected from an OJ system for comprehensive data analysis, as well as training, validation, testing and experimentation with machine learning models for code assessment and classification. First, we analyzed the different features and programming-related problems using a real dataset. We identified various programming errors, including time limit exceeded, memory limit exceeded, runtime error, and presentation errors in solution codes, as well as the impact of programming skills on academic performance. Next, we developed machine learning-based source code assessment and classification models to better understand the programming code and reduce errors. Finally, the outcome of the dissertation can assist programmers to understand and improve their programming skills.

In *Chapter 2*, a comprehensive data analysis framework is proposed to extract hidden features and association rules using a real-world dataset of an OJ system. Initially, an unsupervised modified K-means (MK-means) clustering algorithm is applied for data clustering, and then the frequent pattern (FP)-growth algorithm is used for association rule mining. We leverage students' program submission logs and academic scores as an experimental dataset. To explore

the correlation between programming skills and overall academic performance, the statistical features of students are analyzed and the related results are presented including hidden features, common errors made by students, submission trends, frequent patterns, association rules, and so on. A number of useful recommendations are provided for students in each cluster based on the identified hidden features. In addition, the analytical results of this Chapter can help teachers prepare effective lesson plans, evaluate programs with special arrangements, and identify the academic weaknesses of students. Furthermore, a prototype of the proposed approach and data-driven analytical results can be applied to other practical courses in ICT or engineering disciplines. Based on the data analysis, we identified most common errors made by programmers during their learning processes. We found that many of the errors encountered could not be evaluated or detected by conventional compilers. Moreover, it is difficult to assess and detect logic errors (e.g., time limit exceeded, run time error, memory limit exceeded, output limit exceeded, etc.) in the source code with traditional compilers, resulting in erroneous code.

In *Chapter 3*, we proposed a source code assessment and classification model. The proposed model is developed based on a long short-term memory (LSTM) neural network with an attention mechanism to assess and classify the source code. The attention mechanism enhances the accuracy of the proposed model for assessment and classification. Thus, the proposed model can detect source code errors with locations and then predict the correct word for error. In addition, the proposed model can classify the source codes whether it is erroneous or not. We trained the model using source codes and then evaluated the performance. The experimental results obtained show that the accuracy in terms of error detection and prediction of the proposed model approximately is 62% and source code classification (correct or incorrect) accuracy approximately is 96% that outperformed a standard LSTM and other state-of-the-art models. Overall, these experimental results indicate the usefulness of the proposed model in professional programming and programming education fields. Furthermore, the proposed model can help programmers to reduce errors in solution codes that cannot be detected by conventional compilers. Despite the good performance of LSTM-based model, it still has a shortcoming that it only considers the past context of the input sequences, but cannot consider any future (i.e., subsequent) context.

In *Chapter 4*, we proposed a sequential language model for evaluating source codes using a bidirectional long short-term memory (BiLSTM) neural network. The BiLSTM model can consider both the past and future context of the input sequences. We trained the BiLSTM model

with a large number of real-world source codes with tuning various hyperparameters. We then used the model to evaluate incorrect code and assessed the model's performance in three principal areas: source code error detection, suggestions for incorrect code repair, and erroneous code classification. Experimental results showed that the proposed BiLSTM model achieved significant correctness in identifying errors and providing suggestions. Moreover, the model achieved an F-score of approximately 97%, outperforming other state-of-the-art models such as recurrent neural networks (RNNs) and LSTM. Furthermore, programmers have recently improved their programming skills and can now write code in many different languages to solve problems. A lot of new code is being generated all over the world regularly. Since a programming problem can be solved in many different languages, it is quite difficult to identify the algorithm from the written source code. Therefore, a classification model is needed to help programmers identify the algorithms in source code written/developed in Multi-Programming Languages (MPLs). The classification model can help programmers learn better programming. However, source code multi-class classification models based on deep learning are still lacking in the field of programming education and software engineering. To address this gap, we also proposed a multilingual source code classification model using stacked BiLSTM. To accomplish this task, we collect a large number of source codes from the Aizu Online Judge (AOJ) system. The stacked BiLSTM model is trained, validated, and tested on the real-world dataset. Various hyperparameters are fine-tuned to improve the performance of the model. Based on the experimental results, the stacked BiLSTM model achieved an accuracy of about 93% and an F1-score of 89.24%. Moreover, the model outperforms the state-of-the-art models in terms of other evaluation matrices such as precision (90.12%) and recall (89.48%).

Chapter 1

Introduction

1.1 Overview

The demand for information technology (IT)-based tools and services is evolving and changing day by day. Accordingly, the development and production of technology is an essential part of this evolving world, in which computer programming plays an important role. So, the information and communication technology (ICT) industry needs highly qualified programmers to develop new technologies. Also, computer programming is one of the fundamental subjects in ICT discipline. The conventional learning process of computer programming is insufficient to prepare highly skilled programmers due to the limited number of exercise classes, limited practice opportunities, and lack of individual tutoring. In addition, most educational institutions, such as schools, colleges, and universities are struggling to build more educational facilities to increase academic activity (e.g., additional exercise classes, practice, and individual tutoring) due to financial, logistical, and organizational constraints [1].

The growing number of people in classrooms in educational institutions, the large number of students per class, and some lectures are conducted with more than a thousand participants in the massive open online courses (MOOCs)¹ which complicate the individual tutoring process [2]. Furthermore, the growing ratio between students and educators raises the question of how to provide individual support to students to improve their problem-solving skills. Especially, when learning computer programming, students need a lot of practice and individual tutoring to improve their programming knowledge and skills. Although, programming practice and competition can play an important role in acquiring good programming skills [3].

¹<https://www.mooc.org/>

Recently, a new technology known as smart personal assistants (SPAs) has become available to end-users. Some examples of SPAs are Siri (Apple)², Alexa (Amazon)³, and Assistant (Google)⁴. These technologies are highly intelligent and interactive, which can be useful for learning [4]. In addition, SPA technology will be powered by more than 870 million devices by 2022 [5]. An SPA is an application of artificial intelligence (AI) that provides assistance (e.g., answering questions, recommendations, executing actions, suggestions, etc.) based on user input (e.g., voice, images, and other types of information) [6]. These systems are hosted by big technology giants to receive voice or text data and produce the relevant output [7]. Despite the many advantages of SPA systems for quick answers, they are difficult to use for the purpose of programming learning and evaluation.

However, research in technology-assisted learning (TAL) has focused on addressing these challenges by exploiting the potential of IT. Over the past 40 years, educational researchers have examined the effects of IT on learning outcomes [8]. In the last 5 to 15 years, researchers have been able to demonstrate the importance of TAL systems which are more effective than non-personal tutoring conditions [9–11]. Many TAL systems assist students at every step of the problem-solving session rather than just evaluating the final answer. These systems follow a variety of scaffolding strategies to support students' learning processes [12]. Scaffolding strategies transform learning activities into smaller modules, maximizing the use of tools and structures to support students to gain more knowledge [13]. Scaffolding strategies can be of two types, namely, dynamic and static [14]. In dynamic scaffolding strategies, TAL systems continuously analyze student activity and provide the necessary support based on students' problems and responses. In contrast, static scaffolding strategies provide static support to students based on the analysis of students' previous difficulties and responses.

1.1.1 Online Judge System

An online judge (OJ) system is designed to provide users with a reliable evaluation environment in the Cloud where submitted source codes are compiled and tested, and then deliver evaluation results in the shortest possible time in Cloud [15]. OJ systems are becoming popular all over the world because of their variety of useful applications, such as job hunting, recruitment, programming competitions, development platforms, education, data mining, etc. We can

²<https://www.apple.com/siri/>

³<https://alexa.amazon.com/>

⁴<https://assistant.google.com/>

define the OJ system as follows:

Definition 1 (*Online Judge System*) *The OJ system is a web-based platform that receives solution code (S_i), then evaluates the solution codes (S_i) based on a set of test input/output instances (T_i), and finally produces a verdict (V_i). So, the function of OJ system can be expressed as $O(S_i, T_i) \rightarrow V_i$.*

The possible verdicts (V_i) generated by the OJ systems are listed along with their brief descriptions. (i) *Compile Error (CE)* Compilers are unable to compile the submitted solution codes. (ii) *Runtime Error (RTE)* A solution code is failed to execute, possible causes are pointer value has been exceeded, some values have been divided by zero, stack overflow, etc. (iii) *Time Limit Exceeded (TLE)* The time allotted for a solution code has exceeded. (iv) *Memory Limit Exceeded (MLE)* The memory allotted for a solution code has exceeded. (v) *Output Limit Exceeded (OLE)* A solution code produces many outputs. (vi) *Wrong Answer (WA)* A solution code generates incorrect output during the test cases. (vii) *Presentation Error (PrE)* Output does not match (blank lines, extra spaces, etc.) with the test cases. (viii) *Accepted (AC)* A solution code has passed all the test cases [16]. If a solution code has received the verdict *AC* for all test instances, then the final verdict of the solution code is also *AC*. Otherwise, the final verdict would be different from *AC* [15].

$$V = AC \iff \forall_i V_i = AC \quad (1.1)$$

$$V = V_j \iff (\forall_{i < j} V_i = AC) \wedge (V_j \neq AC) \quad (1.2)$$

1.1.2 Programming Learning with Online Judge System

OJ systems introduce an alternative programming learning platform where students and programmers can conduct their programming learning activities throughout the year [15]. An OJ system is a type of TAL system used as an academic and professional learning tool (e.g., programming, logical implementations, and various types of exercises). In 1970, the idea of the OJ system was first introduced in the ACM International Collegiate Programming Contest (ICPC) organized by Texas A&M University [15]. After several decades, the ACM ICPC has become one of the most prestigious programming competitions in the world. In 2015, more than 40,000 students from 2,000 universities and 102 countries participated in the regional phase of this

event [15]. Programming contests are one of the fastest growing extracurricular activities in the field of computer science, having a significant impact on the development of programmers' programming skills [17]. Many educational institutions are developing their own OJ or programming assessment systems to provide students with the opportunity to learn better programming. A few examples of OJ systems are AOJ (University of Aizu) [18–20], UVa (University of Valladolid) [17], URI (Universidade Regional Integrada) [21], and Judge.org (Universitat Politècnica de Catalunya) [22], which are being used as academic tools for different programming and exercise courses. OJ systems are not only effective for programming courses but also have wide adaptations across different domains [23]. Because of these OJ systems, a large number of solution codes and logs are generated on a regular basis that can be valuable resource for programming educational research.

1.1.3 Programming Data Analysis

In the last few years, e-learning/OJ platforms have become more popular for a variety of reasons and demands, including teacher shortage, unbalanced student-teacher ratio, logistical and infrastructure constraints, natural disasters, high cost of technical and professional courses, dissemination of education to a large number of people, time saving and easy access to many courses, and programming education [3, 24]. As the use of e-learning/OJ systems increases, different types of data are being generated regularly. Some data are structured whereas some are unstructured. Therefore, it is very difficult to retrieve useful information from this huge amount of mixed data archives using traditional statistical algorithms [25].

However, clustering techniques are widely used in data analysis and play an key role in this field. Clustering is an unsupervised machine learning algorithm, and the overview of the clustering process is shown in Figure 1.1.

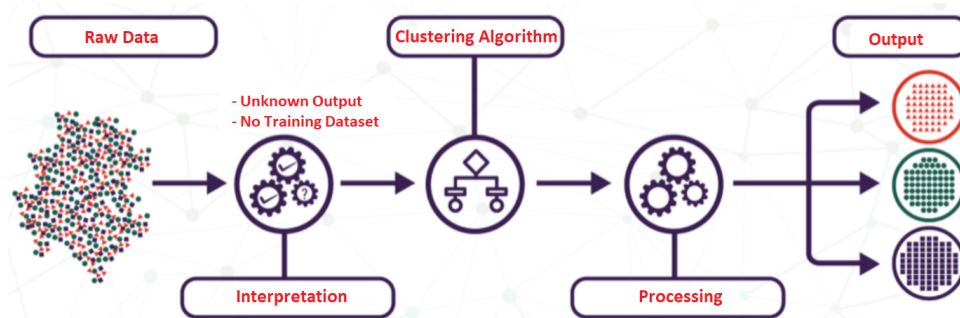


Figure 1.1: An overview of unsupervised clustering process

With the diversification of data, many variations of clustering techniques have been developed simultaneously to analyze different types of data. The clustering technique is used to gain insight into the structure of the data, and homogeneous data is grouped by calculating the Euclidean or correlation-based distance. Each clustering technique has its advantages and disadvantages for clustering data. To the best of our knowledge, there is no single clustering technique that can handle all types of data including text, numbers, images, and videos. Xu and Tian [26] conducted a comprehensive survey on clustering techniques and discussed their advantages, disadvantages, evaluations, and their complexity. Clustering techniques can be classified into different groups (hierarchy, fuzzy theory, distribution, density, graph theory, grid, fractal theory, and model) based on their working procedures. Here, we present some examples of classical clustering techniques such as K-means, SVM, KNN, PAM, CLARA, BIRCH, CURE, FCM, FCS, DBCLASD, GMM, DBSCAN, OPTICS, CLICK, STING, CLIQUE, and COBWEB [26]. There are many improved and extended versions of these classical clustering algorithms. Among them, K-means is considered one of the most widely used clustering algorithms because of its simplicity.

If the data is in Euclidean space, the following equation (1.3) can be used to calculate the objective function (J) that measures the cluster quality in the K-means clustering algorithm. Usually, K-means is not limited to data in Euclidean space, but is also applied to document data. In this case, the cosine similarity measure is used according to equation (1.4).

$$J = \sum_{j=1}^k \sum_{i=1}^n \|x_i^{(j)} - c_j\|^2 \quad (1.3)$$

where k is the number of clusters, n is the number of data points, and c_j is the centroid of cluster j .

$$\cos(\theta) = \frac{G.H}{\|G\|\|H\|} = \frac{\sum_{i=1}^n G_i H_i}{\sqrt{\sum_{i=1}^n G_i^2} \sqrt{\sum_{i=1}^n H_i^2}} \quad (1.4)$$

where G and H are two vectors of attributes, G_i and H_i are components of vector G and H respectively.

On the other hand, association rule mining (ARM) has been widely used for data mining purposes [27]. ARM is an unsupervised algorithm and was first introduced in research [28]. Association rules are considered to be IF-THEN relationships, meaning that if a customer bought an item A , then there is a high probability that a customer will choose an item B , as shown in

Figure 1.2.

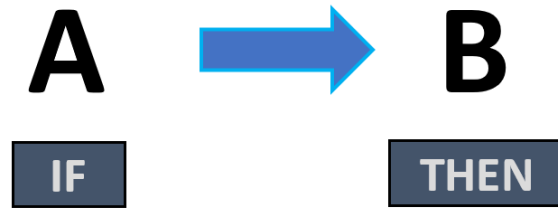


Figure 1.2: A simple IF-THEN relationship for ARM approach

There are diverse applications of the ARM technique in various fields such as pattern mining, education, social, medical, census, market-basket, and big data analysis. ARM is an efficient technique for obtaining frequent items from large datasets [29, 30]. Among the many types of ARM algorithms, Apriori and frequent pattern (FP)-growth algorithms are the most widely used [31, 32]. Several terms are important for mining patterns and association rules, such as *support* and *confidence*. *Support* is useful to know the most frequent items or frequently purchased items in the data set. *Confidence* provides information about how often items A and B occur simultaneously ($A \rightarrow B$) for item A .

$$Support (Sup) = \frac{frequency (A, B)}{N} \quad (1.5)$$

$$Confidence (Con) = \frac{frequency (A, B)}{frequency (A)} \quad (1.6)$$

where N is the total number of transactions in a dataset.

The amount of content on e-learning/OJ platforms is increasing, and at the same time, opportunities for research using the resources of e-learning platform are also increasing. Recommending relevant and appropriate content to users (e.g., students, instructors, teachers, etc.) is a challenging and tough task for any e-learning/OJ platform. Perumal et al. [33] proposed a novel personalized recommending system (RS) to provide appropriate supportive content to users. Recently, some RSs have been using a mixed approach of content-based filtering and collaborative filtering to achieve high-quality results in specific context [34]. In addition, most RSs are built with a collaborative, knowledge-based, content-based, and hybrid approaches [35].

1.1.4 Machine Learning in Programming

Over the past few decades, artificial neural networks (ANN) or artificial intelligence (AI) have achieved high accuracy in solving many complex tasks such as object recognition, computer vision, language translation, program code evaluation and classification, speech recognition, and so on. In 1956, McCarthy et al. first introduced the concept of ANN at the Dartmouth Conference, whose structure is completely similar to the nervous system of the human brain. Since then, many ANN have been developed in recent decades to achieve human-like performance, and they show great potential in many fields. The evolution of ANN is shown in Figure 1.3.

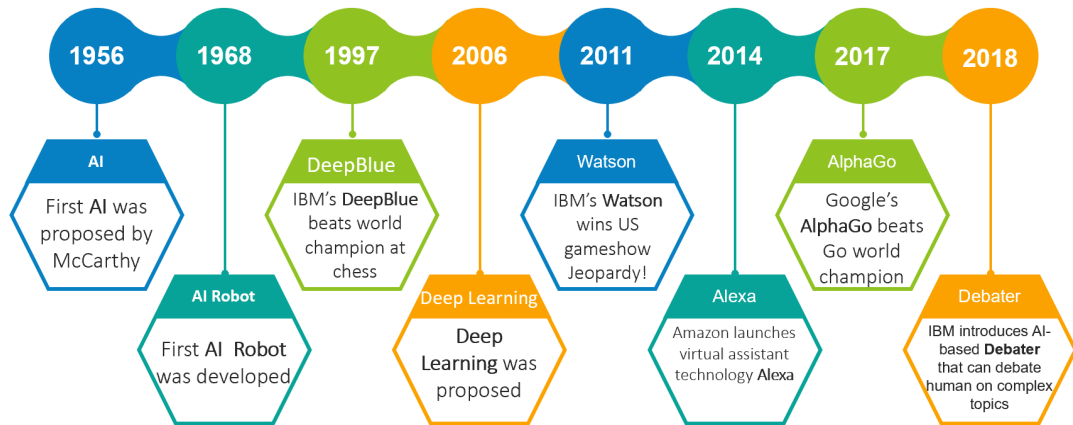


Figure 1.3: Evolution of the ANN

In addition, there are many variations of ANNs, including single-layer neural network (NN), feed forward neural network (FNN), deep neural network (DNN), etc. Here we briefly describe the mathematical concepts of the basic single-layer NN. A single-layer NN is often referred to as a perceptron, where X is a set of inputs $X = \{x_0, x_1, x_3, \dots, x_n\}$, W is a connecting weight of each input $W = \{w_0, w_1, w_3, \dots, w_n\}$, and b is a bias value. A single-layer NN always produces a single output via a non-linear function such as sigmoid, Tanh, and ReLU. The output of a single layer NN can be calculated by the equation (1.7).

$$Y = \sigma(XW + b) \quad (1.7)$$

In contrast, there is a hidden layer between input and output layers, which is called multi-layer NN. Basically, the multi-layer NN is a foundation for DNNs such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short-term memory (LSTM), etc.

The output of the multi-layer NN can be written as follows:

$$Y = \sum_{i=1}^h o_i \sigma(x_i) \quad (1.8)$$

$$x_i = \sum_{j=1}^n w_{ij} x_j + b_i \quad (1.9)$$

where, w_{ij} is the weight connecting from the input unit j to the hidden unit i , o_i is the weight connecting from the hidden unit i to the output unit, b_i is a bias of the hidden unit i , and $\sigma(x_i)$ is the sigmoid activation function.

Practically, DNN-based models have yielded some significant results that go beyond human-level expertise. DNN-based models are used in many complex practical application domains (i.e., robotics, virtual assistant, gaming, debating, and source code completion) worldwide. For example, Google has developed a DNN model called AlphaGo that can play Go games ⁵; in 2017, AlphaGo defeated Ke Jie (then ranked #1 in Go) at the Future of Go Summit in South Korea ⁶. This was a major landmark achievement in AI, making people believe that AI is now a reality and the future of the world. At the same time, graphics processing units (GPUs) made AI more powerful to learn millions of data through parallel computations. Figure 1.4 provides a brief overview of AI, machine learning (ML), and DNN and their applications in programming.

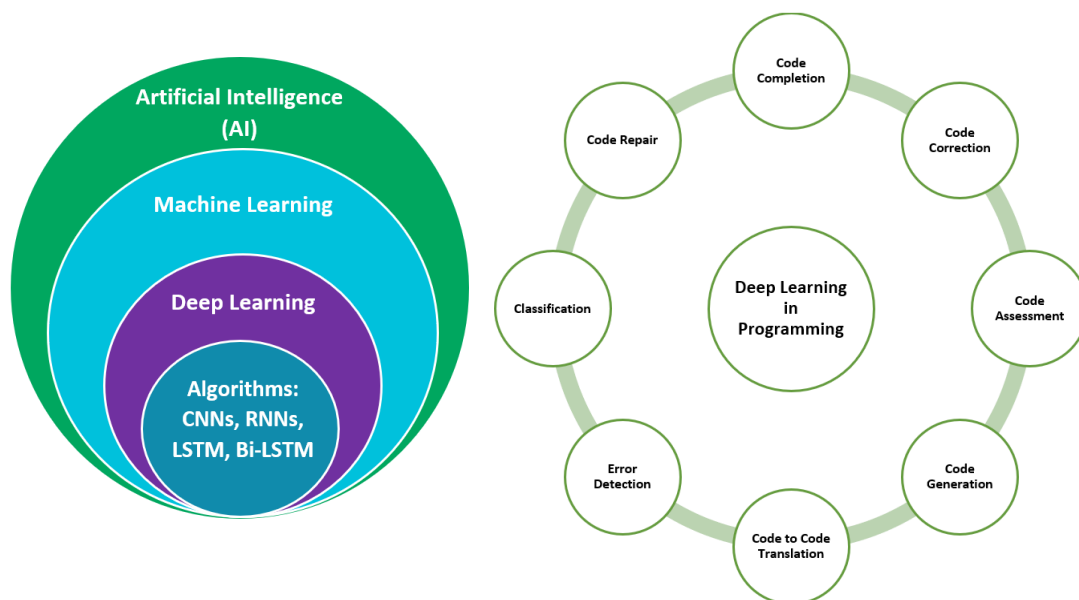


Figure 1.4: The AI landscape and the application of DNN in programming

⁵[https://en.wikipedia.org/wiki/Go_\(game\)](https://en.wikipedia.org/wiki/Go_(game))

⁶<https://en.wikipedia.org/wiki/AlphaGo> and <https://deepmind.com/research/case-studies/alphago-the-story-so-far>

1.1.4.1 RNN-based Code Assessment Models

In recent years, DNNs have shown their potential in programming education, performing a variety of complex tasks related to programming. The deep architecture of the DNN learns more complex relationships of the program codes to achieve better performance in programming-related tasks. Due to the availability of large amounts of program codes generated by e-learning/OJ platforms, DNN-based models become effective by leveraging these resources. Pu et al. [36] proposed a source code correction method based on LSTM using code segment similarities. The study leveraged the sequence-to-sequence (seq2seq) neural network model with natural language processing tasks for the code correction process. Another study [37] proposed a deep software language model based on RNNs. The experimental results showed that the model outperforms traditional language models such as n -gram and cache-based n -gram in a Java corpus. The software language model shows great promise in the field of software engineering. An example of the training and prediction process of a sequential LSTM language model is shown in Figure 1.5.

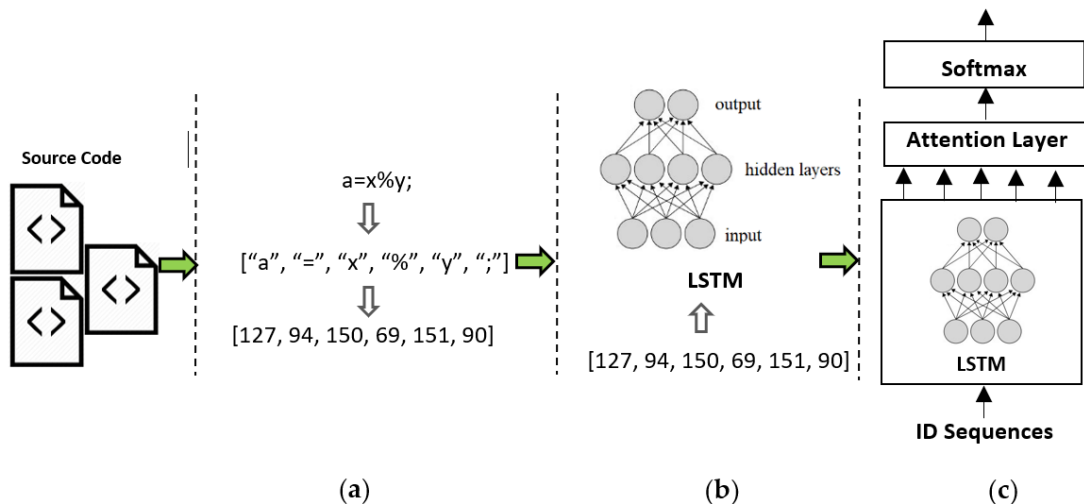


Figure 1.5: An example of (a) word embedding and encoding process, (b) LSTM model training, and (c) token prediction by the trained LSTM model

Terada et al. [38] proposed an LSTM-based model for programming education where the model predicts the next word by analyzing incomplete source code. Novice programmers often struggle to write a complete program from scratch. To help them, the model predicts the next word to complete a program. The LSTM-based model achieved a high degree of prediction accuracy. Fault detection in source code has become an important research topic [39]. Ram and Nagappan [40] proposed a hierarchical model that uses CNNs and LSTM for sentiment analysis

in software engineering. Tai et al. [41] presented a model called Tree-LSTM where an LSTM network works like a tree. The model evaluates the tasks of prediction of semantic relatedness based on sentence pairs and sentiment classification. Reyes et al. [42] classified archived source code by type of programming language using an LSTM network. Fan et al. [43] presented an attention-based RNN for source code defect prediction.

1.1.4.2 Transformer-based Code Assessment Models

The Transformer-based seq2seq language model has shown excellent performance in language translation, classification, speech recognition, code generation, code summarization, and code completion due to its internal attention and residual structure. The Transformer architecture was first proposed in 2017 by A. Vaswani et al. [44] and has since become one of the most innovative models in the field of language modeling. Usually, the Transformer is a type of NN architecture consisting of a multi-headed self-attention mechanism with an encoder-decoder structure. A basic architecture of the Transformer model [44] is shown in Figure 1.6.

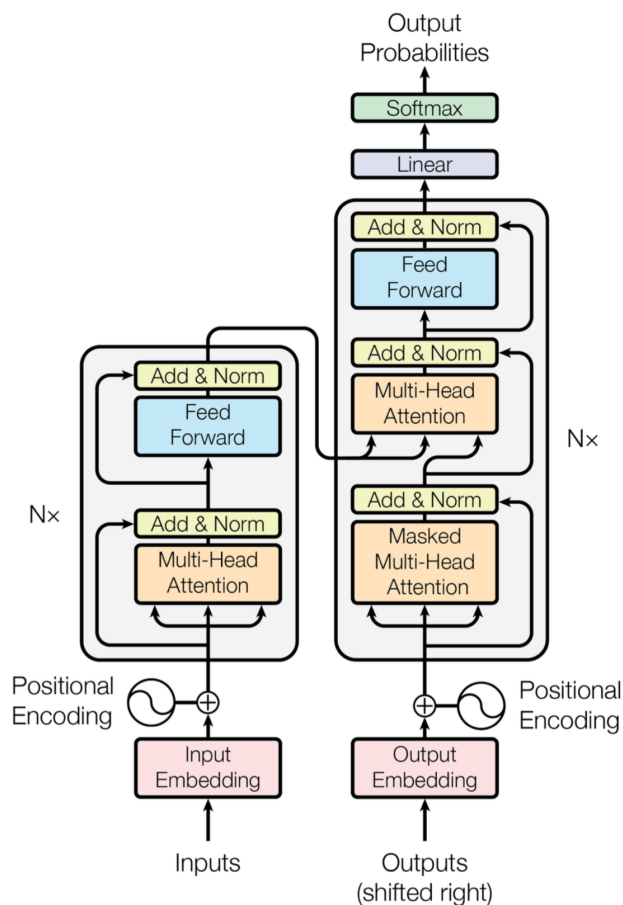


Figure 1.6: A basic architecture of the transformer model

Basically, Transformer is all about attention, which uses three (03) primary terms such as Query (Q), Key (K), and Value (V) to calculate attention weights. The attention formula (1.10) can be written as follows:

$$attention = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1.10)$$

where Q is the vector representation of a single word, K is the vector representation of all words, and V is the vector representation of all words in the sequence.

Transformers offer many advantages over traditional NNs or even RNNs, including (i) the ability to consider very wide-ranging dependencies, (ii) no possibility of gradient vanishing and explosion, (iii) fewer training steps, (iv) no recurrence barrier for parallel computations, and (v) much higher computational efficiency. Various models using Transformer architecture have been proposed, such as the GPT (Generative Pre-Training) model using the decoder part of the Transformer [45], the BERT (Bidirectional Encoder Representations from Transformers) model using the encoder part of the Transformer [46], and the T5 (Text-to-Text Transfer Transformer) model using the encoder-decoder structure of the Transformer [47]. RoBERTa [48] is an improved model based on the BERT developed by the Facebook AI team, which improves the performance of BERT in various evaluation metrics.

In particular, the Transformer-based models have also produced state-of-the-art results in various programming tasks such as code completion, code summarization, code generation, and error identification. IntelliCode [49] is a Transformer-based model leveraged to complete multilingual source code. Z. Sun et al. [50] proposed TreeGen for code generation, which uses the attention mechanism of the Transformer model to overcome the long-term dependency of source code. CodeBERT [51] is a bimodal pre-trained Transformer-based model used for both programming and natural languages. CodeBERT can be used for a variety of downstream applications of programming and natural language-related tasks, such as code completion, generation, and language translation. In [52], a deep learning-based transformation (DLBT) model is proposed for automatic generation of Pseudo-code from source codes.

The above research shows that RNN- and Transformer-based models have used for various programming tasks, including program code evaluation, code generation, code translation, error detection, code repair, code completion, and classification.

1.2 Scope and Motivation of the Study

In recent years, OJ systems have been increasingly used for programming learning and assessment purposes in educational institutions. Usually, OJ systems have stored various parameters including submission time, verdicts, CPU time, and memory usages when evaluating solution codes. However, a typical e-learning platform cannot automatically evaluate solution codes nor store such parameters. These valuable resources (e.g., code archives, verdicts, submission logs, etc.) from OJ systems provide room for further research and analysis. The results of this research could reveal programming flaws of students/programmers and thus broaden the scope of possible improvements. Therefore, we are motivated to use these valuable real-world resources for data analysis as well as the development of a machine learning-based model to evaluate solution codes. We are primarily motivated by the following two reasons, which will be reflected in this dissertation.

1. **It is important to develop a data analysis framework using practical (programming, OJ, etc.) data, the results of which could have an impact on programming education and industry.**
2. **A large number of solution codes are regularly archived in OJ systems. These real-world solution codes can be used in the development of a machine learning model to support programming learning in academia and industry.**

The main goals of this dissertation are to develop (*i*) a data analysis framework for discovering students' programming-related activities, invisible features, shortcomings, and (*ii*) machine learning-based models for code assessment and classification using real-world data of an OJ system to address shortcomings. To achieve such targets, the following key components were used in the proposed data analysis framework and ML models:

- **Modified K-means (MK-means) clustering algorithm:** K-means clustering is an unsupervised ML algorithm used to group similar data through mathematical processes. It is one of the simplest and most commonly used clustering algorithms. As an improvement to the classic K-means algorithm, we proposed an MK-means clustering algorithm in this dissertation, which contains two modules. The first module is the initial center selection module (ICSM), which is used to (*i*) select optimal centers and (*ii*) form clusters with the most similar data. The second module is the outlier detection module (ODM), which is

used to (i) detect outliers (irrelevant/insignificant data points), (ii) remove them from the dataset, and (iii) improve the overall quality of the cluster. We used solution submission logs and scores from a programming course (Algorithms and Data Structures) conducted on an OJ system. After clustering, we extracted hidden/invisible features that are not clearly visible in the plain data.

- **FP-growth algorithm:** FP-growth is also an unsupervised ML algorithm used for data mining, pattern mining and association rule mining purposes. The FP-growth algorithm is also known as ARM algorithm. This algorithm has been used in many fields such as educational data analysis, medical data analysis, market-basket analysis, and census data. Usually, ARM aims to find a set of cooccurring high-frequency items and extract the correlation among items from large dataset. In this dissertation, we used the FP-growth algorithm for data mining (i.e., frequent items, pattern mining, etc.) and association rules mining.
- **Recurrent Neural Network:** LSTM is a kind of recurrent neural network that can process long input sequences using its special gate structures. It consists of four gates such as input, output, forget, and cell state. Since our solution codes are long sequences of statements, the LSTM structure can lead to better results. In this dissertation, we also exploited different variants of LSTM neural network such as *attention – based LSTM*, *bidirectional LSTM*, and *stacked bidirectional LSTM* neural network to develop the models for code evaluation and classification.

1.3 Dissertation Contributions

In this section, we summarize the contributions of this dissertation. We start our research in two challenging directions. *First*, we developed a novel data analysis framework using real-world data from OJ system to identify the programming-related problems encountered by students/programmers. The framework helped us to extract various unseen features and analytical results from the programming data. *Second*, we developed machine learning models for programming code assessment and classification. The model enabled us to evaluate the code with human-like accuracy and detect the errors in codes that could not be recognized by the conventional compilers. The detailed research methodology and experimental results are presented in *Chapter 2*, *Chapter 3*, and *Chapter 4*. The main contributions of each chapter are

summarized as follows:

1.3.1 Chapter 2: A Comprehensive Data-driven Analysis to Explore the Impact of Programming in Education

The OJ systems are now widely used by educational institutions as learning tools in programming and other exercise-based classes. These platforms play an important role in improving students' programming skills, knowledge, and overall academic performance. The vast resources (e.g., code archives, submission logs, etc.) generated by these systems can help researchers to find students' flaws in programming and thus expands the scope of available improvements. Therefore, in Chapter 2, we proposed a novel data analysis framework to extract hidden features, analytical results, patterns, and association rules from programming-related data. This Chapter makes the following contributions:

- We proposed a data analysis framework for programming education. To demonstrate the effectiveness of the proposed framework, experiments are conducted on real-world programming/problem-solving data.
- The correlation between programming skills and academic performance are presented.
- Various programming and academic weaknesses and strengths are highlighted through empirical analysis.
- Important and relevant features, rules, and patterns from the submission logs and scores are extracted that are not plainly visible in a simple form of data.
- Useful recommendations are generated for students and educators on the basis of the extracted features, rules, and patterns.
- The proposed framework and its data analysis process can be useful for other related academic courses and disciplines to discover hidden features/correlations in e-learning data. For example, this framework can be applied to a course that consists of theory and hands-on activities and collects resources/data, like a programming course.
- The proposed data analysis framework can be integrated into e-learning platforms and OJ systems.

- The MK-means clustering algorithm was proposed, and the effectiveness of data clustering in Euclidean space was also demonstrated.

1.3.2 Chapter 3: Code Assessment and Classification Using Attention-Based LSTM Neural Network

A large amounts of solution codes are regularly accumulated by OJ systems at the academic and industrial levels. Typically, the solution codes contain various types of errors including syntax, semantics, communication, computation, and logic errors. Depending on the nature of these errors in the solution codes, the OJ systems have made different verdicts including *CE*, *RTE*, *TLE*, *MLE*, *OLE*, *PrE*, *AC*, and *WA*. It is sometimes difficult for students or professional programmers to detect logic errors (*TLE*, *MLE*, *OLE*, *WA* etc.) in solution codes, even with the help of traditional compilers. Helping programmers, especially novice programmers, to correctly evaluate solution codes and classify codes has become an important research topic. Given the importance of this research gap, we proposed a machine learning model for solution code evaluation and classification in Chapter 3. We trained the model with real-world solution codes collected from an OJ system. To strengthen the proposed model, we combined the attention mechanism with LSTM to enhance the model performance for solution code assessment and classification. We also fine-tuned various parameters of the network to achieve better results. The main contributions of this Chapter are summarized below:

- The proposed machine learning based model can help students, novice and professional programmers to evaluate their solution codes.
- The proposed model can detect such errors (*TLE*, *MLE*, *OLE*, *WA*, etc.) that cannot be identified by conventional compilers.
- The proposed model accuracy is approximately 62% that outperformed other state-of-the-art models.
- The proposed model can classify (correct or incorrect) the source codes based on the detected errors. The classification accuracy is 96% that is much higher than other compared models.
- The proposed model highlights defective position with location/line number in source codes.

- The proposed model improves the ability of learners to fix errors in source code easily by using the location/line numbers.

1.3.3 Chapter 4: Code Assessment and Classification Using Bidirectional LSTM

Basically, LSTM neural networks only consider past input sequences for model training and predictions. However, the functions, classes, methods, and variables of a source code may depend on both past and subsequent sections or lines of code. In such cases, the LSTM may not provide optimal results. To address this gap, we proposed a bidirectional LSTM (BiLSTM) language model for source code assessment and classification. A BiLSTM neural network can combine both past and future code sequences to produce output. Furthermore, a stacked BiLSTM model for classifying codes built in multi programming languages (MPLs) is proposed. Since methods, classes, variables, tokens, and keywords have both short-term and long-term dependencies, the stacked BiLSTM layers make the model deeper and provide a better understanding of the context of the codes. In Chapter 4, we made incremental improvements to Chapter 3 and also developed a novel multi-class classification model for identifying algorithms in codes. The main contributions of the Chapter are summarized as follows:

- The proposed BiLSTM language model for code assessment can effectively detect errors including logical errors (*TLE*, *MLE*, *OLE*, *WA*, etc.) and provide corrections for erroneous codes.
- The BiLSTM model can be helpful to students, programmers (especially novice programmers), and professionals who often struggle to resolve code errors.
- The model can be used for different real-world programming learning and software engineering platforms and services.
- The stacked BiLSTM model classifies source codes based on the algorithms. The model can help students and programmers to identify the algorithms used in the source code. Programmers can understand the code better if they know the written algorithm in the code.
- The stacked BiLSTM model can be deployed in the field of software engineering to recognize code in the large code archives.

1.4 Dissertation Outline

The visualized outline of this dissertation is illustrated in Figure 1.7. In this dissertation, we mainly focus on two issues: (i) developing a data analysis framework to explore the invisible information from programming and academic data, and (ii) developing a machine learning model for program code evaluation, binary classification (correct or incorrect), and multi-class classification. The remainder of this dissertation is organized as follows. *Chapter 1* presents the background of the research, which primarily includes TAL and its impact on education. The application of OJ systems in conducting programming and other exercise-based courses in educational institutions is also discussed. Chapter 1 also presented the research opportunities for using the outcomes of these learning tools, including OJ systems. In addition, Chapter 1 presents the practical implementation of machine learning techniques for data analysis as well as code evaluation and classification.

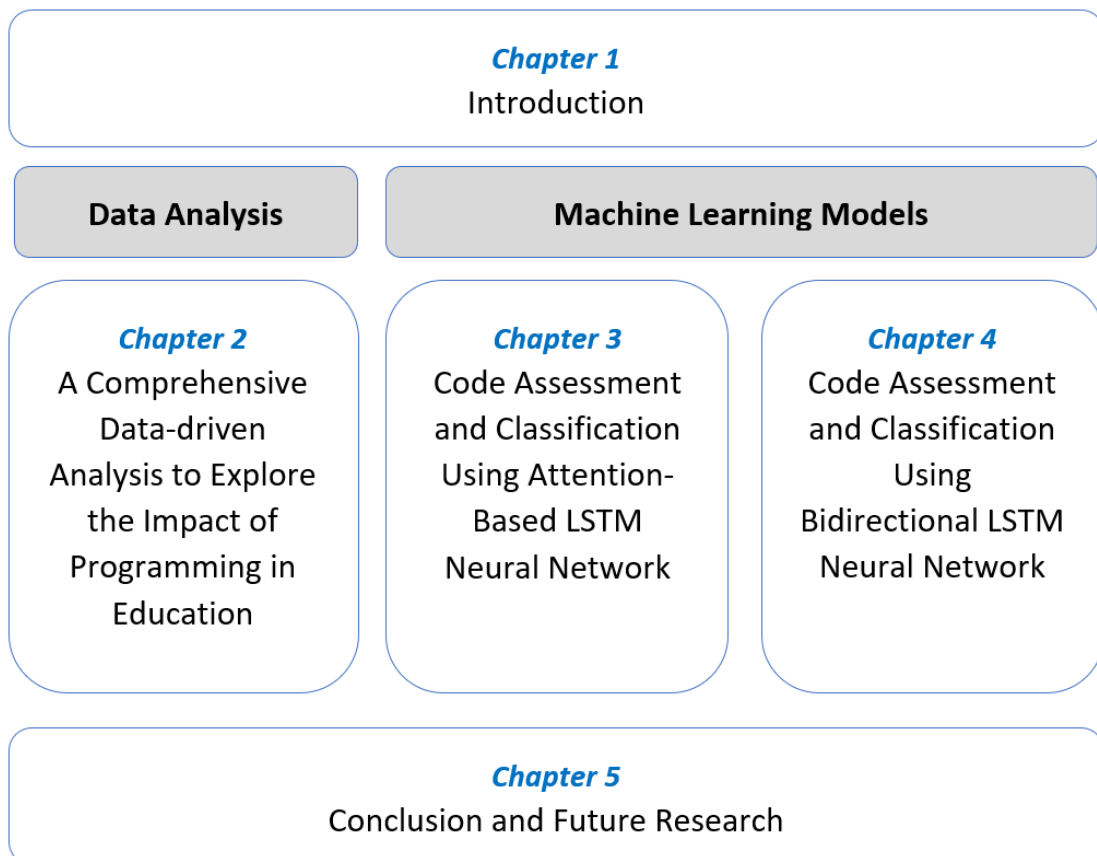


Figure 1.7: Outline of the Dissertation

Chapter 2 provides the framework for data analysis that can handle both OJ system data and academic data. It also shows the impact of programming skills on academic outcomes

through a comprehensive analysis of these data. Particularly, the background and related works are presented in Section 2.2. Section 2.3 describes the dataset and preprocessing. In Section 2.4, the proposed approach is presented. The experimental results are presented in Section 2.5 and discussed in Section 2.6. Section 2.7 summarizes the Chapter 2.

Chapter 3 focuses on developing a machine learning model for code evaluation and binary classification using real-world solution codes collected from an OJ system. Particularly, the background and prior researches are presented in Section 3.2. Section 3.3 focuses on the overview of language models and RNNs. In Section 3.4, we present the proposed machine learning based approach for code assessment. Data collection and problem description issues are presented in Section 3.5. The experimental results are presented in Section 3.7. Section 3.8 discusses the experimental results. The summary of this Chapter in Section 3.9.

Chapter 4 develops a model for program code evaluation, binary classification, and multi-class classification using BiLSTM neural networks. In particular, the research background and related works are described in Section 4.2. The architecture and mathematical background of the proposed BiLSTM model are presented in Section 4.3. In Section 4.3.4, we present the experimental results of code assessments and classification using BiLSTM. Section 4.4 presents stacked BiLSTM model for multi-class classification task. In Section 4.4.6, we present the experimental results of multi-class classification using stacked BiLSTM. Section 4.5 concludes the Chapter.

Finally, we summarize this dissertation in *Chapter 5* with outlooks on the future research directions.

1.5 Publications

The research and experimental results of this dissertation have been published or submitted to the following peer-reviewed journals and conferences. The results of the following papers are presented in Chapters 2, 3, and 4 of the dissertation.

Major Journals

1. **Md. Mostafizer Rahman**, Yutaka Watanobe, Taku Matsumoto, Rage Uday Kiran, Keita Nakamura. “Educational Data Mining to Support Programming Learning Using Problem-Solving Data,” *IEEE Access*, 2022.

2. **Md. Mostafizer Rahman**, Yutaka Watanobe, Rage Uday Kiran, Truong Cong Thang, Incheon Paik. “Impact of Practical Skills on Academic Performance: A Data-Driven Analysis,” *IEEE Access*, 2021.
3. **Md. Mostafizer Rahman**, Yutaka Watanobe, Keita Nakamura. “A Bidirectional LSTM Language Model for Code Evaluation and Repair,” *Symmetry*, 13(2), 2021.
4. **Md. Mostafizer Rahman**, Yutaka Watanobe, Keita Nakamura. “A Neural Network Based Intelligent Support Model for Program Code Completion,” *Scientific Programming*, vol. 2020, Article ID 7426461, 2020.
5. **Md. Mostafizer Rahman**, Yutaka Watanobe, Keita Nakamura. “Source Code Assessment and Classification Based on Estimated Error Probability Using Attentive LSTM Language Model and Its Application in Programming Education,” *Applied Sciences*, 10(8), 2020.
6. Raihan Kabir, Yutaka Watanobe, Md Rashedul Islam, Keitaro Naruse, **Md. Mostafizer Rahman** “Unknown Object Detection Using a One-Class Support Vector Machine for a Cloud–Robot System,” *Sensors*, 22(4), 2022.
7. Yutaka Watanobe, **Md. Mostafizer Rahman**, Taku Matsumoto, Rage Uday Kiran, Ravikumar Penugonda. “Online Judge System: Requirements, Architecture, and Experiences,” *Int’l Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2022.

Major Conferences

1. **Md. Mostafizer Rahman**, Yutaka Watanobe, Rage Uday Kiran, Raihan Kabir. “A Stacked Bidirectional LSTM Model for Classifying Source Codes Built in MPLs,” 1st Workshop on Machine Learning in Software Engineering (MLiSE 2021) Held in conjunction with the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2021), *Communications in Computer and Information Science (CCIS)*, Springer, 2021. (**Best Paper Award**)
2. **Md. Mostafizer Rahman**, Yutaka Watanobe, Rage Uday Kiran, Truong Cong Thang, Incheon Paik. “Challenges and Exit Strategies for Adapting Interactive Online Education Amid the Pandemic and its Aftermath,” 2021 IEEE International Conference on Engineering, Technology & Education (TALE 2021), IEEE, 2021.

3. **Md. Mostafizer Rahman**, Yutaka Watanobe, Rage Uday Kiran, Keita Nakamura. “A Novel Rule-Based Online Judge Recommender System to Promote Computer Programming Education,” The 34th International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE 2021), *Lecture Notes in Computer Science (LNCS)*, Springer, 2021.
4. **Md. Mostafizer Rahman**, Yutaka Watanobe, Keita Nakamura. “Evaluation of Source Codes Using Bidirectional LSTM Neural Network,” 2020 3rd IEEE International Conference on Knowledge Innovation and Invention (ICKII), pp. 140-143, IEEE, 2020.
5. Yutaka Watanobe, **Md. Mostafizer Rahman**, Alexander Vazhenin, Jun Suzuki. “Adaptive User Interface for Smart Programming Exercise,” 2021 IEEE International Conference on Engineering, Technology & Education (TALE 2021), IEEE, 2021.
6. Atsushi Takamiya, **Md. Mostafizer Rahman**, Yutaka Watanobe. “A Framework and Its User Interface to Learn Machine Learning Models,” 14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc 2021), IEEE, 2021.
7. Yutaka Watanobe, **Md. Mostafizer Rahman**, Rage Uday Kiran, Ravikumar Penugonda. “Online Automatic Assessment System for Program Code: Architecture and Experiences,” The 34th International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE 2021), *Lecture Notes in Computer Science (LNCS)*, Springer, 2021.
8. Shunsuke Kawabayashi, **Md. Mostafizer Rahman**, Yutaka Watanobe. “A Model for Identifying Frequent Errors in Incorrect Solutions,” 2021 10th International Conference on Educational and Information Technology (ICEIT), pp. 258-263, IEEE, 2021.

Non-Major Conferences

1. **Md. Mostafizer Rahman**, Shunsuke Kawabayashi, Yutaka Watanobe. “Categorization of Frequent Errors in Solution Codes Created by Novice Programmers,” The 3rd ETLTC International Conference on Information and Communications Technology (ETLTC2021), SHS Web of Conference, 2021.

Chapter 2

A Comprehensive Data-driven Analysis to Explore the Impact of Programming in Education

2.1 Introduction

Most courses in information and communication technology (ICT), computer science, and engineering-related disciplines are designed with a practical basis. Basically, each course consists of two parts namely, theory and exercise where theory develops students' theoretical knowledge, ideas, and memorization. In contrast, exercise or practical application develops logic, critical thinking, problem-solving skills, and implementation skills. Computer programming is an example of a practical course in these disciplines. The necessity of programming education is rapidly growing in pace with the increasing expansion of computers into our daily lives; thus, computer programming is among the key courses in the ICT discipline and has become a foundational course in other disciplines, as well [53]. In a recent effort to encourage students, including children, to take an increased interest in programming, numerous online programming platforms have become available. Here, it should be noted that because the primary requirement of programming education is to ensure that students achieve computer literacy [53], most educational institutions that teach programming have redesigned their academic curriculum to effectively meet the basic literacy requirements of programming education.

Basic computer programming courses are normally available in the first semester of university studies. Initial programming classes have the ancillary role of attracting students to the

field of computer programming. Because students may make decisions based on these initial programming classes, it is essential that those classes impart positive programming experiences. Note that, introductory programming courses have a significant rate of failure and dropout [54]. However, due to limited amounts of class time, classrooms and teachers, and limitations in other forms of logistic support, it is difficult to fully educate students in programming through traditional programming classes alone. To overcome these problems, OJ systems provide additional platforms that enable students to continue their programming studies over a period of years [15]. Such systems normally contain large collections of interesting programming problems [55] that students can pursue independently or teachers can assign to stimulate students' interest. The concept of the OJ system was first introduced at the 1977 International Collegiate Programming Contest (ICPC) [17, 56], which is now held annually. Furthermore, because OJ systems have proven useful, many universities and colleges are now attempting to develop online support systems for programming education [21, 22, 57].

Today, OJ systems are used by many educational institutions to conduct courses related to programming, computing, and software engineering [58, 59]. Many universities have created their own automated program assessment (APA) systems for programming courses to accelerate students' learning [60–62]. As a result, a large number of programming-related submission logs are created every day by OJ or APA systems in various organizations worldwide, which can be valuable resources for research and analysis [63, 64]. Therefore, this Chapter aims to use programming-related resources (submission logs and scores) for empirical research and analysis.

Educational data collected from various e-learning platforms such as Moodle, MOOCs, OJs, and APAs are not unified, structured, well-organized, neat and in a collected format because the data archiving format differs from one e-learning platform to another. Therefore, educational data mining (EDM) and learning analytics (LA) techniques are effective in transforming these big educational data into useful knowledge and patterns that can be applied to improve overall education. EDM has become an effective technique for exploring invisible knowledge and useful patterns in educational data [27]. Nowadays, traditional education is changing at an unprecedented pace and many academic activities are conducted on e-learning platforms. The collection of this vast amount of educational data has opened up opportunities for research and analysis to understand and improve learning outcomes. V. Hegde and S. Rao H.S. [65] presented an EDM-based framework to analyze students' performance in programming. The results of the

analysis helped students to improve their weak concepts through frequent faculty support and also offered benefits to institutions. Ang et al. [66] conducted a comprehensive survey and presented architectures and their challenges for growing big educational data.

On the other hand, LA refers to the collection, analysis, and visualization of educational data to understand and improve the learning processes and outcomes better. LA provides interventions based on the analysis of educational data to improve both learning and the learning environment [67]. Also, LA encompasses broader components of other disciplines such as EDM, academic analytics, learning sciences, cognitive sciences, human factors, psychology, and so on. Maher et al. [68] proposed a Personalized Adaptive Gamified E-learning (PAGE) model to enhance MOOCs LA and visualization in the learning process. The PAGE model helped learners in learning adaptation and visualization.

In this Chapter, our goal is to investigate the impact of practical/programming skills on academic performance through a comprehensive analysis using real-world e-learning data. Considering the context of this Chapter, these two important terms such as practical skills and academic performance are defined as follows.

Practical skills relate to reasoning, critical-thinking, problem-solving, and implementation skills. Let consider a basic programming course that consists of two learning activities such as theory-based and practice-based. The practice-based activities include programming, programming-related assignments, and coding tests. In this Chapter, performance in practice-based activities is referred to as practical skills. On the other hand, academic performance refers to theoretical knowledge, innovative ideas, and memorization. Performance in various theory-based activities includes algorithmic-based assignments, theory-based assignments, and paper-based tests which are referred to as academic performance.

To accomplish this study, a novel framework is proposed to extract students' hidden features, patterns, and association rules. Hidden features derived from submission logs and scores carry significant meaning. Chapter 2 makes the following contributions:

- We proposed a data analysis framework for programming education. To demonstrate the effectiveness of the proposed framework, experiments are conducted on real-world programming/problem-solving data.
- The correlation between programming skills and academic performance are presented.
- Various programming and academic weaknesses and strengths are highlighted through

empirical analysis.

- Important and relevant features, rules, and patterns from the submission logs and scores are extracted that are not plainly visible in a simple form of data.
- Useful recommendations are generated for students and educators on the basis of the extracted features, rules, and patterns.
- The proposed framework and its data analysis process can be useful for other related academic courses and disciplines to discover hidden features/correlations in e-learning data. For example, this framework can be applied to a course that consists of theory and hands-on activities and collects resources/data, like a programming course.
- The proposed data analysis framework can be integrated into e-learning platforms and OJ systems.
- The MK-means clustering algorithm was proposed, and the effectiveness of data clustering in Euclidean space was also demonstrated.

The rest of the Chapter is structured as follows. In Section 2.2, the background and related works are presented. Section 2.3 describes the dataset and preprocessing. In Section 2.4, the proposed approach is presented. The experimental results are presented in Section 2.5 and discussed in Section 2.6. Section 2.7 summarizes this Chapter.

2.2 Background and Related Works

In this section, we briefly introduce OJ or APA systems and their applications in programming education. In addition, supervised and unsupervised learning algorithms, association rule mining (ARM) algorithms, educational data mining and learning analytics are also presented.

2.2.1 Online Programming Learning Platform

OJ or APA systems are now widely used by educational institutions as academic learning tools in programming and other exercise-based classes. These platforms play an important role in improving students' programming skills, knowledge, and overall academic performance. The vast resources (e.g., code archives, submission logs, etc.) generated by these systems can help

researchers to find students' flaws in programming and thus expands the scope of available improvements. As a result, numerous studies have focused on programming education, educational data mining, and data-driven analysis using resources from OJ or APA systems.

In [57], the authors used learning log data extracted from the M2B system. A recurrent neural network is used to predict student performance. This study showed that numerous useful hidden features can be extracted by analyzing the M2B system's data. Mekterović et al. [60] proposed an APA system for conducting programming courses and created the educational software Edgar to automatically evaluate programming assignments and other programming-related tasks. Edgar provides a variety of services, including content writing, course administration, system monitoring, and troubleshooting. Furthermore, Edgar produces the results of various statistics in a visual format. APA systems provide many benefits for students as well as instructors. Meanwhile, a ranking system [69] based on student performance and quick responses has positively impacted programming learning. APA systems have extended the conventional use of the OJ systems for evaluating programming assignments and their use significantly stimulates students' interest in programming.

In [70], the authors extended the BOCA OJ system to improve its suitability for programming classes. The resulting PROBOCA project was used to aid classroom teachers. This method identifies problems by degree of difficulty, thus making it easier for teachers to match problems with each student's programming experience. Another study [71] presents a continuous programming assessment system for programming courses using automated assessment tools (AATs). A quantitative analysis was performed based on the relationship between the student and the AAT outcome. The submitted solutions are analyzed in depth using an AAT and judgments (either correct or incorrect) are provided. The experimental results showed that AATs help students to better understand computer programming. Lu et al. [72] presented programming education via an OJ system that has increased student performance levels in programming and other academic activities. Their experimental results show that the OJ system enhanced performance levels, as well as stimulated students' interest throughout the year- or semester-long course.

Toledo et al. [73] presented a fuzzy recommender system for OJ programming that provides suggestions to learners regarding their upcoming problems based on their past performance in the OJ system. That method also provided useful information to students via recommendations. In [74], OJ programming problems were classified using two topic-modeling algorithms, latent

dirichlet allocation and non-negative matrix factorization in order to extract relevant features from problem descriptions. The classification of OJ programming problems can help novice and advanced students to pick and solve appropriate problems.

Our approach differs from that of existing research by focusing on discovering hidden features from submission logs and scores to improve programming skills and academic performance. We also focus on finding the correlation between practical skills and academic performance based on the extracted hidden features. To the best of our knowledge, no study has been conducted to address this issue by using submission logs and scores.

2.2.2 Supervised and Unsupervised Learning Algorithms

Within the context of artificial intelligence and machine learning (ML), supervised and unsupervised algorithms are frequently used in real-world applications. In short, both input data and output labels are known in supervised learning (SL) algorithms. Formally, SL involves ML algorithms that are trained with known input data and associated output labels. Let $U = \{u_1, u_2, u_3, \dots, u_n\}$ be the set of input data and $V = \{v_1, v_2, v_3, \dots, v_n\}$ be the set of corresponding output labels of the input data U . Thus, the output function can be written as $V = f(U)$, where the output V depends on the input U and f is a mapping function. After training, the ML algorithm can predict the output label for all new input data. SL algorithms are divided into two categories such as classification and regression.

Classification is an SL approach that classifies a given set of data into classes. The classification model predicts the target class for a given data point. After training, the model predicts class names for data it has not seen before. There are two types of classification in ML such as binary classification (*true* or *false*) and multi-class classification. Typically, the evaluation of a classification model is done by computing the *precision*, *recall*, and *accuracy* scores. Examples of some classification algorithms include support vector machine, decision tree, random forest tree, artificial neural network, similarity learning, and k-nearest neighbor [75]. Similarly, regression is an SL approach used to predict the continuous output variable based on one or more independent (predictors) variables. Mainly, this approach is used for forecasting, time series modeling, prediction, and determining market trends. Examples of regression algorithms include linear regression, logistic regression, polynomial regression, decision tree regression, and random forest regression [75].

In contrast, unsupervised learning (USL) is a kind of ML algorithm in which models are

trained with unlabeled datasets. The USL algorithm can group the data based on their similarity features by applying some mathematical procedures. The USL algorithms have the following advantages over SL including hidden feature extraction, useful insights, human-like learning, handling unlabeled and uncategorized data. USL algorithms are divided into two categories such as clustering and association. Clustering is a USL algorithm used to group data into clusters, similarity characteristics of the data in a group are high, on the other hand, there is a minimal similarities with the data of another group. Examples of clustering algorithms include k-means, k-medoids, Density-based Spatial Clustering of Applications with Noise (DBSCAN), Clustering Large Applications based on RANdomized Search (CLARANS), and Clustering Large Applications (CLARA). Association is a USL algorithm that is used to find relationships between items in a large database. This algorithm determines the set of items that co-occur in a database. For example, if three items M , N , and O exist in the database, the algorithm can generate patterns/rules that co-occur such as $M \rightarrow N$, $(M \& N) \rightarrow O$, and $N \rightarrow M$. These patterns/rules are useful for analyzing market-basket, educational data, and so on. Examples of association algorithms include Apriori and frequent pattern (FP)-growth.

In [76], students have been classified by a clustering approach based on their learning behaviors. The clustering by fast search and finding of density peaks via heat diffusion (CFSFDP-HD) algorithm has achieved a better clustering performance than other clustering algorithms. The authors also proposed an e-learning system architecture that detects and responds to teaching content based on student learning capabilities. Tabanao et al. [77] proposed a method that classifies programmers using submission log data, such as compilation profiles, error profiles, compilation frequency, and error quotient profiles produced during an introductory programming course. This study identified correlations between the submission log data and the midterm examination scores of students.

In case of our dataset, the output labels are unknown because the submission logs and class performance scores have not yet output information that could be used for labeling (e.g., poor, good, very good, or genius). Accordingly, as it is necessary to select an algorithm that can group students based on their source code submission logs and class performance scores, we expected that a clustering approach would provide the best-suited solution to group the students from unlabeled datasets. The most commonly used and effective clustering approaches, such as k-means, k-medoids, DBSCAN, agglomerative hierarchical cluster tree, and other variations of k-means were found based on a review. The MK-means clustering algorithm [78], which

we found to be a robust, scalable, and effective tool, is a variant of the conventional k-means clustering algorithm.

2.2.3 Association Rule Mining Algorithms

ARM algorithm is a USL algorithm used for data mining in big data. ARM was first proposed by Agrawal [28] and has since been used in many fields, such as educational data analysis, medical data analysis, market-basket analysis, and census data. Usually, ARM aims to find a set of cooccurring high-frequency items and extract the correlation among items from large dataset. Although the Apriori algorithm [28] is often used for data mining, many enhancements are proposed based on Apriori to improve performance and scalability, such as the sampling approach [79], hashing technique [80], dynamic counting [81], partitioning technique [82], and incremental mining [83]. Prior studies showed that the Apriori algorithm achieved significant results, but some methods also reported the worse results by generating a large number of candidate item sets, additional scans, etc.

Subsequently, a new algorithm called FP-growth was proposed without the leverage of candidate item set generation [84]. This method used a partitioning-based divide-and-conquer approach. Previous studies have shown that it significantly reduced the search space and time compared to Apriori [85]. Similarly, many extensions are added to the FP-growth algorithm to improve efficiency. Some examples of enhanced FP-growth algorithms are h-mine [86], depth-first mining [87], pattern-growth mining in both directions (bottom-up and top-down), and tree structures [88, 89]. In contrast, Zaki [90] proposed the Equivalence CLASS Transformation (Eclat) algorithm for ARM applied to vertical data. The Eclat uses the same candidate-generation process like Apriori. In brief, Apriori, FP-growth, and Eclat ARM algorithms are most the frequently used in many applications, and also serve as the foundation of many other ARM algorithms. In this Chapter, an FP-growth algorithm is used.

2.2.4 Rule-based Recommendation Systems

The volume and variety of content on e-learning platforms are increasing at an unprecedented rate, and at the same time the opportunities for research using the resources of e-learning platforms are also increasing. Recommending relevant and appropriate content to users (e.g., students, instructors, and teachers) is a challenging and tough task for any e-learning platform. Perumal et al. [33] proposed a novel personalized RS to provide appropriate supportive content

to users. In their approach, FP-growth algorithm is applied to generate frequent items patterns, and fuzzy logic is used to partition the content into three levels. Recently, some RSs have been using a mixed approach of content-based filtering and collaborative filtering to achieve high-quality results in specific contexts [34]. In addition, most RSs are built with a collaborative, knowledge-based, content-based, and hybrid approaches [35]. Conventional e-learning platforms are insufficient to assess exercise-based content such as programming solution codes automatically, instead they can assess exercise-based contents semi-automatically [91]. Thus, usual RS in e-learning platforms have limited suitability for programming and exercise-based education.

2.2.5 Educational Data Mining and Learning Analytics

EDM is the same as traditional data mining, except that it is applied to educational fields. EDM is used to extract hidden knowledge and discover patterns from the data in different educational learning platforms [92]. In the study [92], various data mining techniques including clustering, classification, ARM are exploited to discover useful information from the educational data. They used EDM tools (Rapid Miner and Weka) to analyze data from Moodle in a programming course. Fernandes et al. [93] presented a predictive analysis of students' academic performance. The Gradient Boosting Machine (GBM) classification model was applied to predict students' academic performance at the end of the school year. In another study [94], a semi-supervised learning algorithm was used to predict the students' performance in the final exams. In the study [95], a survey of EDM and its future directions is presented. It also discusses some recent trends in the field of EDM research.

LA has become an important research topic in the field of educational technology. This involves understanding and analyzing real-world educational data to provide useful support for improving learning and teaching. Tran et al. [96] used LA for a learning management system (LMS). The experimental results showed that LA plays an important role in improving productivity, learning, and support for LMS user. Ang et al. [66] discussed LA from five different perspectives: learning and assessment analysis, personalized learning, behavior learning, collaborative and interactive learning and social network analytics. Current LA trends and practices to improve teaching and learning in education are presented in [97, 98].

In addition, numerous studies have been conducted using the resources of OJ or APA systems. These systems are actively used for education, e-learning, computing, programming com-

petitions, and software engineering. The importance of empirical data-driven analysis to make critical decisions, and even to change algorithm configurations automatically, is growing [99]. However, this data-driven analytical research differs from previous research in that a real-world dataset has been used. The analytical findings of this Chapter are beneficial to assist students in improving their academic and practical performance, as well as for educational planning.

2.3 Dataset and Preprocessing

In this section, we introduce the Aizu Online Judge (AOJ) system, which is the source of the submission logs. Moreover, we describe submission logs and class performance scores collected from the AOJ and a programming course, respectively as our datasets. The data types, structure, and preprocessing steps are also presented.

2.3.1 Aizu Online Judge System

The AOJ system [18–20] is a popular OJ platform in Japan and worldwide. It has been running for more than 15 years to host programming competitions, practices, assignments, and education. In addition, the AOJ system is officially employed to conduct programming- and algorithm-related courses at the University of Aizu, Japan. The AOJ’s typical courses include Introduction to Programming I (ITP1), Algorithms and Data Structures I (ALDS1), Introduction to Programming II (ITP2), Datasets and Queries, Discrete Optimization Problems, Graph Algorithms, Computational Geometry, and Number Theory. Thus, plenty of source codes and submission logs are generated on a regular basis. AOJ has a rich repository with approximately 100,000 users, 3,000 problems, and 5.5 million code archives and submission logs. All the problems are systematically categorized [100]. The AOJ’s resources have been used for various research and application purposes [101] [102]. Recently, AOJ’s dataset has been used in the IBM CodeNet Project [103].

2.3.2 Solution Submission logs

In this Chapter, submission logs from a programming course (ALDS1) were collected for experiments. Usually, the problems in the ALDS1 course are assigned to students to solve, as shown in Table 2.1. The overall topic-wise success rate (%) of this course is also mentioned in Table 2.1. The ALDS1 course has 13 topics, and each topic consists of three (03) or four (04)

Table 2.1: Topic-wise problem list of ALDS1 course

| No. | Topic | Theme | Average Success (%) |
|-----|------------------------------|--|---------------------|
| 1 | Getting Started | Insertion Sort Greatest Common Divisor Prime Numbers Maximum Profit | 36.81 |
| 2 | Sort I | Bubble Sort Selection Sort Stable Sort Shell Sort | 44.92 |
| 3 | Elementary Data Structures | Stack Queue Doubly Linked List Application of Stack | 39.57 |
| 4 | Search | Linear Search Binary Search Dictionary Application of Binary Search | 40.10 |
| 5 | Recursion/Divide and Conquer | Exhaustive Search Merge Sort Koch Curve The Number of Inversions | 42.74 |
| 6 | Sort II | Counting Sort Partition Quick Sort Minimum Cost Sort | 43.18 |
| 7 | Tree | Rooted Trees Binary Trees Tree Walk Reconstruction of a Tree | 43.95 |
| 8 | Binary Search Trees | Binary Search Tree-I Binary Search Tree-II Binary Search Tree-III Treap | 54.96 |
| 9 | Heaps | Complete Binary Tree Maximum Heap Priority Queue Heap Sort | 38.43 |
| 10 | Dynamic Programming | Fibonacci Number Matrix Chain Multiplication LCS of Strings | 51.82 |
| 11 | Graph I | Graph Depth First Search Breadth First Search Connected Components | 47.16 |
| 12 | Graph II | Minimum Spanning Tree Single Source Shortest Path I Single Source Shortest Path II | 54.38 |
| 13 | Heuristic Search | 8 Queens Problem 8 Puzzle 15 Puzzle | 45.76 |

problems which we call problems A , B , C , and D .

Table 2.2: Sample submission logs generated by the AOJ system

| UID | JID | P | Verd | Lang | CPU time (1/100 s) | Memory usage (KB) | Code size (Byte) | Submission date (ms) |
|-------|---------|----------|------|---------|--------------------|-------------------|------------------|----------------------|
| u_1 | 3573821 | p_1 | RTE | Python3 | 0 | 0 | 71 | 1557999872496 |
| u_2 | 3574251 | p_2 | AC | C++11 | 0 | 3444 | 1885 | 1558007384660 |
| u_3 | 3573537 | p_3 | CE | C++ | 1 | 3404 | 1684 | 1557998419203 |
| u_4 | 3556699 | p_4 | WA | C++ | 2 | 3104 | 1708 | 1557482997985 |
| u_5 | 3536901 | p_5 | TLE | C++ | 399 | 3280 | 2199 | 1556795320402 |
| u_4 | 3383318 | p_{10} | AC | C | 29 | 5748 | 936 | 1550129425882 |

The logs are generated by the AOJ system based on the submitted solution codes by the students over two semesters, the size of the submission logs is approximately 69,000. Each solution log has a set of information, such as the judge id (jid), user id (uid), problem id (pid), language (C, C++, python, etc.), accuracy, verdict (accepted, wrong answer, compile error, etc.), CPU time, memory usage, code size, submission date, and judge date. Let UID be a set of users (students) i.e., $UID = \{uid_1, uid_2, \dots, uid_n\}$, $n \geq 1$. JID is a set of judge IDs $JID = \{jid_1, jid_2, \dots, jid_m\}$, $m \geq 1$; $Prob$ is a set of problems $Prob = \{prob_1, prob_2, \dots, prob_k\}$, $k \geq 1$ where $prob_1, prob_2, \dots, prob_k$ are unique problems; judge verdicts are $Verd = \{AC, WA, CE, RTE, MLE, TLE, OLE, PrE\}$ where AC = Accepted, WA = Wrong Answer, CE = Compile Error, RTE = Run Time Error, TLE = Time Limit Exceeded, OLE = Output Limit Exceeded, MLE = Memory Limit Exceeded, and PrE = Presentation Error; and the programming languages are $Lang = \{C, C++, C++11, Ruby, Python 2, Python 3, Java, Haskell, C\#, PHP, Rust, \dots\}$. A corresponding submission log is created immediately after submitting a solution code to the AOJ system. Thus, a sample output log of AOJ system can be written as $O_{logs} = \{u_r, j_s, p_t, v_u, l_v, ct, mu, cs, sd, jd\}$, where $u_r \in UID$, $j_s \in JID$, $p_t \in Prob$, $v_u \in Verd$, $l_v \in Lang$, ct = CPU time, mu = Memory usage, cs = Code size, sd = Submission date, jd = Judge date. Some sample logs generated by the AOJ system are listed in Table 2.2.

2.3.3 Class Performance Scores

In addition to the submission logs, we collected various test (exam) scores for the ALDS1 course from 357 students in two different years at the University of Aizu, Japan. Usually, most students take the ALDS1 course as part of their regular study. This course consists of vari-

ous tests, such as algorithm assignment (AA), programming assignment (PA), code validation ($CVal$), coding test (CoT), and paper-based test (PT). Note that PA and $CVal$ are calculated based on the student's program submission to the AOJ system. To check the plagiarism/similarity/duplication of submitted solution codes, a plagiarism checking software (PCS) has been developed and integrated into the management system of AOJ. The PCS checks solution codes submitted by the students against the existing source codes in the AOJ. The PCS generates a $CVal$ score for submitted codes based on the degree of similarity, and the codes are collected from a specific time period and users. A score of 1 means that there is no copying/duplication, 0.5 means that a few codes are copied from others, and 0 means that a number of codes are copied from others. In addition, $CVal$ is used to justify the scores of PA . Some sample data distribution of student evaluations are listed in Table 2.3.

Table 2.3: Sample data distribution of student evaluations

| UID | AT | AA | PA | CVal | PT | CoT | T | Prac |
|-------|----|----|-----|------|----|-----|------|-------|
| u_1 | 12 | 85 | 90 | 1 | 80 | 75 | 82.5 | 82.2 |
| u_2 | 10 | 75 | 85 | 0.5 | 85 | 70 | 79.8 | 38.6 |
| u_3 | 11 | 80 | 100 | 1 | 75 | 90 | 77.5 | 94.9 |
| u_4 | 13 | 90 | 110 | 0.5 | 90 | 110 | 90.0 | 55.0 |
| u_5 | 9 | 65 | 70 | 0.5 | 75 | 60 | 69.8 | 32.4 |
| u_6 | 13 | 95 | 105 | 1 | 90 | 100 | 92.5 | 102.5 |
| u_7 | 10 | 78 | 85 | 0 | 60 | 70 | 68.4 | 0.0 |

Definition 2 The $CVal$ score refers to the degree/level of program code plagiarism.

Example 1 If a user u_{15} copies/replicates programs from others, then u_{15} receives a $CVal$ score of 0.5; if user u_{15} copies/replicates the code from others with malicious intent, $CVal$ is 0.

Each exercise class is divided into two parts. First, students are asked to submit an AA , which consists of a few questions and is also considered student attendance (AT). Second, three or four problems are given to the students as a PA . The students are then encouraged to submit their solutions through the AOJ system. Students are allowed to consult with each other, teachers, and teaching assistants to solve problems during PA . In contrast, CoT is conducted in exercise rooms with a separate workstation for each student, providing a process by which each student's actual programming capabilities can be verified. Note that it is strictly forbidden for a student to consult with other students during the CoT . Similarly, the PT is a closed-book test that is given to check the true level of each student's theoretical understanding. The test scores distribution can be expressed as $T_{score} = \{UID, AT, AA, PA, CVal, PT, CoT, T, Prac\}$,

where $AT \in \mathbb{N}$ and $0 \leq AT \leq 13$, $AA \in \mathbb{N}$ and $0 \leq AA \leq 100$, $PA \in \mathbb{N}$ and $0 \leq PA \leq 120$, $CVal \in \mathbb{R}$ and $0 \leq CVal \leq 1$, $PT \in \mathbb{N}$ and $0 \leq PT \leq 120$, $CoT \in \mathbb{N}$ and $0 \leq CoT \leq 120$, $T \in \mathbb{N}$ and $0 \leq T \leq 120$, $Prac \in \mathbb{N}$ and $0 \leq Prac \leq 120$. To better evaluate the students of the ALDS1 course by considering the importance of theoretical and practical knowledge, the equations (2.1), (2.2), and (2.3) are developed for the Theory (T), Practical ($Prac$) and Final Score (FS) calculations, respectively, based on the different test scores. Note that the equations for the ALDS1 course are approved by the course coordinator.

$$T = \sqrt{AA \times PT} \quad (2.1)$$

$$Prac = \sqrt{(PA \times CoT)} \times CVal \quad (2.2)$$

$$FS = \min(100, \lfloor \frac{AT+1}{10} \rfloor \times \sqrt{(T \times Prac)}) \quad (2.3)$$

For explanation, the student evaluation process is compared using the following two scenarios: (i) the conventional case and (ii) the proposed case (based on the equations).

Conventional Case: In this case, the final results are usually generated by averaging $Prac$ and T scores. For example, if student $s1$ gets 10 points on the $Prac$ test and 90 points on the T test, the final result of $s1$ using the conventional method is $(10 + 90)/2 = 50$.

Proposed Case: In this case, $Prac$ and T scores are given equal priority to generate final results, so the equations (2.1 – 2.3) are introduced to emphasize both the $Prac$ and T scores. Let us assume that if student $s1$ gets 10 points on the $Prac$ test and 90 points on the T test, the final result of $s1$ using our equations will be $\sqrt{10 \times 90} = 30$. We observed that the proposed evaluation method considers both the $Prac$ and T scores, although there is no balance between the $Prac$ and T scores when calculating the final result using the conventional method.

For statistical feature extraction and ARM, Tables 2.2 and 2.3 are joined ($O_{logs} \bowtie T_{scores}$) to produce the operational data, as shown in Table 2.4. In addition to the existing attributes, a new attribute (*Accuracy*) has been added to the operational data.

Definition 3 *The number of accepted solutions out of total submissions is called the solution Accuracy of users.*

$$Accuracy(Accu) = \frac{\sum_{i=1}^{p_n} TAS_i}{\sum_{i=1}^{p_n} TS_i} \quad (2.4)$$

where p_n = number of problems, TAS = total accepted solutions, and TS = total submissions.

Table 2.4: Sample operational data distributions by joining submission logs (Table 2.2) and evaluation scores (Table 2.3)

| UID | P | Verd | Accu (%) | mu | cs | CVal | PA | CoT | PT |
|-------|-------|------|----------|------|-----|------|-----|-----|----|
| u_1 | p_1 | AC | 70 | 1164 | 116 | 1 | 90 | 75 | 80 |
| u_2 | p_2 | WA | 65 | 1072 | 125 | 0.5 | 85 | 70 | 85 |
| u_3 | p_7 | RTE | 84 | 1124 | 239 | 0.5 | 100 | 90 | 75 |
| u_1 | p_2 | TLE | 70 | 1064 | 96 | 1 | 90 | 75 | 80 |
| u_5 | p_1 | AC | 88 | 1143 | 209 | 0.5 | 70 | 60 | 75 |

Example 2 Let u_5 be a user who has submitted a total of 39 solutions to the AOJ system, of which, a total of 28 have been accepted. Then, the solution accuracy of the user u_5 is $(28/39) = 71\%$, according to (2.4).

Another important term is trial and error ($T\&E$), we use the $T\&E$ method to estimate a programmer's ability to solve problems. In this Chapter, the following definition is adopted for the $T\&E$ method.

Definition 4 A number of repeated attempts are taken until a problem is successfully solved; this process is called trial and error ($T\&E$).

$$T\&E = \frac{\sum_{j=1}^{p_n} TS_j}{\sum_{j=1}^{p_n} T AS_j} \quad (2.5)$$

where p_n = number of problems, TS = total submissions, and TAS = total accepted solutions.

Example 3 Suppose that u_{10} is a user who has received a total of 25 accepted (AC) verdicts from the AOJ for 5 problems, but has taken a total of 129 attempts ($T\&E$) to achieve it. Then, the average $T\&E$ of user u_{10} for each solved problem is $(129/25) = 5.16$, according to (2.5).

2.4 Approach

Figure 2.1 shows an overview of the proposed framework of the data-driven approach. We employed the framework to a real-world dataset to extract the hidden features and association rules of students to explore the importance of practical skills. Experimental data are collected from AOJ system and ALDS1 programming course, respectively. The proposed approach consists of four main steps: (i) data collections and preprocessing, (ii) data clustering, (iii) statistical hidden features extraction from clusters, and (iv) patterns and association rules mining from clusters. A MK-means clustering algorithm is applied for data clustering, where the elbow

method used to select the optimal k values for the MK-means. Furthermore, the FP-growth ARM algorithm is leveraged to extract the association rules from each cluster. The methods and algorithms used for the proposed approach are discussed below.

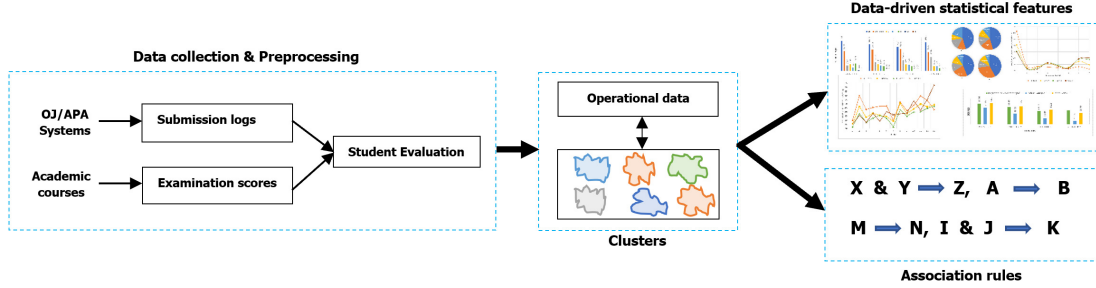


Figure 2.1: The overall framework of the data-driven approach

2.4.1 Elbow Method

The elbow method is a proven technique to determine the optimal number of clusters k for the k-means algorithm. It uses the sum of squared errors (SSE) of each cluster to calculate the optimal number of clusters. The SSE is calculated by the equation (2.6).

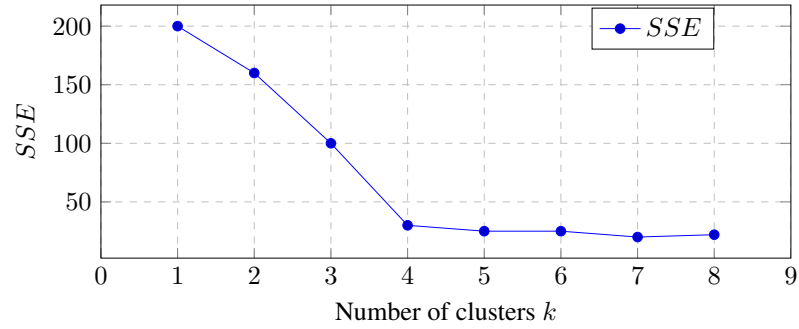
$$SSE = \sum_{i=1}^k \sum_{x \in C_i} dist^2(m_i, x) \quad (2.6)$$

where k is a number of clusters, x is a data point in cluster C_i , and m_i is the center of cluster C_i .

The elbow method reduces unnecessary clustering in the dataset, where a small SSE value indicates a better cluster. Normally, increasing the value of k automatically decreases the SSE value. When the SSE value is drastically decreased, that point is caught as the ideal number (k) of clusters for k-means. The elbow method was applied to our dataset to obtain the optimal k value ($k = 4$) for the MK-means clustering algorithm, as shown in Figure 2.2.

2.4.2 Modified K-means Clustering Algorithm

Usually, the k-means clustering algorithm randomly chooses the initial centroid, so it is possible to select an irrelevant data point as the initial centroid. In addition, conventional k-means algorithms cannot detect and remove outliers from the dataset. Consequently, the results may have a negative impact on the overall clustering process and results. To address these problems, the MK-means clustering algorithm [78] integrates two important modules, (i) optimal initial

Figure 2.2: Elbow method for optimal k selection

centroid selection and (ii) outlier detection and removal.

Algorithm 1 Initial Centroid Selection Module (ICSM)

Define: Distance: D , Origin: $O(0, 0)$, Cluster Number: K

Input: Dataset: $X = \{x_1, x_2, x_3, \dots, x_n\}$

Output: Optimal initial centroids $C_n = []$

```

for  $x_j \in X$  do
  |  $D \leftarrow distance(x_j, O)$ 
end
for  $d_i \in D$  do
  | Apply sorting on  $D$ 
  |  $D \leftarrow d_1, d_2, d_3, \dots, d_n$ 
end
if  $K \leq |X|$  then
  | Divide sorted data  $D$  into  $K$  subsets
  |  $s_1 \subseteq D, s_2 \subseteq D, s_3 \subseteq D, s_4 \subseteq D, \dots, s_k \subseteq D$ 
end
while  $k \leq K$  do
  | Calculate Mean value of each subset
  |  $M_k = \frac{\sum_{x \in S_k} x}{|S_k|}$ 
  | for  $x_j \in S_k$  do
  | |  $C_n \leftarrow mindistance(M_k, x_{j \in S_k})$ 
  | end
end

```

To the best of our knowledge, this is a unique modification of the k-means clustering algorithm, and these two modules makes the algorithm more efficient, robust, and scalable. The MK-means algorithm takes approximately 17.33% fewer iterations to construct clusters for our dataset than other random initial centroid-selection algorithms. The first module is initial centroid selection module (ICSM) which leverages to (i) select optimal centroids and (ii) build clusters with the most similar data. The pseudocode of ICSM is provided in Algorithm 1.

The second module is the outlier detection module (ODM), which is used to (i) detect outliers (irrelevant/insignificant data point), (ii) remove them from the datasets, and (iii) improve the overall cluster quality. The pseudocode of the ODM is presented in Algorithm 2. More-

over, the MK-means algorithm showed the effectiveness of clustering multidimensional data in Euclidean space.

Algorithm 2 Outlier Detection Module (ODM)

Define: Number of Cluster: K , Cluster: C

Input: Dataset: $X = \{x_1, x_2, x_3, \dots, x_n\}$, Distance: D

Output: Outliers: O , SSE

Run ICSM and Calculate **min-max average (MMA)** using *sorted* distance $d \in D$

$$MMA = \frac{d_{min} + d_{max}}{2}$$

while $k \leq K$ **do**

for $x_i \in X_{C_k}$ **do**

if $distance(x_i, center_{C_k}) > MMA$ **then**

Remove x_i from the cluster C_k

$O \leftarrow x_i$

 Recalculate SSE

end

end

end

2.4.2.1 Multidimensional Data Clustering in Euclidean Space

The growing collection of structured, unstructured, and multidimensional data in a variety of media has created challenges in the field of data science. However, clustering of multidimensional data is a tricky and challenging task. Conventional clustering algorithms have limitations in clustering multidimensional data. To alleviate this problem, the MK-means clustering algorithm can efficiently handle multidimensional data (numerical data) in Euclidean space for clustering. Table 2.5 shows an example of a four-dimensional (A_1, A_2, A_3, A_4) data set. According to the ICSM module of the MK-means clustering algorithm, the distance from the origin (O) is calculated for each data point. Here we describe a mathematical procedure for calculating the distance of multidimensional data points.

Table 2.5: An example of 4-dimensional dataset

| Data Points | A_1 | A_2 | A_3 | A_4 | Distance |
|-------------|-------|-------|-------|-------|----------|
| z_1 | 14 | 5 | 9 | 12 | 21.12 |
| z_2 | 7 | 10 | 11 | 21 | 26.66 |
| z_3 | 15 | 9 | 6 | 10 | 21.02 |
| z_4 | 20 | 28 | 17 | 13 | 40.52 |
| z_5 | 18 | 24 | 19 | 15 | 38.55 |

The mathematical model for multidimensional data normalization for clustering by ICSM (Algorithm 1) is as follows:

Let $Z = \{A_1, A_2, A_3, \dots, A_n\}$ and $O = \{O_1, O_2, O_3, \dots, O_n\}$ be the multidimensional

data point and the origin, respectively, where n is the number of dimensions. Now, the distance between Z and O can be expressed by Equation 2.7.

$$D(O, Z) = \sqrt{(0_1 - A_1)^2 + (0_2 - A_2)^2 + \cdots + (0_n - A_n)^2} \quad (2.7)$$

The generalized formula for the distance calculation between multidimensional data points and the origin is shown in Equation 2.8.

$$\text{Distance } D(O, Z) = \sqrt{\sum_{i=1}^N (O - Z_{A_i})^2} \quad (2.8)$$

where N is the number of dimensions, O is the origin, and Z_{A_i} is the data in each dimension of a data point.

Here we calculated the distance according to Equation 2.8 for each data point (in Table 2.5) as follows:

$$\begin{aligned} d_{z_1} &= \sqrt{(0 - 14)^2 + (0 - 5)^2 + (0 - 9)^2 + (0 - 12)^2} \\ &= 21.12 \end{aligned}$$

$$\begin{aligned} d_{z_2} &= \sqrt{(0 - 7)^2 + (0 - 10)^2 + (0 - 11)^2 + (0 - 21)^2} \\ &= 26.66 \end{aligned}$$

$$\begin{aligned} d_{z_3} &= \sqrt{(0 - 15)^2 + (0 - 9)^2 + (0 - 6)^2 + (0 - 10)^2} \\ &= 21.02 \end{aligned}$$

$$\begin{aligned} d_{z_4} &= \sqrt{(0 - 20)^2 + (0 - 28)^2 + (0 - 17)^2 + (0 - 13)^2} \\ &= 40.52 \end{aligned}$$

$$\begin{aligned} d_{z_5} &= \sqrt{(0 - 18)^2 + (0 - 24)^2 + (0 - 19)^2 + (0 - 15)^2} \\ &= 38.55 \end{aligned}$$

Furthermore, as per the ICSM module (Algorithm 1), the data points are sorted (in ascend-

ing or descending order) based on their calculated distance values. Subsequently, the data points are partitioned according to the specified k values. In addition, optimal initial centers are selected and other clustering tasks are performed. In the clustering process, ODM is another effective module (Algorithm 2) that can detect and remove the most irrelevant data points from the dataset. In this way, the probability of selecting irrelevant data points as optimal initial centers is reduced. Figure 2.3 shows an example of two-dimensional data distribution in Euclidean space, where the distance of the data points from the origin can be calculated.

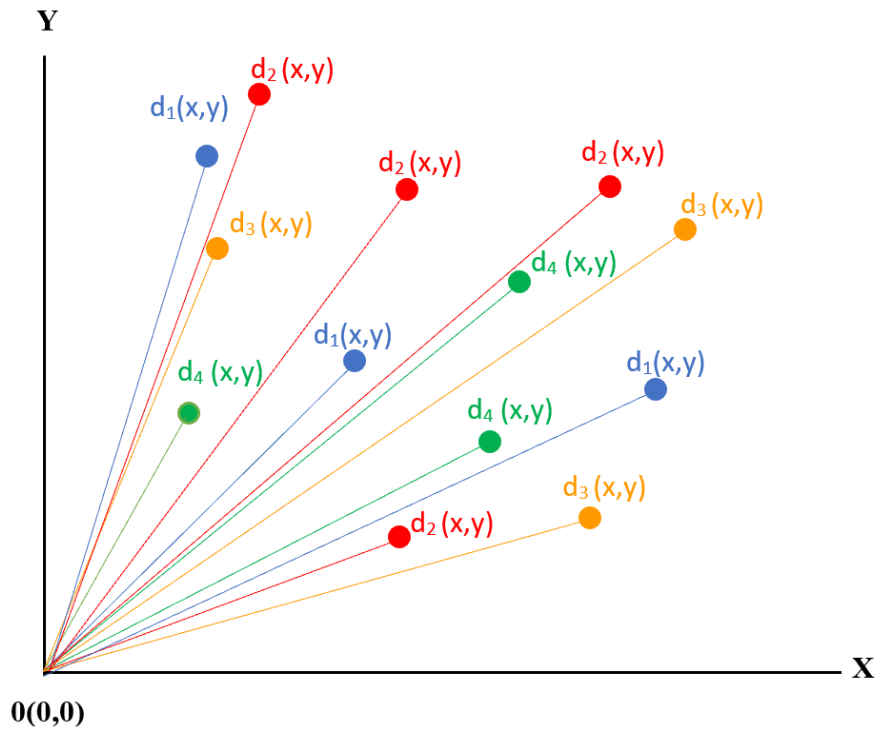


Figure 2.3: An example of two-dimensional data distribution in Euclidean space

2.4.3 FP-growth Algorithm

In the field of data mining, the Apriori, Eclat, and FP-growth algorithms are the most commonly used [32]. The FP-growth algorithm is much more efficient and faster than Apriori because the Apriori algorithm repeatedly scans the database, whereas the FP-growth algorithm only scans twice to complete the process. The FP-growth algorithm [84] basically consists of two (2) main steps, namely (i) construction of the FP-tree and (ii) FP mining based on the FP-tree. Let $L = \{l_1, l_2, l_3, l_4, \dots, l_d\}$ be the set of all items in the database. The databases are built based on a set of tuples/transactions $T = \{t_1, t_2, t_3, t_4, \dots, t_N\}$, where each transaction t_i is a subset of $L (t_i \subseteq L)$. The formula of an association rule can be written as $R = X \rightarrow Y$,

where X, Y is a subset of L ($X \subseteq L, Y \subseteq L$) and $X \cap Y = \phi$. The set of items in X is often called the preceding (*if*), and the set of items in Y is called the subsequent (*then*). Mathematically, the support count for item set X is expressed as $\varsigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$, where $|\cdot|$ denotes the number of elements in a set. The minimum support (*minSup*) and minimum confidence (*minConf*) are two important terms that are used to create association rules and patterns. The *minSup* threshold is used to find item frequencies in a database, whereas the *minConf* threshold value is applied to these frequent items to construct the association rules. The support (*Sup*) and confidence (*Con*) are represented by the equations (2.9) and (2.10), respectively.

$$Sup(X \longrightarrow Y) = \frac{\varsigma(X \cup Y)}{N} \quad (2.9)$$

$$Con(X \longrightarrow Y) = \frac{\varsigma(X \cup Y)}{\varsigma(X)} \quad (2.10)$$

where N = total number of transactions.

2.5 Experimental Results

In this section, the experimental results are presented. We first cluster students based on their submission logs and scores, and then extracted the hidden features from each cluster. The association rules are generated from each cluster using the FP-growth algorithm to validate the features. Finally, all the correlated features are accumulated for discussion.

2.5.1 Clustering the Data

According to the proposed framework (Figure 2.1), the MK-means clustering algorithm is applied to the Table 2.3 for the clustering process. Before clustering begins, the elbow method is applied to the same data to generate the optimal number ($k = 4$) of clusters, as shown in Figure 2.2. Now, four clusters have been formed, named clusters $P, Q, R,$ and S . Note that multidimensional data (Table 2.3) are clustered. To visualize the data distribution of each cluster, we applied principal component analysis (PCA) technique [104] to multidimensional clustered data to convert it into a two-dimensional (2D) shape. For this reason, the first two components (PCA 1 and PCA 2) of the PCA that explain the majority of the variance in the data are used for the 2D visualization. The visualized clusters are shown in Figure 2.4.

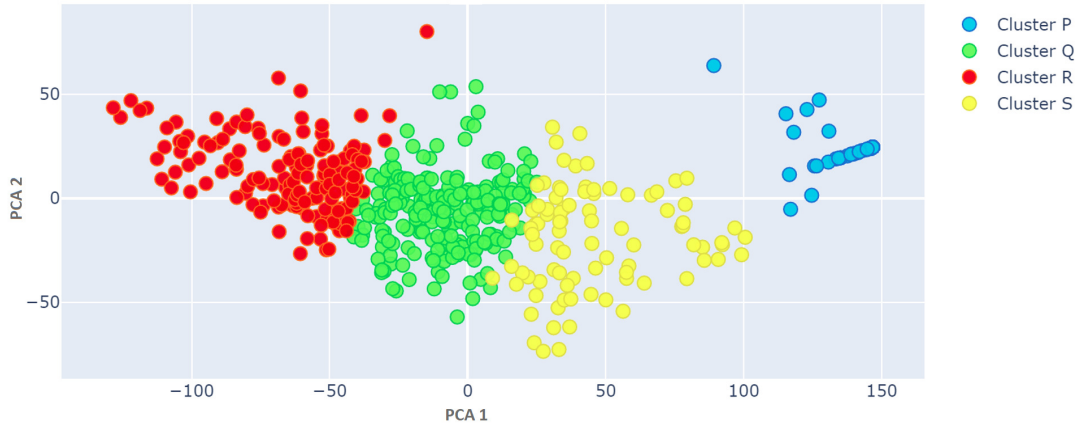


Figure 2.4: 2D visualization of the clusters

First, the preliminary statistical information related to each cluster, (i) the number of students per cluster and (ii) the total number of problems solved by the students in each cluster, is presented in Table 2.6. We found that approximately 33.33% of the students are in cluster Q , which is the largest, and approximately 16.01% of the students are in cluster P , which is the smallest. On the other hand, the students in cluster Q generated the largest submission log of 22,110, and the students in cluster S produced the smallest submission log of 5,153.

Table 2.6: Preliminary statistical information of each cluster

| Cluster | Submission logs | Students (%) |
|---------|-----------------|--------------|
| P | 13099 | 16.01 |
| Q | 22110 | 33.33 |
| R | 17274 | 32.02 |
| S | 5153 | 18.62 |

2.5.2 Extracting Hidden Features

In this section, different features of students are extracted from clusters P , Q , R , and S . We calculated the solution verdicts (considering problems A , B , C , and D) in each cluster, as denoted in Table 2.7. Each submission log contains at least one judge verdict out of many (AC , WA , CE , etc.). Therefore, each verdict determined the ultimate result of a submitted solution. A few observations can be illustrated from the Table 2.7: (i) clusters P and S have the highest AC rates, (ii) the students of cluster R achieved the lowest AC rates, and (iii) the students of cluster R received higher error verdicts than those in other clusters.

Also, we enumerated problem-wise statistics of the submitted solutions to find out how many submissions belong to each problem such as A , B , C , and D in each cluster, as presented

Table 2.7: Overview of the judge verdicts of ALDS1 course

| Verdict | Cluster <i>P</i> | Cluster <i>Q</i> | Cluster <i>R</i> | Cluster <i>S</i> |
|---------|------------------|------------------|------------------|------------------|
| AC (%) | 40.88 | 36.34 | 32.70 | 39.19 |
| WA (%) | 26.71 | 28.92 | 29.46 | 25.69 |
| CE (%) | 7.37 | 11.95 | 14.88 | 18.59 |
| RTE (%) | 8.81 | 8.80 | 8.71 | 6.29 |
| PrE (%) | 4.76 | 7.02 | 7.94 | 6.46 |
| TLE (%) | 10.19 | 6.48 | 6.11 | 3.80 |
| MLE (%) | 1.20 | 0.44 | 0.16 | 0.00 |
| OLE (%) | 0.03 | 0.04 | 0.04 | 0.00 |

in Table 2.8. A few observations can be drawn from the Table 2.8: (i) Students of cluster *P* submitted the fewest solutions to problem *A*, at 30.99%, compared to clusters *Q*, *R*, and *S*. (ii) Cluster *P* students submitted the highest number of solutions for problems *C* and *D*, at 32.82% and 10.55%, respectively, compared to clusters *Q*, *R*, and *S*. (iii) Students of clusters *R* and *S* submitted the fewest solutions for problems *C* and *D*, compared to clusters *P* and *Q*, respectively.

Table 2.8: Overview of the submission statistics for each type of problem

| Problem | Cluster <i>P</i> | Cluster <i>Q</i> | Cluster <i>R</i> | Cluster <i>S</i> |
|----------|------------------|------------------|------------------|------------------|
| <i>A</i> | 30.99% | 39.17% | 48.73% | 49.19% |
| <i>B</i> | 25.65% | 30.28% | 32.19% | 29.09% |
| <i>C</i> | 32.82% | 25.04% | 16.02% | 18.11% |
| <i>D</i> | 10.55% | 5.51% | 3.06% | 3.61% |

In contrast, the error verdicts of each cluster are also calculated. The segmentation of error verdicts received by the students in each cluster are shown in Figure 2.5. For that, the error verdicts are divided into five (05) categories based on the error types in codes such as (i) *WA*, (ii) *CE*, (iii) *RTE*, (iv) *PrE*, and (v) Resource Limitation (*TLE*, *MLE*, *OLE*) i.e., *RL*. Detailed error statistics for each cluster are presented in Figure 2.5.

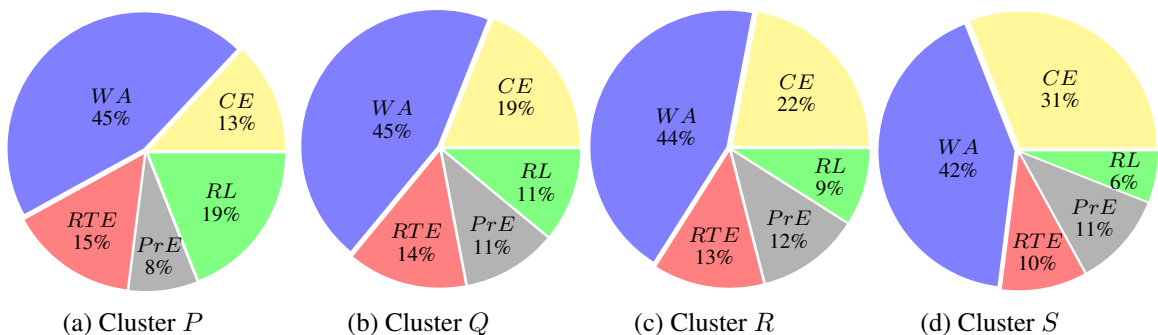


Figure 2.5: Segmentation of error verdicts received by the students

Now, the solution *accuracy* and *T&E* are calculated for each cluster, as enumerated in Table 2.9. A few observations can be drawn: (i) the students of clusters *P* and *Q* have more *T&E* as well as higher solution *accuracy*, and (ii) both the solution *accuracy* and *T&E* of cluster *R* are lower than those of other clusters. Note that the solution *accuracy* and *T&E* are calculated by equations (2.4) and (2.5), respectively.

Table 2.9: Cluster-wise solution accuracy and problem-solving *T&E*

| Cluster | Avg. solution accuracy (%) | Avg. T&E |
|----------|----------------------------|----------|
| <i>P</i> | 55.71 | 12.40 |
| <i>Q</i> | 48.45 | 10.14 |
| <i>R</i> | 43.15 | 9.52 |
| <i>S</i> | 45.18 | 9.86 |

Next, the average score and standard deviation (σ) for each cluster are calculated, as shown in Table 2.10 and the comparative views presented in Figure 2.6. Standard deviation (σ) is used to measure the variation of values in a cluster. Thus, a low value of σ indicates that the values are likely close to the mean (average). The following observations can be drawn: (i) the *CoT*, *PA*, and *PT* scores of cluster *P* are much higher than those of clusters *Q*, *R*, and *S*; (ii) the *PA*, *CoT*, and *PT* scores of cluster *Q* are higher than those of clusters *R* and *S*; (iii) the *CoT* score of cluster *R* is comparatively lower than the other *PA* and *PT* scores of this cluster; (iv) the *CoT* score of cluster *S* is also much lower than those of clusters *P*, *Q*, and *R*.

Table 2.10: Overview of the average scores and standard deviation (σ) in each cluster

| Cluster | PA | σ -PA | CoT | σ -CoT | PT | σ -PT |
|----------|-------|--------------|-------|---------------|-------|--------------|
| <i>P</i> | 97.46 | 13.95 | 78.55 | 16.70 | 98.79 | 11.67 |
| <i>Q</i> | 81.43 | 15.97 | 48.21 | 12.61 | 84.51 | 15.15 |
| <i>R</i> | 60.92 | 17.80 | 28.46 | 10.68 | 68.26 | 16.60 |
| <i>S</i> | 65.19 | 34.96 | 16.55 | 14.48 | 53.79 | 30.05 |

We found more interesting features from the clusters. For example, students solved numerous additional problems beyond their regular exercise assignments through the AOJ platform, solely for their own interests and amusement. The cluster-wise extra problem solution statistics are listed in Table 2.11. The following observations can be drawn from the Table 2.11: (i) the students of cluster *P* solved a huge number of problems beyond their regular exercise assignments, which clearly indicates their enthusiasm for programming, and (ii) the students in other clusters (*Q*, *R*, and *S*) did not solve a significant number of extra problems.

The tendency to submit each assignment in the ALDS1 course is analyzed for more information. There are a few rules to submit each *PA* task through the AOJ platform: (i) problems

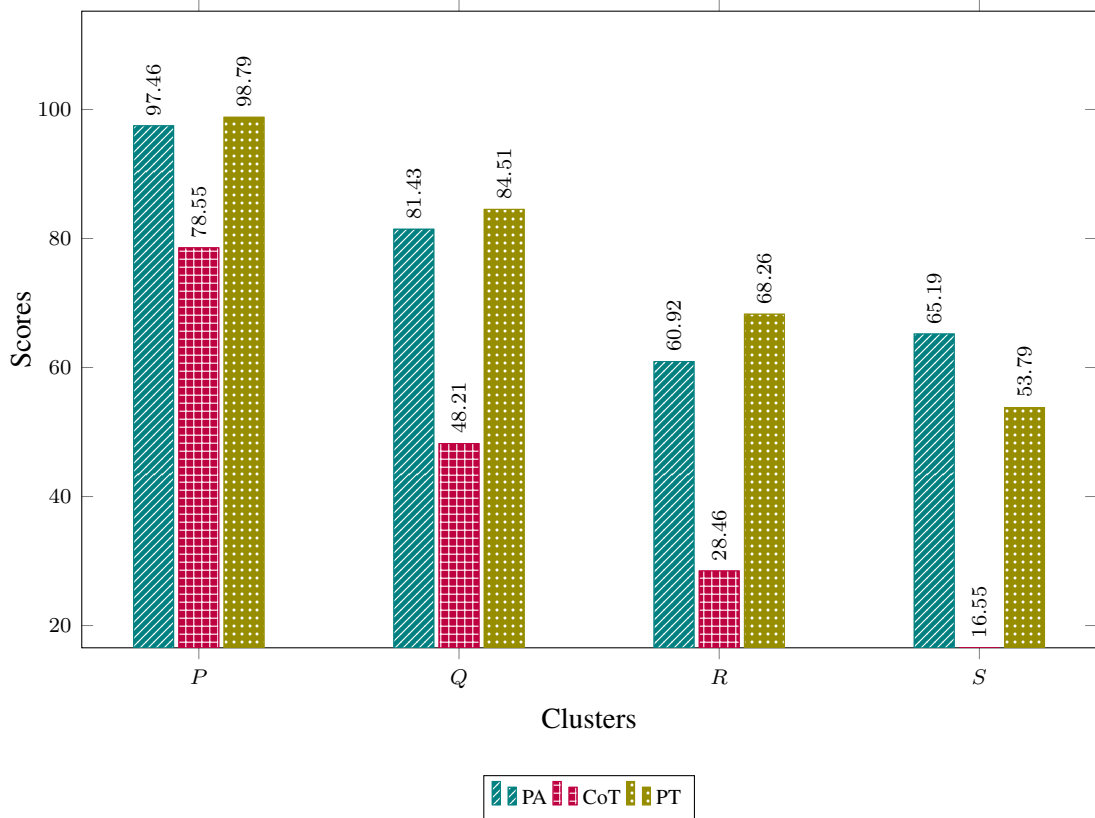


Figure 2.6: Comparison of scores in different tests

Table 2.11: Statistics of extra problem solutions

| Cluster | Average number of extra problem solutions |
|----------|---|
| <i>P</i> | 56.14 |
| <i>Q</i> | 0.68 |
| <i>R</i> | 2.31 |
| <i>S</i> | 0.00 |

A and *B* must be solved by a certain predetermined deadline, where students usually have eight (08) days to submit each assignment, and (ii) problems *C* and *D* can be submitted by the end of the semester. One of our goals is to observe students' submission trends for each topic, how they submitted solutions to problems *A* and *B* within the allotted time, because problems *A* and *B* are mandatory for scoring. The average submission trend among all clusters over a period of time (08 days) is shown in Figure 2.7. The following observations can be illustrated from the Figure 2.7: (i) the students of cluster *P* tried very hard to solve and submit their assignments (problems *A* and *B*) on the very first day of the submission period, (ii) the students of clusters *R* and *S* made less effort in submitting assignments during the first few days of the submission period compared to clusters *P* and *Q*, and (iii) more students from clusters *R* and *S* submitted their assignments on the last day (8th) of the submission deadline than students from clusters *P*

and Q .

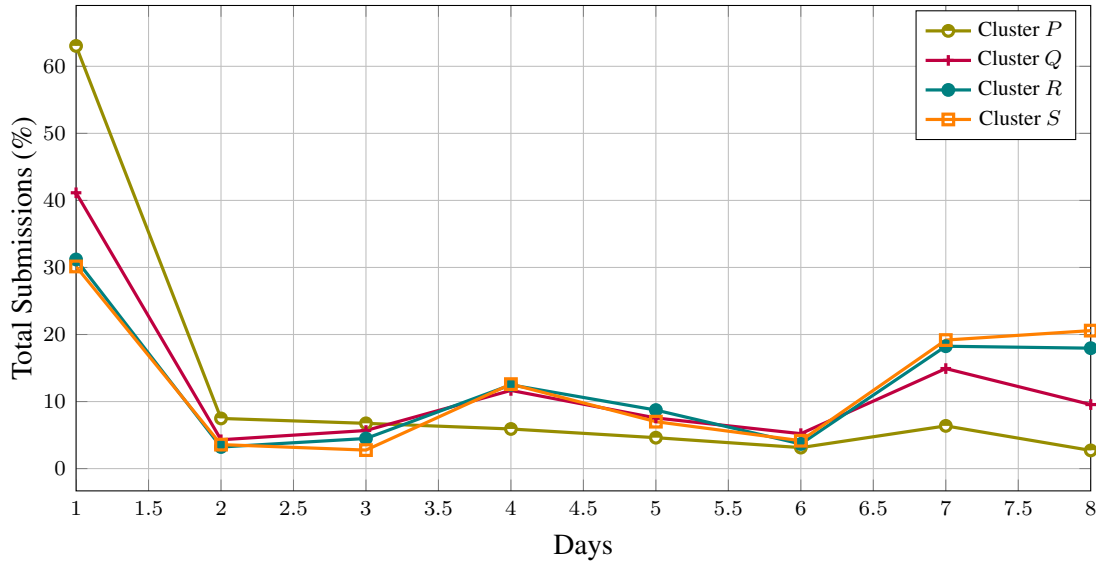


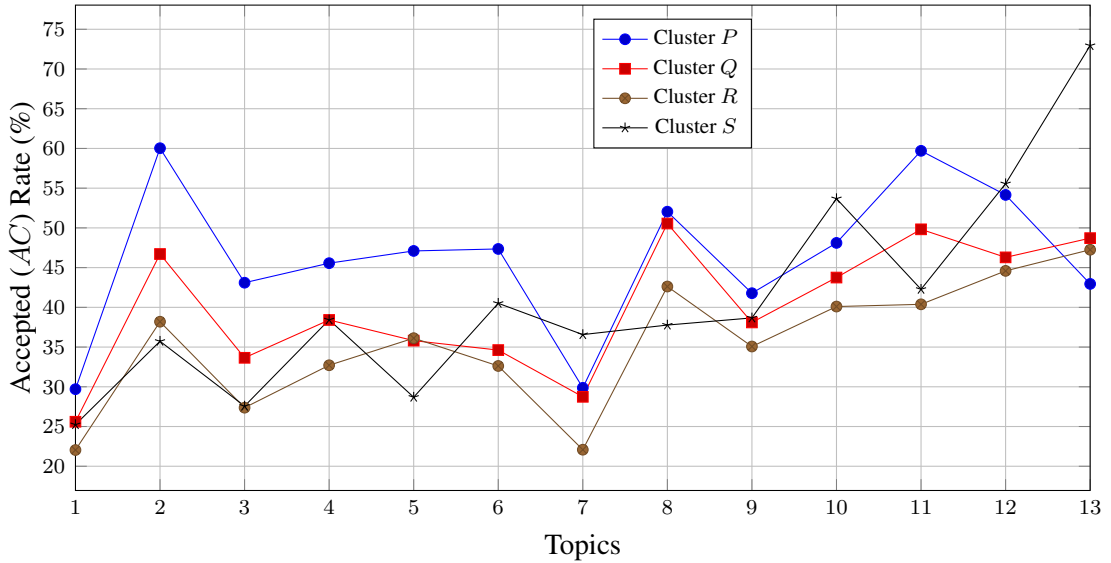
Figure 2.7: Tendency to submit assignments within the allotted period (08 days)

Sometimes students submitted their solutions after the deadlines. The topic-wise accepted (AC) solution rate and average accepted (AC) rate for all clusters are calculated and listed in Table 2.12. Moreover, a visual comparison between all topics for all clusters is presented in Figure 2.8. A few observations can be found: (i) the students of cluster P received the highest acceptance against all their assignment submissions and (ii) the students of cluster R obtained the lowest acceptance rate compared to those in clusters P , Q , and S .

Table 2.12: Topic-wise average accepted (AC) solution rate

| Topic | Accepted Solution (%) | | | |
|----------------|-----------------------|--------------|--------------|--------------|
| | Cluster P | Cluster Q | Cluster R | Cluster S |
| 1 | 29.70 | 25.60 | 22.04 | 25.24 |
| 2 | 60.03 | 46.71 | 38.19 | 35.73 |
| 3 | 43.10 | 33.65 | 27.37 | 27.57 |
| 4 | 45.56 | 38.41 | 32.71 | 38.43 |
| 5 | 47.10 | 35.81 | 36.12 | 28.69 |
| 6 | 47.35 | 34.62 | 32.62 | 40.51 |
| 7 | 29.87 | 28.74 | 22.08 | 36.57 |
| 8 | 52.04 | 50.55 | 42.62 | 37.77 |
| 9 | 41.78 | 38.10 | 35.07 | 38.67 |
| 10 | 48.11 | 43.75 | 40.10 | 53.68 |
| 11 | 59.70 | 49.82 | 40.38 | 42.27 |
| 12 | 54.15 | 46.29 | 44.59 | 55.56 |
| 13 | 42.95 | 48.71 | 47.25 | 72.95 |
| Average | 46.24 | 40.06 | 35.47 | 41.05 |

We also analyzed the data across all clusters to find the assignment submission trends on the

Figure 2.8: Comparison of topic-wise accepted (AC) rate

last (8^{th}) day of the allotted time. A comparative analysis of the tendency to submit assignments on the last day of each topic is presented in Figure 2.9. The average submission rate on the last day is also calculated across all topics, as shown in Table 2.13.

It can be observed that among all clusters, (*i*) students of clusters R and S submitted most solutions on the last day and (*ii*) students of cluster P submitted the fewest solutions on the last day.

Table 2.13: Average submission rate on the last day

| Cluster | Average submission rate (%) |
|---------|-----------------------------|
| P | 2.90 |
| Q | 10.28 |
| R | 20.06 |
| S | 22.22 |

To obtain more interesting hidden features that are not plainly visible in the dataset, we analyzed the data of each cluster and found that many students repeatedly solved problems (already accepted) for optimization in terms of memory usage, CPU time, code refactoring, etc. The repetition tendency (only for accepted problems) of students in each cluster is calculated. The ALDS1 course has thirteen topics, each with four problems (A , B , C and D), for a total of 52 unique problems (say, total problem $TP = 52$). We determined how many students in each cluster repeatedly solved 25%, 50%, and 75% of the TP , as enumerated in Table 2.14.

The students participation (maximum and minimum) from each cluster are enumerated as follows: (*i*) 92.86% students of cluster P repeatedly solved 25% of the TP whereas 28.21%

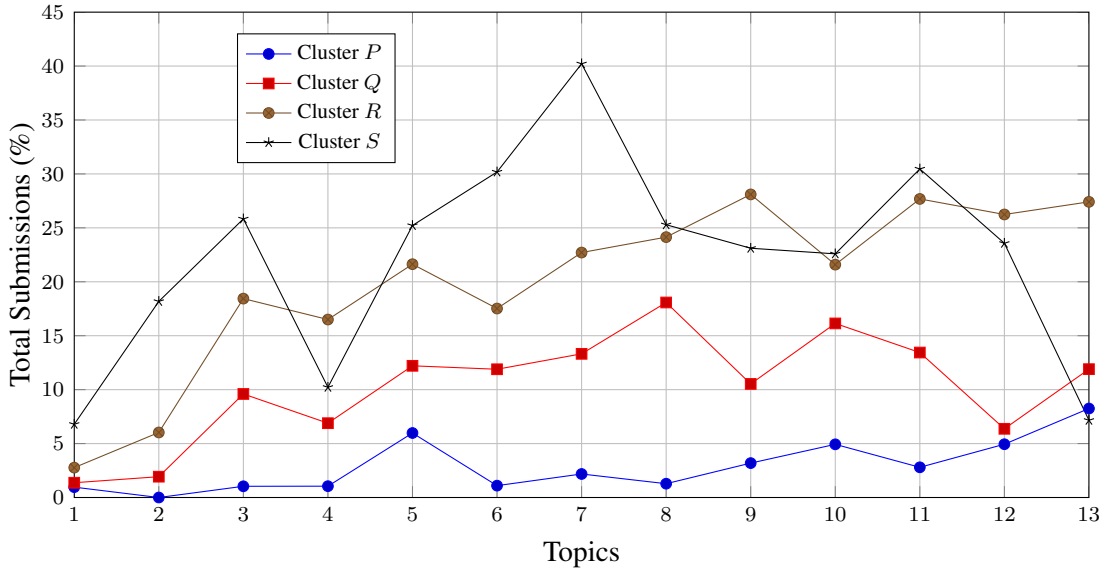


Figure 2.9: Tendency to submit assignments on the last day

Table 2.14: Repetition tendency of accepted problems

| Problems | Number of students (%) | | | |
|-----------|------------------------|-----------|-----------|-----------|
| | Cluster P | Cluster Q | Cluster R | Cluster S |
| 25% of TP | 92.86 | 80.90 | 63.16 | 28.21 |
| 50% of TP | 44.05 | 24.16 | 5.85 | 5.13 |
| 75% of TP | 9.52 | 3.37 | 0 | 0 |

of students of cluster S participated, which is the lowest; (ii) for 50% TP repetition, 44.05% and 24.16% of students from clusters P and Q participated, respectively; and (iii) for 75% TP repetition, 9.52% students participated from cluster P , which is the largest and no students (0%) participated from clusters R and S .

Here, the $CVal$ scores are calculated for each cluster, with average scores of 1, 0.96, 0.96, and 0.44 for clusters P , Q , R , and S , respectively. The students in cluster S received an average $CVal$ score of 0.44, indicating poor coding skills. According to the Definition 2, they likely copied someone else’s code to solve the assignments. In contrast, the students of clusters P , Q , and R obtained higher $CVal$ scores.

2.5.3 Discovering Frequent Data Patterns for Clusters

In this part of the experiment, the FP-growth algorithm is used to discover the frequent data patterns in each cluster. Before the FP-growth algorithm is applied, we prepare clusters’ data in a uniform data format. Therefore, the prominent attributes such as $Prob$, $Accu$, $Verd$, PA , CoT , and PT are selected for ARM. Let $W = \{\{Prob\}, \{Accu\}, \{Verd\}, \{PA\}, \{CoT\},$

$\{PT\}$ be a set of attributes, where $Prob = \{catId \mid catId \in \mathbb{N}, 1 \leq catId \leq 59\}$, $Accu = \{catId \mid catId \in \mathbb{N}, 60 \leq catId \leq 69\}$, $Verd = \{catId \mid catId \in \mathbb{N}, 70 \leq catId \leq 79\}$, $PA = \{catId \mid catId \in \mathbb{N}, 80 \leq catId \leq 89\}$, $CoT = \{catId \mid catId \in \mathbb{N}, 90 \leq catId \leq 99\}$, and $PT = \{catId \mid catId \in \mathbb{N}, 100 \leq catId \leq 109\}$. Thus, the sets $Prob$, $Accu$, $Verd$, PA , CoT , and PT are a subset of W , i.e., $Prob \subseteq W, Accu \subseteq W, Verd \subseteq W, PA \subseteq W, CoT \subseteq W, PT \subseteq W$. The values of the elements in each set have been converted into uniform categorical IDs (catId) according to the dictionary in Table 2.15. After the cluster data is converted into uniform catId, the sample data formats of the tuples are as follows: $W_1 = \{29, 60, 70, 81, 90, 100\}$, $W_2 = \{17, 61, 71, 80, 92, 102\}$, and $W_3 = \{32, 58, 70, 81, 91, 101\}$.

Table 2.15: Dictionary for the set attributes

| catId | Categorization of values | Attributes |
|-------|--|--------------------------------|
| 1-59 | $prob_1, prob_2, prob_3 \dots prob_{57}$ | Problems ($Prob$) |
| 60 | $\geq 75\%$ | Accuracy ($Accu$) |
| 61 | 60%–74% | |
| 62 | 45%–59% | |
| 63 | $< 45\%$ | |
| 70 | Accepted (AC) | Verdicts ($Verd$) |
| 71 | Compile Error (CE) | |
| 72 | Memory Limit Exceeded (MLE) | |
| 73 | Output Limit Exceeded (OLE) | |
| 74 | Presentation Error (PrE) | |
| 75 | Run Time Error (RTE) | |
| 76 | Time Limit Exceeded (TLE) | |
| 77 | Wrong Answer (WA) | |
| 80 | $\geq 80\%$ | Programming Assignment(PA) |
| 81 | 65%–79% | |
| 82 | 45%–64% | |
| 83 | $< 45\%$ | |
| 90 | $\geq 80\%$ | Coding Test (CoT) |
| 91 | 65%–79% | |
| 92 | 45%–64% | |
| 93 | $< 45\%$ | |
| 100 | $\geq 80\%$ | Paper-based Test (PT) |
| 101 | 65%–79% | |
| 102 | 45%–64% | |
| 103 | $< 45\%$ | |

First, the frequency of different attributes in each cluster is calculated, then, the ranking of the attributes based on frequency is also enumerated. Finally, we compute the frequent data patterns for each cluster by varying the minimum support ($minSup$) value. Note that $minSup$ is used to find frequent itemsets from transactions in the database. For better understanding, we assume $minSup = 3$ for a transactional database (TDB). In this case, an item a appears 4 times

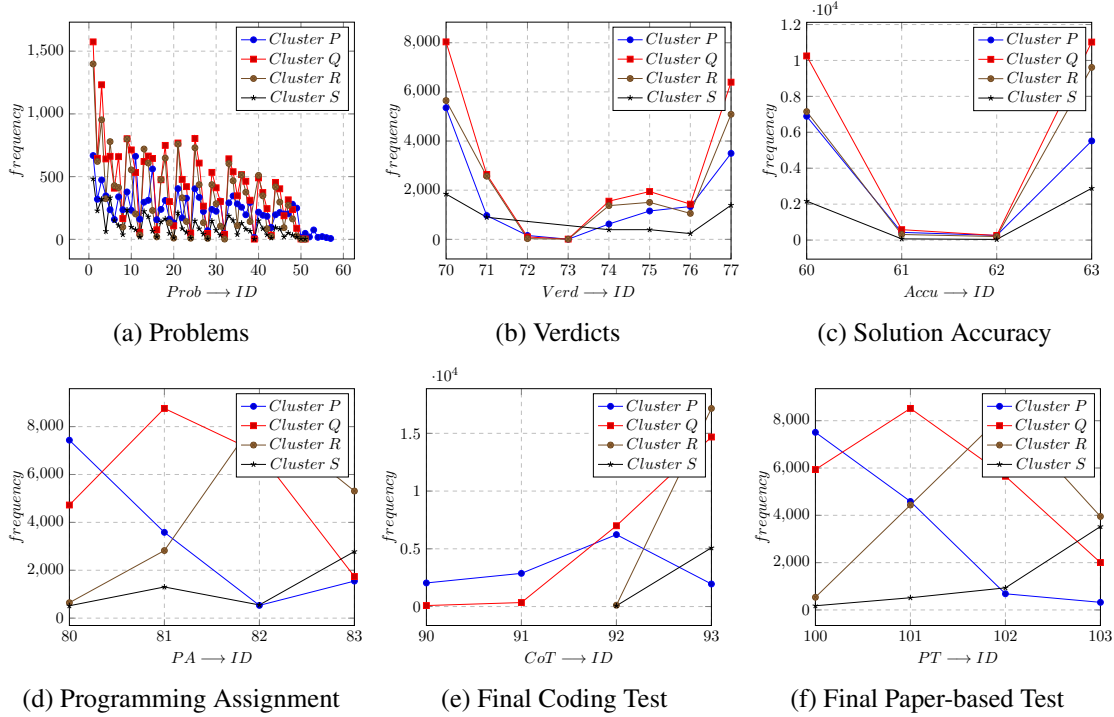


Figure 2.10: Overview of the frequency of attributes in different clusters

out of total 8 transactions in TDB, and since item a satisfies the $minSup$ value ($a \geq minSup$), item a is called a frequent item. In Figure 2.10, we enumerate the $frequency$ of each attribute to investigate interesting patterns of attributes in each cluster. Since the values of the attributes are divided into different groups, and each group is represented by a specific ID . So, in each sub-figure of Figure 2.10, the X -axis represents the ID of the attribute, and the Y -axis represents the $frequency$.

Several observations can be made from this figure. (i) As shown in Figure 2.10a, most students are interested in solving the first 30 problems, but out of this common trend, students in cluster P attempted to solve all problems. In addition, students in clusters Q and R solved more problems than the other clusters. (ii) In Figure 2.10b, students in cluster Q achieved the most AC and WA verdicts among the other clusters based on their submissions. (iii) As shown in Figure 2.10c, students in all clusters maintain the same pattern for $Accu$. Most of the students in cluster Q achieved high $Accu$ than those of clusters P , R , and S , at the same time students in clusters Q and R obtained the most number of low $Accu$. (iv) For the PA scores, as shown in Figure 2.10d, most of the students in cluster P obtained high scores, while most of the students in cluster R obtained low scores. (v) As shown in Figure 2.10e, students in clusters R and S did not achieve high scores (below 65%) in the CoT , instead most of them received low scores. However, more students from cluster P achieved high scores in the CoT than those of cluster

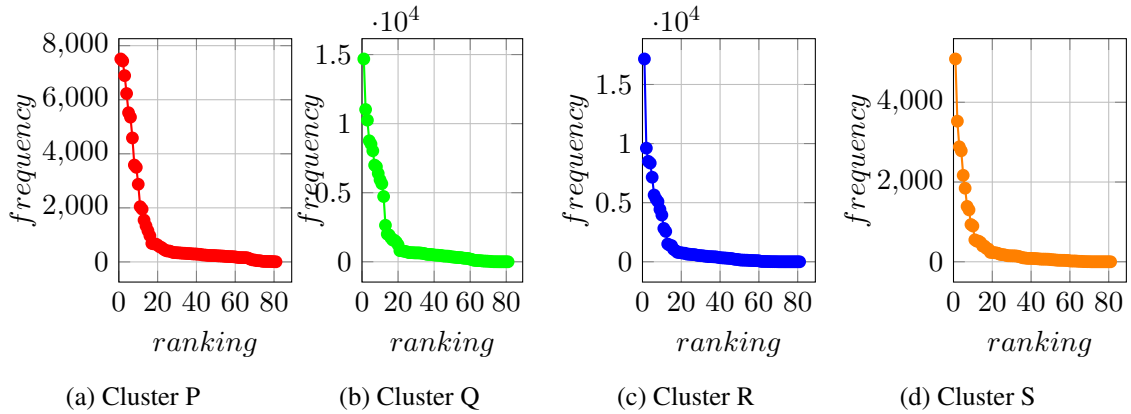


Figure 2.11: Overview of attribute ranking using frequencies for each cluster

Q. (vi) As shown in Figure 2.10f, most students in clusters *P* and *Q* obtained high scores in the *PT*, in contrast, more students from clusters *R* and *S* obtained low *PT* scores.

Attribute ranking was calculated on the basis of the *frequency* values in each cluster. Figure 2.11 shows the frequency distribution of attributes. The *X*-axis shows the ranking of the attributes in order of *frequency* values, and the *Y*-axis shows the *frequency* of the attributes. From this figure, the following observations can be drawn: (i) the distribution of attribute *frequency* is a long tail (or exponential) distribution; (ii) the *frequency* of a small number of attributes (especially the top 20 attributes) is high, and the *frequency* of the remaining attributes is relatively low; (iii) there are similarities in the *frequency*-based ranking patterns of each cluster. The attributes of clusters *Q* and *R* are more frequent (higher *frequency*) than those of the other clusters.

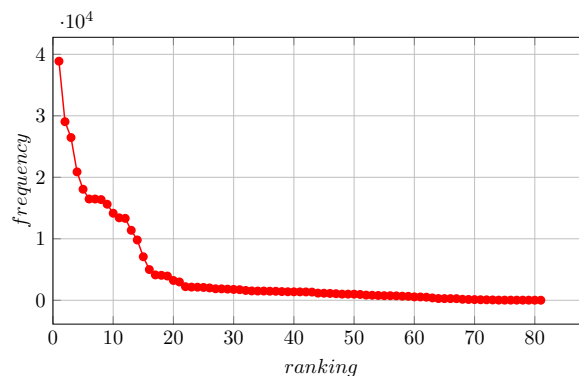


Figure 2.12: Overall ranking-trend of the attributes based on frequencies of clusters

Here, we summarized the frequencies of all clusters and created a ranking curve for the attributes, as shown in Figure 2.12. This ranking curve is similar to the curves for the individual clusters, reflecting the fact that the distribution of attribute frequencies is a long tail, with the first few attributes achieving high frequencies.

Next, we generated frequent patterns from each cluster. Figure 2.13 shows the number of frequent patterns available for each cluster at various minimum support ($minSup$) values. In order to generate patterns for each cluster, we diversified the value of $minSup$ in the range of 500 to 3,500. From this figure, several observations can be made. (i) As the $minSup$ value for each cluster increases, the number of frequent patterns decreases. Because of many patterns could not satisfy the increasing $minSup$ value. (ii) Clusters Q and R have the highest number of patterns generated with any number of $minSup$ values. Cluster S has the lowest number of patterns generated with any number of $minSup$ values.

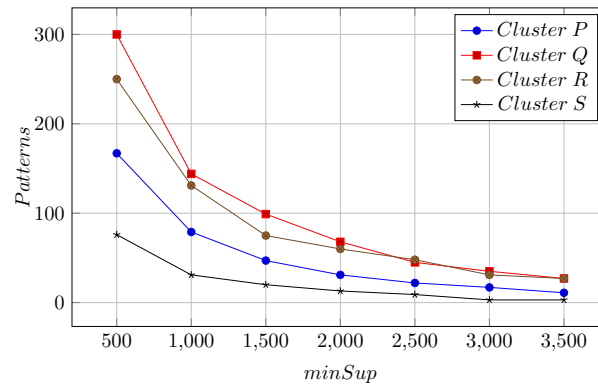


Figure 2.13: A number of generated frequent patterns based on different $minSup$ values

2.5.4 Association Rule Mining

The FP-growth algorithm is applied to the clustered data to find the association rules, which help identify the actual relationship between programming skills and academic performance. In addition, the association rules are used to verify the extracted statistical features of each cluster. Interesting and relevant association rules are obtained from each cluster by setting the optimal minimum support ($minSup$) and confidence ($minConf$) threshold values. For cluster P , we set $minSup = 1500$ and $minConf = 90\%$. Consequently, the frequent rules shown in Table 2.16 are obtained.

Table 2.16: Association rules for the students of cluster P

| Rules |
|---|
| R1: $PT \geq 80\% \ \&\& \ Verd == AC \ \&\& \ PA \geq 80\% \rightarrow Accu(\geq 75\%)$ |
| R2: $PT(65\% - 79\%) \ \&\& \ Verd == AC \rightarrow Accu(\geq 75\%)$ |
| R3: $PT \geq 80\% \ \&\& \ Verd == AC \rightarrow Accu(\geq 75\%)$ |
| R4: $Verd == AC \ \&\& \ PA \geq 80\% \rightarrow Accu(\geq 75\%)$ |
| R5: $Verd == AC \ \&\& \ CoT(45\% - 64\%) \rightarrow Accu(\geq 75\%)$ |

In Table 2.17, the association rules extracted from cluster Q using the values $minSup =$

2000 and $minConf = 90\%$ are listed.

Table 2.17: Association rules for the students of cluster Q

| Rules |
|--|
| R1: $PT(65\% - 79\%) \ \&\& \ Verd == AC \rightarrow Accu(\geq 75\%)$ |
| R2: $PT(45\% - 64\%) \ \&\& \ Verd == AC \rightarrow Accu(\geq 75\%)$ |
| R3: $Verd == AC \ \&\& \ PA(65\% - 79\%) \ \&\& \ CoT < 45\% \rightarrow Accu(\geq 75\%)$ |
| R4: $PT(65\% - 79\%) \ \&\& \ Verd == AC \ \&\& \ CoT < 45\% \rightarrow Accu(\geq 75\%)$ |
| R5: $Verd == AC \ \&\& \ CoT < 45\% \rightarrow Accu(\geq 75\%)$ |
| R6: $Verd == AC \ \&\& \ CoT(45\% - 64\%) \rightarrow Accu(\geq 75\%)$ |
| R7: $Verd == AC \ \&\& \ PA(65\% - 79\%) \rightarrow Accu(\geq 75\%)$ |
| R8: $Verd == AC \ \&\& \ PA(45\% - 64\%) \rightarrow Accu(\geq 75\%)$ |
| R9: $Verd == CE \rightarrow Accu(< 45\%)$ |

Similarly, valuable rules are also extracted from cluster R when $minSup = 3000$ and $minConf = 90\%$. The generated rules are listed in Table 2.18.

Table 2.18: Association rules for the students of cluster R

| Rules |
|---|
| R1: $PT(45\% - 64\%) \ \&\& \ Accu < 45\% \rightarrow CoT < 45\%$ |
| R2: $PT(65\% - 79\%) \rightarrow CoT < 45\%$ |
| R3: $Verd == AC \rightarrow Accu \geq 75\% \ \&\& \ CoT < 45\%$ |
| R4: $Accu < 45\% \rightarrow CoT < 45\%$ |
| R5: $PT(45\% - 64\%) \ \&\& \ Accu \geq 75\% \rightarrow CoT < 45\%$ |
| R6: $Accu \geq 75\% \ \&\& \ Verd == AC \rightarrow CoT < 45\%$ |
| R7: $Accu < 45\% \ \&\& \ PA(45\% - 64\%) \rightarrow CoT < 45\%$ |
| R8: $Accu < 45\% \ \&\& \ Verd == WA \rightarrow CoT < 45\%$ |
| R9: $Accu < 45\% \ \&\& \ PA < 45\% \rightarrow CoT < 45\%$ |
| R10: $PA < 45\% \rightarrow CoT < 45\%$ |
| R11: $Verd == AC \ \&\& \ CoT < 45\% \rightarrow Accu(\geq 75\%)$ |
| R12: $Accu \geq 75\% \ \&\& \ PA(45\% - 64\%) \rightarrow CoT < 45\%$ |
| R13: $PT(45\% - 64\%) \ \&\& \ PA(45\% - 64\%) \rightarrow CoT < 45\%$ |
| R14: $PT < 45\% \rightarrow CoT < 45\%$ |

Finally, rules are generated for cluster S when we set $minSup = 1500$ and $minConf = 90\%$, as shown in Table 2.19.

As shown in Figure 2.14, we varied the values of $minConf$ and $minSup$ and showed the number of relevant rules in each cluster. First, we set several values of $minSup$, such as 500, 1000, 1500, 2000, 2500, and 3000. For each value of $minSup$, we varied the value of $minConf$ to 50%, 60%, 70%, 80%, and 90%. As a result, clusters Q and R generated the most association rules based $minSup$ and $minConf$ values. On the other hand, relatively few rules are generated on the basis of different $minSup$ and $minConf$ values in clusters P and S .

The following observations can be obtained based on the association rules from different

Table 2.19: Association rules for the students of cluster *S*

| Rules |
|--|
| R1: $Accu \geq 75\% \rightarrow CoT < 45\%$ |
| R2: $Verd == AC \rightarrow CoT < 45\%$ |
| R3: $PT < 45\% \ \&\& \ Accu < 45\% \ \&\& \ PA < 45\% \rightarrow CoT < 45\%$ |
| R4: $PT < 45\% \ \&\& \ Accu < 45\% \rightarrow CoT < 45\%$ |
| R5: $PT < 45\% \ \&\& \ PA < 45\% \rightarrow CoT < 45\%$ |
| R6: $Accu \geq 75\% \ \&\& \ Verd == AC \rightarrow CoT < 45\%$ |
| R7: $PA < 45\% \ \&\& \ CoT < 45\% \rightarrow PT < 45\%$ |
| R8: $Accu < 45\% \ \&\& \ PA < 45\% \rightarrow CoT < 45\%$ |
| R9: $Verd == AC \ \&\& \ CoT < 45\% \rightarrow Accu(\geq 75\%)$ |
| R10: $Accu < 45\% \rightarrow CoT < 45\%$ |
| R11: $Accu < 45\% \ \&\& \ PA < 45\% \ \&\& \ CoT < 45\% \rightarrow PT < 45\%$ |

clusters: (i) in cluster *P*, association rules are involved with higher *accuracy*, *PA*, *PT*, and *CoT*, as well as the most frequent accepted (*AC*) verdicts; (ii) students of cluster *Q* showed with higher *accuracy*, *PA*, and *PT* but lower scores in *CoT*; (iii) students of cluster *R* tended to have lower scores in *CoT*, *PT*, and *PA*, as well as infrequent *AC* verdicts; (iv) cluster *S* students showed lower scores in *CoT*, *PT*, and *PA*, as well as lower *accuracy*; and (v) clusters *Q* and *R* yielded the most association rules for any *minSup* and *minConf* values.

2.5.5 Accumulation of Correlated Features

Many significant features are generated from each cluster by employing the proposed framework. These features are deeply correlated to each other and meaningful. These correlated features and rules are accumulated for each cluster.

Students of the cluster *P* (i) took an average problem-solving *T&E* of 12.40 (Table 2.9), (ii) solved an average of 56.14 extra problems beyond their academic assignments (Table 2.11), (iii) repeated more accepted (*AC*) solutions for optimization than those of other clusters (Table 2.14), (iv) submitted their assignments on the very first day more than those in other clusters (Figure 2.7), and (v) had the lowest average last-day submission rate of approximately 2.90% than clusters *Q*, *R*, and *S* (Table 2.13). These features are interdependent and deeply correlated to each other. The features mentioned above have interesting meanings; overall, these features indicate that students in this cluster are committed to programming, which has a positive impact on their programming and academic performance.

Cluster *P* also had an overall *AC* rate (considering problems *A*, *B*, *C*, and *D*) of 40.88% (Table 2.7), topic-wise *AC* rate (considering problems *A* and *B*) of 46.24% (Table 2.12), aver-

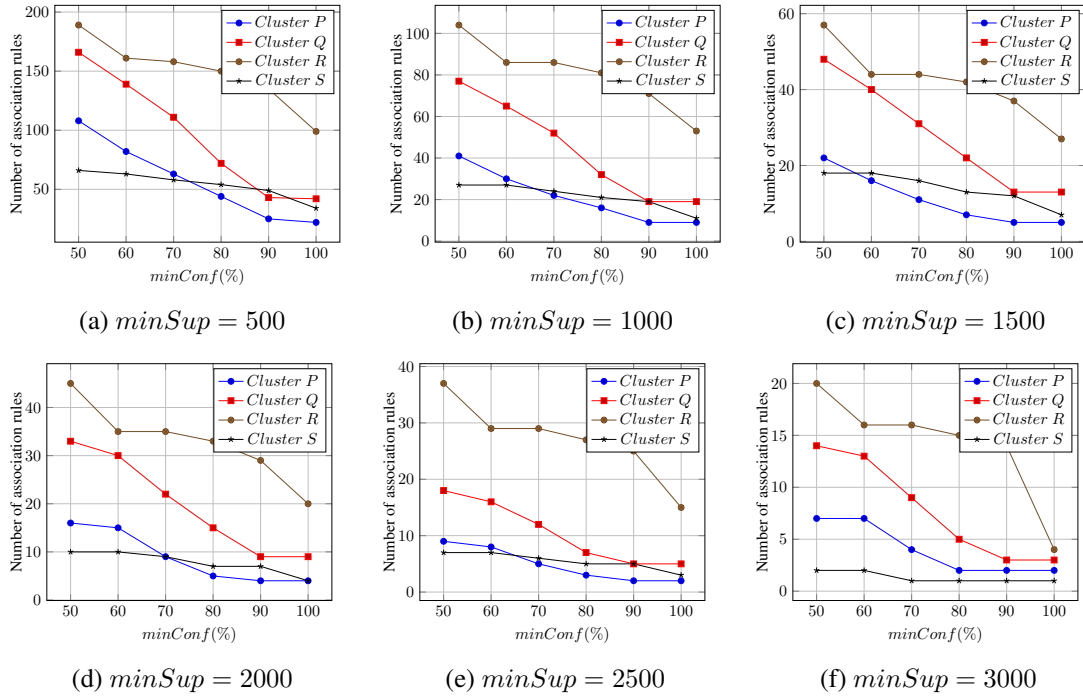


Figure 2.14: Overview of association rules generated using different $minSup$ (500, 1000, 1500, 2000, 2500, 3000) and $minConf$ (50%, 60%, 70%, 80%, 90%, 100%) values

age solution *accuracy* of 55.71% (Table 2.9), and higher *CV al* score of 1. These higher success rates in programming enabled higher scores in *PA*, *CoT*, and *PT* that are also validated by the association rules (Table 2.16).

In cluster *Q* the overall *AC* rate considering all problems is 36.34% (Table 2.7) and the topic-wise average *AC* rate (considering problems *A* and *B*) is 40.06% (Table 2.12), which are lower than those of clusters *P* and *S*. The students of cluster *Q* consistently maintained a high *AC* rate throughout the thirteen topics (Figure 2.8) but solved minimal extra problems beyond their academic assignments (Table 2.11). They had a lower tendency to submit solutions on the last day than students of clusters *R* and *S* instead submitted their assignments early (Table 2.13). The students of cluster *Q* obtained higher scores in *PA* and *PT* than in *CoT* (Table 2.10), as shown by the association rules (Table 2.17). Most of the features involved high values, indicating that they put a great effort into programming. However, the lower attempt to solve additional problems likely affected the *CoT* scores in this cluster.

Students of cluster *R* (*i*) took an average of 9.52 attempts/trials to solve problems, (*ii*) did not solve many additional problems outside of regular academic assignments (Table 2.11), (*iii*) submitted their assignments on the deadline or the day before (Figure 2.7), and (*iv*) rarely repeated the *AC* problems more than once (Table 2.14). These features are related to their

various programming activities and indicate that they did not put much effort into programming. These students also received the highest error (WA , CE , RTE , TLE , etc.) verdicts of 67.30% and the lowest AC rate of 32.70% compared to clusters P , Q , and S . Their topic-wise AC rate is not coherent across all thirteen topics. Students in cluster R obtained good scores in PA (60.92) and PT (68.26), but lower scores in CoT (28.46) (Table 2.10). The association rules showed that students were involved with lower scores and infrequent AC verdicts. Note that the coding test (CoT) is used to verify the students' core programming skills. Thus, less effort in programming negatively affects this CoT score.

Students of cluster S (i) undertook an average of 9.86 attempts/trials to solve problems (Table 2.9) (ii) solved no additional problems (Table 2.11), (iii) had the highest rate of last-day submission for each assignment with an average of 22.22% solutions submitted on the last day (Figure 2.7 and Table 2.13) compared to clusters P , Q , and R , and (iv) obtained very low CV_{al} score of 0.44. Furthermore, an insignificant number of students attempted to repeat the AC problems more than once (Table 2.14). Collectively, these features indicate that students in cluster S did not perform well in programming. Most features are negatively prioritized. Consequently, students in this cluster obtained the lowest scores in CoT (16.55), which is alarming for actual coding performance. Most of the association rules are connected with lower CoT scores (Table 2.19). In addition, we found an interesting correlation: most of the students submitted their solutions on the last day, but achieved higher AC rates and *accuracy*. This trend differs from that of clusters P and Q .

2.6 Discussion

In this Chapter, many hidden features are obtained by employing the proposed framework, where MK-means is applied for data clustering and then FP-growth is applied to the clustered data to discover the association rules. Interesting features and behaviors are observed that are not readily apparent in the base dataset. After applying the elbow and MK-means algorithms to the dataset, four (04) clusters are found. Different features and rules are extracted from each cluster considering the different conditions presented in the experimental results section (2.5). Next, we discuss the features and the resulting explanations, recommendations, assessments, practical applications, and limitations.

For a better understanding, ten main features are listed in Table 2.20 with three indicator

Table 2.20: List of the main features

| No. | Features | Values |
|-----|---|--------------|
| a | Trial and error (<i>T&E</i>) | <i>H/M/L</i> |
| b | Extra problem solutions beyond academic assignments | <i>H/M/L</i> |
| c | Tendency to submit programming assignments | <i>E/M/D</i> |
| d | Topic-wise average last day submission | <i>H/M/L</i> |
| e | Number of students who repeatedly solved accepted (<i>AC</i>) problems for optimization | <i>H/M/L</i> |
| f | Accepted (<i>AC</i>) rate (both topic-wise and problems) | <i>H/M/L</i> |
| g | Overall accuracy (<i>Accu</i>) | <i>H/M/L</i> |
| h | Scores in programming assignment (<i>PA</i>) | <i>H/M/L</i> |
| i | Scores in coding test (<i>CoT</i>) | <i>H/M/L</i> |
| j | Scores in paper-based test (<i>PT</i>) | <i>H/M/L</i> |

values: higher (*H*), medium (*M*), and lower (*L*). Feature *c*, which indicates when the assignments are submitted within the allotted time, uses indicator values of early (*E*), mid-time (*M*), and delay (*D*).

2.6.1 Analysis and Recommendations

In the summary graph of the main features shown in Figure 2.15, we observe that the students of cluster *P* performed extraordinarily well in different programming activities and academic tests. Importantly, most students in this cluster are highly enthusiastic about programming, with more than 62% of total solutions (Figure 2.7) submitted on the very first day of all assignments. They also solved an average of 56.14 extra problems in addition to their academic assignments. The tendency to submit solutions on the last day is approximately 2.90% which is the lowest compared to clusters *Q*, *R*, and *S* (Table 2.13). For solution optimization, a large number of students repeated their *AC* solutions (Table 2.14). In addition, these students achieved higher *AC* rates of 46.24% for problems *A* and *B* (Table 2.12), *accuracy* of 55.71% (Table 2.9), and scores on various tests of 81.22%, 65.46%, and 82.33% for *PA*, *CoT*, and *PT*, respectively than those of clusters *Q*, *R*, and *S* (Table 2.10), as reflected by the association rules (Table 2.16). In contrast, the total error verdict is analyzed from this cluster, with approximately 45% error due to *WA*, 19% due to resource limitations (*TLE*, *MLE*, *OLE*), and 15% due to *RTE* (Figure 2.5).

Note that, to develop students' programming skills and ensure the efficiency of the solution codes, several constraints are set for problems such as input and output limits/numbers, space and time complexity. In this case, a solution code must satisfy the set of constraints to be

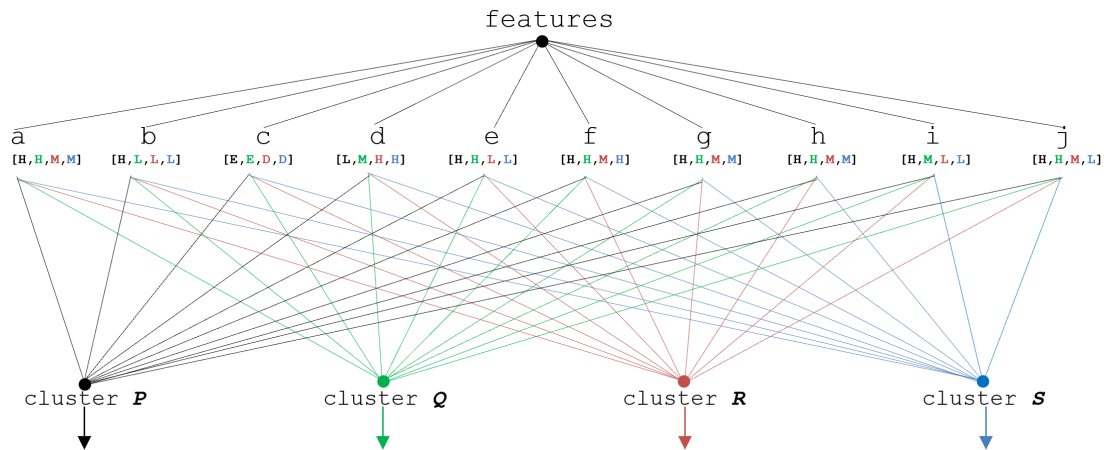


Figure 2.15: A summary graph of the main features

accepted, otherwise it receives error verdicts such as *TLE* and *MLE*. Figure 2.5 shows that students in clusters *Q*, *R*, and *S* received 10.87%, 9.32%, and 6.25% errors due to *TLE* and *MLE*, respectively. In contrast, the students of cluster *P* received about 19.28% errors due to *TLE* and *MLE*, which is the highest compared to clusters *Q*, *R*, and *S*. However, students in cluster *P* took about 12.40 attempts (*T&E*) to solve a problem, which is higher than students in clusters *Q*, *R*, and *S* (Table 2.9). In general, problems *C* and *D* are comparatively more difficult and contain tough constraints than problems *A* and *B*. Students in cluster *P* submitted the highest percentage of solutions, about 43.36%, for problems *C* and *D* compared to clusters *Q*, *R*, and *S* (Table 2.8). Moreover, each student in cluster *P* solved an average of 56.14 additional problems, which is significantly higher than students in clusters *Q*, *R*, and *S* (Table 2.11).

Students in cluster *P* attempted many additional and challenging problems, resulting in a high percentage of errors in *TLE* and *MLE*. Usually, complex algorithm-based problems contain various tough constraints, and sometimes it is very difficult to deal with these kinds of constraints alone without prerequisite knowledge. Our analysis shows that students in cluster *P* have a high tendency to take on difficult problems independently (Table 2.8), and have achieved significant success in solving problems with tough constraints (Table 2.7). Besides, students in this cluster still have the opportunity to further improve their programming skills in dealing tough constraint-based problems. Based on the overall empirical and analytical results, we can summarize that the students of cluster *P* are highly skilled and enthusiastic about programming and perform well on academic tests.

Similarly, students in cluster *Q* achieved higher values in most features, as shown in Figure

2.15. More than 40% of their assignments were submitted on the very first day (Figure 2.7), with higher *accuracy*, *AC* rate, repetition tendency, and scores in *PA* and *PT*. The error verdicts of this cluster have been analyzed approximately 45% of the errors occur due to *WA*, 19% due to *CE*, 11% due to *PrE*, 11% due to resource limitation (*TLE*, *MLE*, *OLE*), and 14% due to *RTE* (Figure 2.5). These students can be understood by analyzing the reasons for each type of error. In addition to these positive features, we found some flaws. The students in cluster *Q* achieved medium (*M*) scores in *CoT* and did not solve a significant number of problems outside of their regular assignments. Usually, *CoT* is used to verify actual programming ability; a medium score in *CoT* means students need to pay more attention in programming. Accordingly, to improve programming skills, students can practice more outside of their academic workload.

Considering all the results and analysis, we determined the following recommendations for clusters *P* and *Q*: (i) special attention to these students can further improve their skills and knowledge; (ii) more difficult problems can be assigned to these students because they find general assignments are very easy; and (iii) they can be involved in real-world problem-solving tasks.

For the students of cluster *R*, the rate of last-day submission was approximately 20.06% (Figure 2.7) which indicates a tendency to delay submission, and they show inconsistent acceptance (*AC*) for all topics (Figure 2.8). Moreover, this cluster had the lowest acceptance (*AC*) and *accuracy* rates among all clusters; very few students repeated their accepted solutions for optimization and solved extra problems. Students of this cluster obtained the highest error rate (67.30%) among all the clusters. The extracted association rules show that most of these students achieve lower *CoT* scores, *accuracy*, *AC* rate, and *WA* verdicts. The students in this cluster scored much higher in *PA* (60.92) than *CoT* (28.46). During *PA*, students can consult with others to solve problems. This may allow some students to solve problems with the help of other students without understanding the problems properly. In contrast, students are not allowed to consult/talk with others during *CoT*, in which students of this cluster rarely obtain good scores. The average *CoT* score is 28.46 out of 120. Considering all the features, it is concluded that (i) students may solve assignments with the help of others without understanding the problems and this cluster (ii) lacks actual programming skills, and (iii) has less effort in programming.

Figure 2.15 shows that students of cluster *S* achieved lower values in most features. Their last-day submission rate is 22.22%, which is the highest among all clusters. They achieved

lower scores in coding and paper-based examinations (CoT and PT), but obtained relatively high scores in PA . Similarly, they had fewer $T\&E$ attempts but achieved higher *accuracy* and AC rate. Most of the association rules are involved with lower scores (CoT and PT) and *accuracy*, as well. The following observations can be obtained from the extracted features: (i) while a large number of solutions were submitted on the last day, there may have been some students who waited for other solutions to become available; this is justified by the $CVal$ score. (ii) There is an unusual trend where students obtained lower scores in CoT while achieving higher AC rate, *accuracy*, and PA scores; this suggests (iii) a lack of actual programming skills and (iv) less effort in programming, and that (v) the students may solve their assignments through collaboration with others.

After analyzing the features and association rules from different perspectives, some deficiencies have been identified in the programming and academic fields for the students of clusters R and S . Accordingly, we provide some recommendations that may help improve students' programming skills and academic performance: (i) special assistance can be provided in the development of algorithms and mathematical logic; (ii) encourage students to solve problems with self-knowledge and understanding; (iii) students can participate in different programming activities, such as competitions, programming lectures, and workshops; and (iv) teachers should give these students additional attention and support in theory and exercise classes and observe their responses.

2.6.2 Pattern and Association Rule Mining

Data patterns and associated rules are mined from the programming/problem-solving data of each cluster using the FP-growth algorithm. Based on the data segmentation (Table 2.15) and mathematical models. This process is useful for visualizing data patterns in detail. Figure 2.10 shows the frequencies of the data, where the frequencies of the individual segmented attributes are also calculated. Figure 2.10 provides some important insight, such as Figure 2.10a showing that students in cluster P attempted and solved all problems, and Figure 2.10e showing that students in clusters R and S did not obtain a high score (below 65%) on the final coding test. Thus, the segmentation of the scores provides a deeper insight into the patterns of the data that would not be visible without the segmentation. The ranking of attributes based on frequencies is shown in Figure 2.11. A long-tail (or exponential) data distribution is observed, with the first 20 attributes having higher frequencies. Figure 2.13 shows the number of patterns generated based

on the *minSup* values. Cluster *Q* and cluster *R* generated the highest number of patterns for any *minSup* value. Furthermore, Figure 2.14 shows the number of association rules for different values of *minSup* and *minConf*. It can be seen that cluster *R* has the highest number of association rules, no matter how many values of *minSup* and *minConf* are used. In addition, some extracted association rules are listed in Tables 2.16, 2.17, 2.18, 2.19 for clusters. The whole process of pattern and rule mining exposed the hidden information from the problem-solving data and can be useful for other real-world educational applications.

2.6.3 Learning and Teaching Strategies for Programming

One of the main objectives of this research is to understand what difficulties students have in solving programming problems, identify the main influencing factors in their programming learning process, and determine what strategies, methods, or technologies can be used in teaching and learning to improve students' programming skills. In Table 2.6, it can be seen that although the number of students in cluster *S* is higher (18.62%) than in cluster *P*, the students of cluster *S* submitted the lowest number of solution codes (7.36%) for evaluation compared to the other clusters (*P*, *Q*, and *R*). These statistics also indicate that students in cluster *S* put less effort in solving the problems. Another important statistic we found in Table 2.7 is that students in clusters *P*, *Q*, *R*, and *S* received about 51.75%, 51.71%, 52.42%, and 42.22% error verdicts (*WA*, *RE*, *PE*, *TLE*, *MLE*, and *OLE*) respectively in the solution codes that cannot be identified by the compilers or correctly recognized by the students. Understanding and reducing these errors (*WA*, *RE*, *PE*, *TLE*, *MLE*, and *OLE*) in the solution code is also a challenging task. Since students in all clusters rated about 50% of the submitted solutions as incorrect excluding compile error (*CE*), and these errors involved semantic, mathematical, and logical errors. To improve students' mathematical and logical skills, they need different types of problems and a suitable practice environment. In addition, some other strategies may be useful, such as rapid response and continuous monitoring of students' programming activities, programming workshops to address students' weaknesses.

2.6.4 Overall Assessments and Practical Applications

Considering all the empirical results and analysis, we see that the students of cluster *P* obtained the highest acceptance rate of 40.88% for all problems (*A*, *B*, *C*, and *D*) (Table 2.7), average solution accuracy of 55.71% (Table 2.9), solved the average additional problem of 56.14

(Table 2.11), a faster propensity to submit assignments early (Figure 2.7), topic-wise accepted solution rate of 46.24 for problems (*A* and *B*) (Table 2.12), lowest submission rate on the last day of 2.90% (Table 2.13), highest number of repetitions (Table 2.14), highest *PA*, *CoT*, and *PT* scores of 81.22%, 65.46%, and 82.33%, respectively (Table 2.10). All the features indicate that the students of cluster *P* invested great efforts in programming-related tasks. In addition, the summary graph (Figure 2.15) of the features shows that the students of cluster *P* are involved in better indicators in all the features. Similarly, students in cluster *Q* received an acceptance rate of 36.34% for all problems (*A*, *B*, *C*, and *D*) (Table 2.7), solution accuracy of 48.45% (Table 2.9), high tendency to submit assignments early (Figure 2.7), the topic-wise acceptance rate of 40.06% for problems (*A* and *B*) (Table 2.12), last-day submission rate of 10.28% (Table 2.13), and *PA*, *CoT*, and *PA* scores of 67.86%, 40.18%, and 70.43% respectively (Table 2.10). As shown in Figure 2.15, most of the features are associated with good indicators. It can be seen that the students of cluster *Q* also performed well in programming.

On the other hand, students in cluster *S* did not solve any additional problems (Table 2.11), had a less repetition tendency (Table 2.14), a higher last day submission rate of 22.22% compared to clusters *P*, *Q*, and *R* (Table 2.13 and Figure 2.7), and received the lowest *CVal* score of 0.44 compared to clusters *P*, *Q*, and *R*. Besides, students scored 54.33%, 13.79%, and 44.83% on *PA*, *CoT*, and *PT*, respectively, which is very poor compared to clusters *P*, *Q*, and *R* (Table 2.10). The overall results show that the students in this cluster did not perform well in programming. In addition, the summary graph (Figure 2.15) of the features and association rules (Table 2.19) show that they were involved with lower indicators in most features.

From the above results, it can be seen that the students of clusters *P* and *Q* made a good effort in programming and obtained good results in various tests, while the students of cluster *S* made less effort and therefore achieved poor results in various tests. So, we can conclude that if students (especially in ICT-related disciplines) perform well in practical applications (e.g., programming, logical implementation) then they are also likely to perform well in different academic activities, including tests. In addition, the current research provided some recommendations for students based on the identified features and flaws. Teachers, instructors, and faculty advisors can use these analytical results and recommendations to improve students' programming and academic performance levels. Furthermore, the proposed framework, experiments, and overall analytical results can be applied to other related courses/disciplines.

However, the ultimate goal of this Chapter is to support and improve student learning by

identifying their weaknesses and strengths. For this purpose, a real-world dataset from a programming course was used. The proposed framework included EDM techniques and LA to find invisible knowledge from the e-learning data. The knowledge was then analyzed and visualized from various perspectives. The results of these analyses highlight the weaknesses and strengths of the students and improve their learning. The proposed research can be suitable for practical applications for the following reasons: (i) the proposed research can provide a useful direction, that is, how to deal with e-learning data, (ii) e-learning data processing has always been a challenging task, in this regard, the proposed research shows the way of handling real-world e-learning data. As the proposed research has already processed OJ (e-learning) data for EDM and LA, (iii) the process of data analysis and its results can be helpful for other related courses to improve students' learning, and (iv) the proposed framework can be integrated with existing e-learning platforms for EDM and LA purposes.

2.6.5 Limitations

The proposed framework is leveraged for data clustering, and then the hidden features and association rules are extracted from each cluster. The results are generated based on a dataset comprising submission logs and scores collected from the AOJ system; they may vary for other datasets due to noise or irrelevant data. The number of association rules may vary depending on the threshold values of *minSup* and *minConf*. The value of *k* for the MK-means clustering algorithm may differ based on the dataset. Therefore, the proposed framework can produce better or worse results for other datasets.

2.7 Summary

In this Chapter, a novel framework is proposed for exploring the effects of programming education. Subsequently, a programming course was selected as a sample course for experiments and analyses. By employing the framework, many meaningful and significant features were extracted from the dataset. The extracted features are deeply correlated to the students' behavior. The analytical results showed that better practical (e.g., programming) skills have a positive effect on academic performance. Moreover, the interaction and interdependence between practical skills and academic performance are presented based on the experimental results. Thus, we have concluded that if a student of an ICT or engineering discipline performs well in prac-

tical assignments (e.g., programming, logical implementation, PL/SQL), then they are likely to perform well in other academic activities. The overall approach of this Chapter is applicable to other fields such as education, EDM, LA, data analytics, and behavior analysis.

Chapter 3

Code Assessment and Classification Using Attention-Based LSTM Neural Network

3.1 Introduction

Programming is one of mankind's most creative and effective endeavors, and vast numbers of studies have been dedicated to improving the modeling and understanding of source code [105]. The outcomes of many such studies are now supporting a wide variety of core source code assessment purposes, such as error detection, error prediction, error location identification, snippet suggestions, code patch generation, developer modeling, and source code classification [105]. Since learners and professionals around the world are constantly creating large numbers of new programs to improve our lives, it is a general truism that no program is ever released without undergoing a comprehensive post-development debugging process. Almost every software product/solution code goes through different testing phases in the software engineering (SE) cycle. In fact, once errors are detected in the solution code at the production or testing phases, the debugging process begins immediately to find and fix the errors. This means that learners and professionals are spending vast amounts of time attempting to find errors in solution codes.

To find a single error, it is often necessary to verify an entire program code, which is a very lengthy process and time-consuming. This adverse situation has resulted in the emergence of a new SE research window [106]. There are significant numbers of errors that are commonly

made by students, novice and professional programmers. These include missing semicolons, delimiters, irrelevant symbols, variables, missing braces, incomplete parentheses, operators, missing methods, classes, inappropriate classes, inappropriate methods, irrelevant parameters, and logic errors (*TLE*, *MLE*, *OLE*, *PrE*). Although such errors often indicate inexperience, insufficient concentration to detail, and other unsuitable behaviors. A Google study on programming showed that such errors can creep into the works of even the most experienced programmers [107].

Usually, programming is a very sensitive and error-prone task and a single mistake can eventually be harmful to software end-users. Furthermore, the source code is highly error-prone during development, so the intelligent support model for code assessment has become interesting research area. Among the solutions now being explored, the use of AI offers fascinating potential for solving source code related complications. In the past few years, natural language processing (NLP) developers have produced some extraordinary outcomes in different domains such as language processing, machine translation, and speech recognition. The reasons for the wide-ranging success of NLP is founded on its corpus-based methods, statistical applications, messenger suggests, handwriting recognition, and increasing large corpora of text. For example, n -gram models are among the stochastic language model forms that can be used for predicting the next item based on corpus. Different n -gram models such as bi-gram, tri-gram, skip gram [36], and GloVe [108] are all statistical language models that are very useful in language modeling applications. This burgeoning usage has stimulated the availability of a large text corpus and is helping NLP techniques to become more effective on a day-by-day basis. However, the NLP language model is not particularly effective when used in complex source code assessment endeavors but still useful for the intuitive language model. As a result, numerous researchers have focused their efforts on source code assessment tasks using neural network-based language models.

Z. Tu et al. [109] proposed a local cache model that dealt with localness of source code, but still encountered problems with small-context source code using an n -gram model. Their study determined that neural network-based language models could provide robust substitutes for source code assessments. Additionally, another study [37] showed that the RNN model, which is capable of retaining longer source code context than traditional n -gram and other language models, has achieved mentionable success in language modeling. However, the RNN model faces limitations when it comes to representing the context of longer source codes due

to gradient vanishing or exploding [110], which makes it is hard for it to be trained using long dependent source code sequences. As a result, it is only effective for a small sequence of source codes. To minimize gradient vanishing or exploding problems, the RNN model has been extended to LSTM neural networks [110]. An LSTM network is a special kind of RNN that can remember longer source code sequences due to its extraordinary internal gate structure.

In this Chapter, we are presenting an intelligent support model for source code/solution code assessment that was designed using an LSTM in combination with an attention mechanism (then known as LSTM-AM) which increases the performances than a standard LSTM model. The attention mechanism is a useful technique that takes into account the results of all past hidden states for prediction. The attention mechanism can improve the accuracy of neural network-based intelligent models. We trained LSTM, RNN, and LSTM-AM networks with different hidden units (neurons) such as 50, 100, 200, 300 using a bunch of solution codes taken from an OJ system. Erroneous solution codes were then input into all models to determine their relative capabilities in regards to predicting and detecting code errors. The obtained results show that the proposed LSTM-AM network extends the capabilities of the standard LSTM network in terms of detecting and predicting such errors correctly. Even some solution codes contain logical errors (i.e., *TLE*, *OLE*, and *MLE*) and other errors (i.e., *PrE* and *WA*) that cannot be detected by the usual compilers whereas the proposed intelligent support model can detect these errors.

Additionally, the LSTM-AM network can retain a longer sequence of solution code inputs and thus generate more accurate output, than the standard LSTM and other state-of-the-art networks. Furthermore, we diversified with different settings and hidden units to create the most suitable model for our research in terms of cross-entropy, training time, accuracy and other performance measurements. Also, proposed model can classify the solution codes based on the defects in codes. We expect that the proposed model can be useful for students, novice programmers, and professionals as well as overall programming education and other aspects of SE. The main contributions of this Chapter are listed below:

- The proposed intelligent support model can help students, novice and professional programmers for the source code completion.
- The intelligent support model detects such errors (logical) that cannot be identified by the conventional compiler.
- The proposed intelligent support model accuracy is approximately 62% that outperformed

other benchmark models.

- The proposed model can classify the solution codes based on the detected errors. The classification accuracy is 96% that is much better than other models.
- The proposed model highlights defective spots with location/line number in solution codes.
- The proposed model improves the ability of learners to fix errors in source code easily by using the location/line numbers.

The remainder of the Chapter 3 is arranged as follows. In Section 3.2, we present the background and related works. Section 3.3 describes the overview of natural language processing and artificial neural networks. In section 3.4, we present the proposed approach. Data collection and problem description issues are presented in Section 3.5. Evaluation metrics are defined in Section 3.6. The experimental results are presented in Section 3.7. In Section 3.8, we discuss the results. To that end, Section 3.9 concludes the Chapter 3 with some future work proposals.

3.2 Background and Related Works

Modern society is flourishing due to advancements in the wide-ranging fields of ICT, where programming is a crucial aspect of many developments. Millions of source codes are being created every day, most of which are tested through manual compiling processes. As a result, an important research field that has recently emerged involves the use of AI systems for source code completion during development rather than manual compiling processes. More specifically, neural network-based models are using for source code assessments in order to achieve more human-like results. Numerous studies have been completed and a wide variety of different methods proposed regarding the use of neural networks in programming-related fields, some of which will be reviewed below.

In [105], the authors present a language model for source code testing that uses a neural network instead of an existing language (i.e. n -gram) model. In most cases, n -gram language models cannot handle long source code sequences effectively, so neural network-based language models were developed to improve source code analyses. In the cited study, in which RNN and LSTM language models were trained and the obtained results showed that LSTM model performed better than the RNN model. That study used a Java project corpus to evaluate

the performance of the language models. In [36], the authors proposed a novel LSTM-based source code correction method that used segment similarities. More specifically, the study utilized the seq2seq model for source code correction process. The seq2seq model is a machine learning approach which is very effective for language modeling such as machine translator, conversational model, text translator, and image captioning.

White et al. [37] proposed a deep software language model using RNN. The obtained experimental results using a Java corpus showed that the proposed software language model outperformed conventional models like cache-based n -gram and standard n -gram. That software language model showed significant promise for use in SE fields. In [41], authors presented a novel Tree-LSTM based model where each LSTM unit used as a tree. That model assesses semantic relatedness prediction tasks based on sentence pairs and sentiment classification. Meanwhile, in [42], the authors proposed a method that classified archived source codes by language type using an LSTM. Their experimental results demonstrated that the proposed LSTM surpassed the linguist classifier, Naive Bayes (NB) classifier and other similar networks.

In [111], the authors proposed a technique that automatically identifies and corrects source code syntax errors using an RNN. Their proposed SYNFix algorithm finds the error location of the next predicted token sequence, after which the identified error is solved by either replacing or inserting a proper word. A significant limitation of this technique is that it cannot recover or handle multiple syntax errors in a source code sequence. Pedroni et al. [112] is studied to find the appropriate type of compiler messages that can assist novice programmers in order to identify source code errors and what actions are needed for the error messages? In that study, the authors experimentally showed that certain message types help novice programmers.

In [113], the authors presented a model for source code syntax error correction that was written in the C programming language. That model, called DeepFix, uses a multi-layered seq2seq neural network combined with an attention mechanism. The authors also proposed a trained RNN that can predict an error with its location number, as well as fix the error with a proper token. The experimental results obtained showed that, out of a total of 6971 source code errors, this approach completely fixed about 27% and partially fixed about 19%. Rahman et al. [63] proposed a language model using LSTM for fixing source code errors. That proposed model was combined attention mechanism with LSTM which increases the effectiveness of standard LSTM. Experimental results showed that the model significantly corrected errors in the solution codes. In [114], authors proposed a source code bug detection technique that works

by varying the hyperparameters of an LSTM in order to investigate perplexity issues and training time. The results obtained show that LSTM produces significant results for source code error detection.

Bahdanau et al. [115] proposed a language translation model that uses RNN. More specifically, the encoder-decoder technique is used as a translator when it is necessary to encode a source text into a fixed-length vector. By utilizing the vector length, decoder can translate the sentences. The paper extends fixed-length limitations by allowing (soft) search from the source sentence to predict a target word instead of the using the hard segments of the source code sentences. Li et al. [116] points out the limitations of neural network language models. To overcome those problems, the authors proposed two new approaches: an attention mechanism enhanced LSTM and a pointer mixture network. The attention mechanism enhanced LSTM is used to alleviate fixed-size vectors and improve memorization capability by providing a variety of ways for gradients to back propagate. In contrast, the pointer mixture network predicts out of vocabulary (UoV) words by considering locally repeating tokens. That study also proposed the use of an abstract syntax tree (AST)-based code completion method.

Li et al. [117] presented a source code defect prediction model using a CNN is called DP-CNN. AST has used to convert the source code into token vectors. Using the word embedding map, each token vector is converted into a numerical vector. CNN used a numerical vector for training. Thereafter, the CNN model creates the source code's semantic and structural features. By comparing with the traditional defect prediction features, DP-CNN is improved the performances by 12% than other state-of-the-art and 16% than other traditional feature basis methods. Also, the CNN model [118] is used to identify the code algorithm based on structural features of the programming code. Dam et al. [119] presented a deep learning model for software defect prediction. The model has used the AST that incorporated with the LSTM network. Each node of the AST structure is treated as an LSTM unit. A deep tree-based LSTM model is stored syntactical and structural information of source codes for accurate prediction. The learning style of the tree-based LSTM model is unsupervised. The model does not clean or replace any erroneous words by predicting correct words. It is used to generate error probability from a source code thereafter a classifier identifies the source code's defect by using the value of probability.

Pham et al. [120] used a CNN as a language model based on FNN. Experimental results demonstrated that the performance of the CNN language model is better than the normal FNN. As for recurrent models, the CNN language model performs well than the RNN, but below

the state-of-the-art LSTM model. In [43], authors proposed an RNN based model for the source code fault prediction. There are two familiar evaluation methods such as the area under the curve (AUC) and F1-measures are used to measure the performance. The proposed attention-based RNN model improves the accuracy of source code classification. The AUC and F1-measure achieved 7% and 14% more accuracy than the other benchmark models.

In summary, a wide variety of methods and techniques have been proposed in various studies, most of which used RNN, LSTM, and CNN models for source code assessments and other applications. It is very difficult to explain which proposed research work is superior over other researches. RNNs perform comparatively better than conventional language models, but RNNs have limited ability to handle long source code inputs [37]. An LSTM is a special kind of RNN network that can remember longer source code sequences due to its extraordinary internal structure, and thus overcome RNN's shortcomings. The model proposed in this Chapter is unlike from other models. The proposed LSTM-AM further extends the capabilities of a standard LSTM to the point where it can be used for detecting and predicting solution code errors as well as code classification. Standard LSTM network only uses the last hidden state to make predictions. In contrast, LSTM-AM network can take the outcomes of all previous hidden states into consideration when making predictions. Therefore, it is a more promising technique for use in source code assessment than other state-of-the-art language models.

3.3 Overview of Language Model and Recurrent Neural Networks

3.3.1 *N*-gram Language Model

The resources of natural text corpora are being enriched by the accumulation of text from multiple sources on a daily basis. The success behind natural language processing is based on this rich text corpus. For this reason, and because of their simplicity and scalability, *n*-gram models are popular in the field of natural language processing. An *n*-gram model predicts the upcoming word or text of a sequence based on probability, and the probability of an entire word sequence $P(w_1, w_2, \dots, w_n)$ can be calculated by using the chain rule of probability.

$$P(W_1^n) = P(W_1)P(W_2|W_1)P(W_3|W_1^2) \cdots P(W_n|W_1^{n-1}) \quad (3.1)$$

$$P(W_1^n) = \prod_{i=1}^n P(W_i|W_1^{i-1}) \quad (3.2)$$

The Markov assumption, which is used when the probability of a word depends solely upon the previous word, is described in

$$P(W_n|W_1^{n-1}) \approx P(W_n|W_{n-1}) \quad (3.3)$$

Thus, the general equation of an n -gram used for the conditional probability of the next word sequence is as follows:

$$\prod_{i=1}^m W_i|W_1^{i-1} \approx \prod_{i=1}^m W_i|W_{i-n+1}^{i-1} \quad (3.4)$$

In practice, the maximum likelihood can be estimated by many smoothing techniques [121], as shown in the following equation:

$$P(W_i|W_{i-n+1}^{i-1}) = \frac{C(W_{i-n+1}^{i-1} W_i)}{C(W_{i-n+1}^{i-1})} \quad (3.5)$$

Cross-entropy is measured to validate the prediction goodness of a language model [122]. Low cross-entropy values imply better language models.

$$H_p \approx -\frac{1}{m} \sum_i^m \log_2 P(W_i|W_{i-n+1}^{i-1}) \quad (3.6)$$

3.3.2 Recurrent Neural Networks

An RNN is a neural network variant that is frequently used in natural language processing, classification, regression, etc. In a traditional neural network, inputs are processed through multiple hidden layers and then output via the output layer. In the case of sequential dependent input, a general neural network cannot manufacture accurate results. For example, in the case of the dependent sentence “*Rose is a beautiful flower*”, a general neural network takes the “*Rose*” input to produce an output based solely on “*Rose*”. Then, when the word “*is*” input is considered, the network does not use the previous of “*Rose*” result. Instead it simply produces the result using the word “*is*”. Similarly, a simple neural network takes other words “*a*”, “*beautiful*”, and “*flower*” to generate results without considering previous result of inputs. To address this problem RNN has emerged with an internal memory that retains previous time step results. A simple RNN structure is shown in Figure 3.1.

Mathematically, an RNN can be presented using equation (3.7). The current state of the

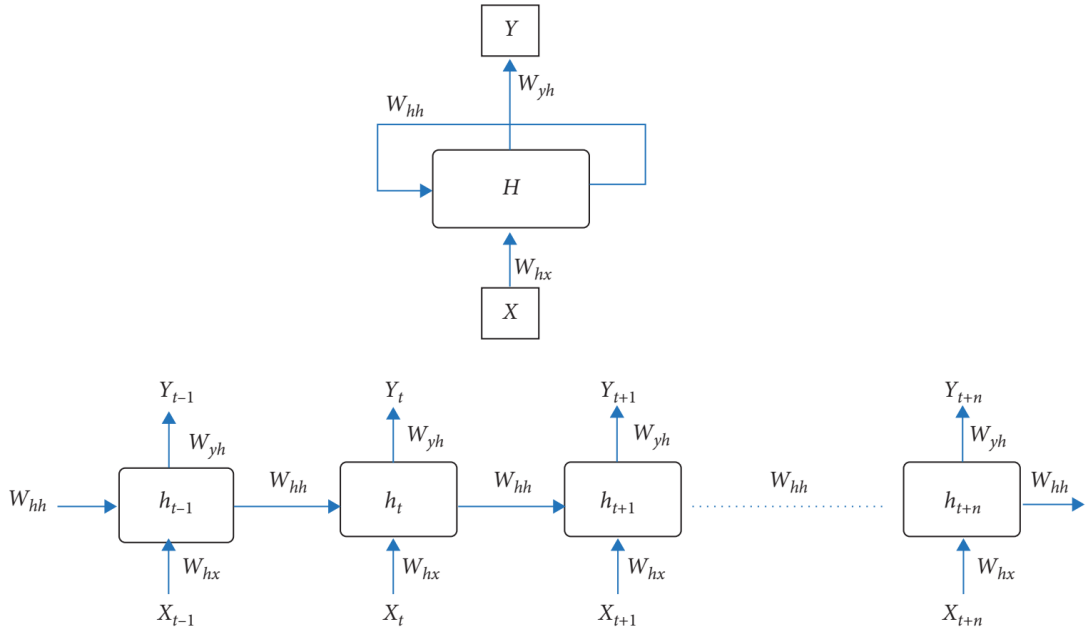


Figure 3.1: A simple RNN structure

RNN can be expressed as

$$h_t = f(h_{t-1}, x_t) \tag{3.7}$$

$$h_t = f(w_{hh}h_{t-1} + w_{hx}x_t) \tag{3.8}$$

where h_t is the current state, h_{t-1} is the previous state, x_t is the current state input, w_{hh} is the weight of recurrent neuron and w_{hx} is the weight of input neuron.

Equation (3.9) is used as an activation function (*tanh*) of RNN:

$$h_t = \tanh(w_{hh}h_{t-1} + w_{hx}x_t) \tag{3.9}$$

Finally, the output function can be written as follows:

$$y_t = \text{softmax}(w_{yh}h_t) \tag{3.10}$$

where w_{yh} is the weight for the output layer and y_t is the output.

RNN has multiple input and output types such as one to one, one to many, many to one, and many to many. Despite all the advantages, RNN is susceptible to the major drawbacks of gradient vanishing or exploding.

3.3.3 Gradient Vanishing and Exploding

In this section, we discuss the gradient vanishing and exploding problem of RNN. It seems simple to train the RNN network, but it is very hard because of its recurrent connection. In case of forward propagation, multiply all the weight metrics and a similar procedure needs to apply for the backpropagation. For the backpropagation, the signal may be strong or weak that causes the exploding and vanishing. Gradient vanishing makes a complex situation to determine the direction of model parameters to improve the loss function. On the other hand, exploding gradients make the learning condition unstable. Training of the hidden RNN network is passed through different time steps using backpropagation. The sum of distinct gradient at every time step is equal to the total error gradient. The error can be expressed by considering total time steps T in the following equation:

$$\frac{\partial E}{\partial L} = \sum_{t=1}^T \frac{\partial E_t}{\partial L} \quad (3.11)$$

Now, apply the chain rule to calculate the overall error gradients:

$$\frac{\partial E}{\partial L} = \sum_{t=1}^T \frac{\partial E}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial L} \quad (3.12)$$

The term $\frac{\partial h_t}{\partial h_k}$ is involved with the product of Jacobians $\frac{\partial h_i}{\partial h_{i-1}}$ as shown in the following equation:

$$\frac{\partial h_t}{\partial h_j} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial h_{t-3}} \frac{\partial h_{t-3}}{\partial h_{t-4}} \dots \frac{\partial h_{j+1}}{\partial h_j} \quad (3.13)$$

$$\frac{\partial h_t}{\partial h_j} = \prod_{i=j+1}^t \frac{\partial h_i}{\partial h_{i-1}} \quad (3.14)$$

The term $\frac{\partial h_t}{\partial h_{t-1}}$ in equation (3.13) is evaluated by equation (3.8).

$$\prod_{i=j+1}^t \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=j+1}^t W^T \text{diag}[f'(h_{i-1})] \quad (3.15)$$

Now, by the Eigen decomposition on the Jacobian matrix $\frac{\partial h_t}{\partial h_{t-1}}$ given by $W^T \text{diag}[f'(h_{i-1})]$, we obtain the eigenvalues $\lambda_1, \lambda_2, \lambda_3 \dots, \lambda_n$ where $\lambda_1 > \lambda_2 > \lambda_3 \dots > \lambda_n$ and the corresponding eigenvectors are $v_1, v_2, v_3 \dots, v_n$. If the direction of a hidden state Δh_t is moved to v_i by any modifications, then the gradient will be $\lambda_i \Delta h_t$. From equation (3.15) the product of the

Jacobians of the hidden state sequences is $\lambda_i^1 \Delta h_1, \lambda_i^2 \Delta h_2, \lambda_i^3 \Delta h_3 \dots, \lambda_i^n \Delta h_n$. It is easy to visualize the term λ_i^t dominating Δh_t . In summary, if the greatest eigenvalue is $\lambda_1 < 1$, then the gradient will vanish and $\lambda_1 > 1$ causes the gradient exploding [123]. To alleviate the gradient vanishing or exploding problems, the gradient clipping, input reversal, identity initialization, weight regularization, LSTM, etc. techniques can be used.

3.3.4 Long Short-Term Memory Neural Network

An LSTM neural network is a special kind of RNN network that is often used to process long inputs. An LSTM is not limited to a single input, but can also process complete input sequences. Usually an LSTM is structured with four gates such as forget, input, cell state and output. Each gate has a separate activity where the cell state keeps complete information of the input sequences and others are used to manage the input and output activities. Figure 3.2 shows the structure of a basic LSTM unit.

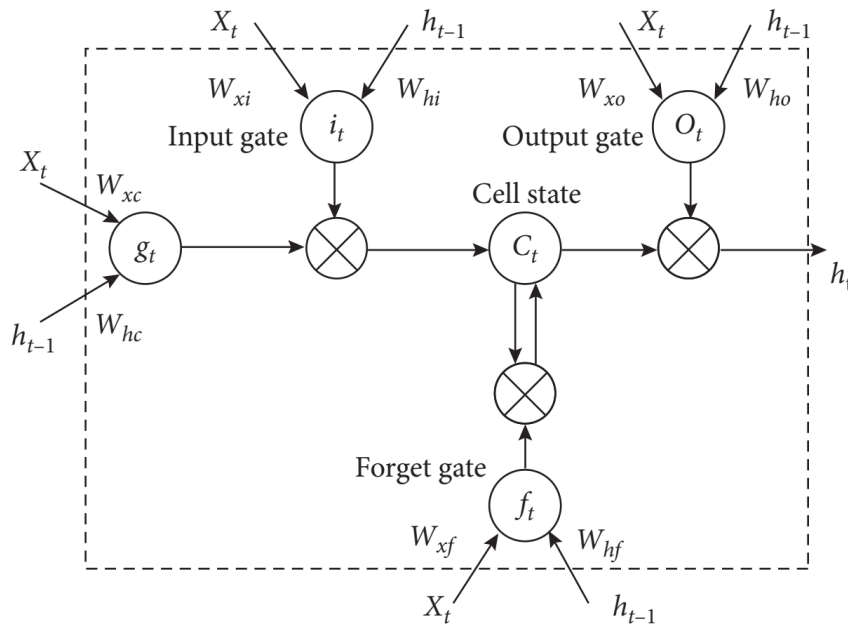


Figure 3.2: Internal structure of an LSTM unit

At the very beginning, processing starts with the forget gate to determine which information to discard from (or retain in) the cell state. The forget gate in cell state c_{t-1} can be expressed by the following equation (3.16) where h_{t-1} is hidden state and x_t is current input. The output (0 or 1) of the forget gate is produced through a sigmoid function. If the result of the forget gate is 1 then keep the data in the cell state otherwise discard the data.

$$f_t = \sigma(w_f \cdot [h_{t-1}, x_t] + b_f) \quad (3.16)$$

The input gate determines which cell state value should be updated when new data appears. Through the tanh function, the candidate value \tilde{c}_t for the cell state is now created.

$$i_t = \sigma(w_i \cdot [h_{t-1}, x_t] + b_i) \quad (3.17)$$

$$\tilde{c}_t = \tanh(w_c \cdot [h_{t-1}, x_t] + b_c) \quad (3.18)$$

Now, the old cell state c_{t-1} is updated by the c_t .

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \quad (3.19)$$

The filtered version of the cell state will be output o_t via the sigmoid function and the weight will also be updated.

$$o_t = \sigma(w_o \cdot [h_{t-1}, x_t] + b_o) \quad (3.20)$$

$$h_t = o_t * \tanh c_t \quad (3.21)$$

Recognizing the strength of LSTM, we were motivated to apply this neural network to error detection, prediction, correction, and classification in source codes.

3.4 Approach

The proposed LSTM-AM network has an effective deep learning architecture that is used as an intelligent support model for source code assessment. Accordingly, we trained the model using source codes and then used it successfully detect errors and predict correct words in erroneous source codes based on trained corpus. Moreover, proposed model can classify the source codes using the prediction results. The proposed model can produce correct words for each erroneous source code after assessment where learners and professionals can benefit from it. The workflow of the proposed model is depicted in Figure 3.3.

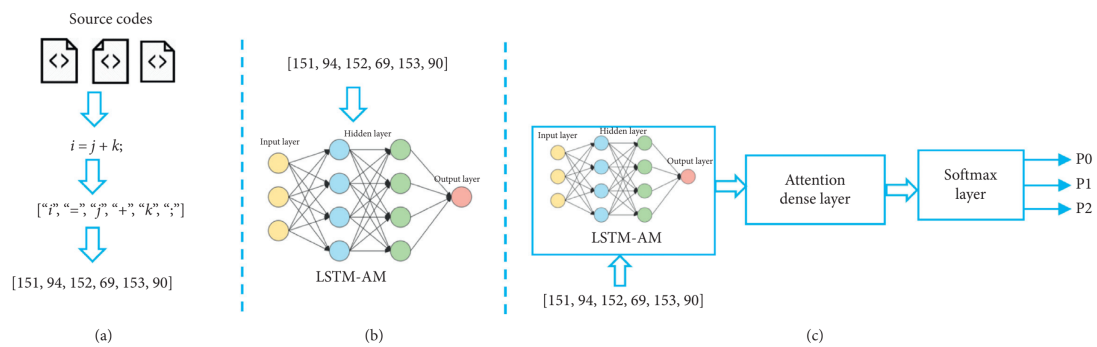


Figure 3.3: The main workflow of our model: (a) conversion of source code to token IDs, (b) model training using token IDs, and (c) results produced by the softmax function.

3.4.1 Proposed LSTM-AM Neural Network Architecture

Over the years, attention mechanisms have been adapted to a wide variety of diverse tasks [124–131], the most popular and effective of which is seq2seq modeling. Typically, in seq2seq modeling the output of the last hidden state is used as the context vector for further consideration. It is very difficult to process long sequenced inputs using the seq2seq model [132]. The attention mechanism makes it possible to map all previous hidden state outputs, including the latest hidden state output, to produce the most relevant and accurate results.

With this point in mind, we incorporated an attention mechanism into a standard LSTM to make LSTM-AM model, as shown in Figure 3.4. This strengthens the proposed model’s ability to predict longer source code sequences. Usually, attention improves the performance of the language and translator model by merging all hidden state outputs with the softmax function, sometimes attention mechanism work as a dense layer. Recently, attention mechanisms have been used in machine translation tasks with great success. Furthermore, sometimes it is necessary for a machine translator model to compress entire input sequences into a smaller size vector, so there is a possibility of information loss. The use of attention mechanisms has fixed this problem. Although the abilities of a standard LSTM to capture long-range dependencies are far superior to those of an RNN. It still encounters when a hidden state has to carry all the necessary data in a small-sized vector [132]. The introduction of attention mechanisms and their alignment with neural language models such as LSTM are aimed at overcoming these problems [115]. The attention mechanism offers neural language models to bring and use appropriate information in all secret states of the past. As a result, the network’s retention ability is improved and diverse paths are provided for gradients to back propagate. More detailed mathematical illustrations of attention mechanism can be found in [116].

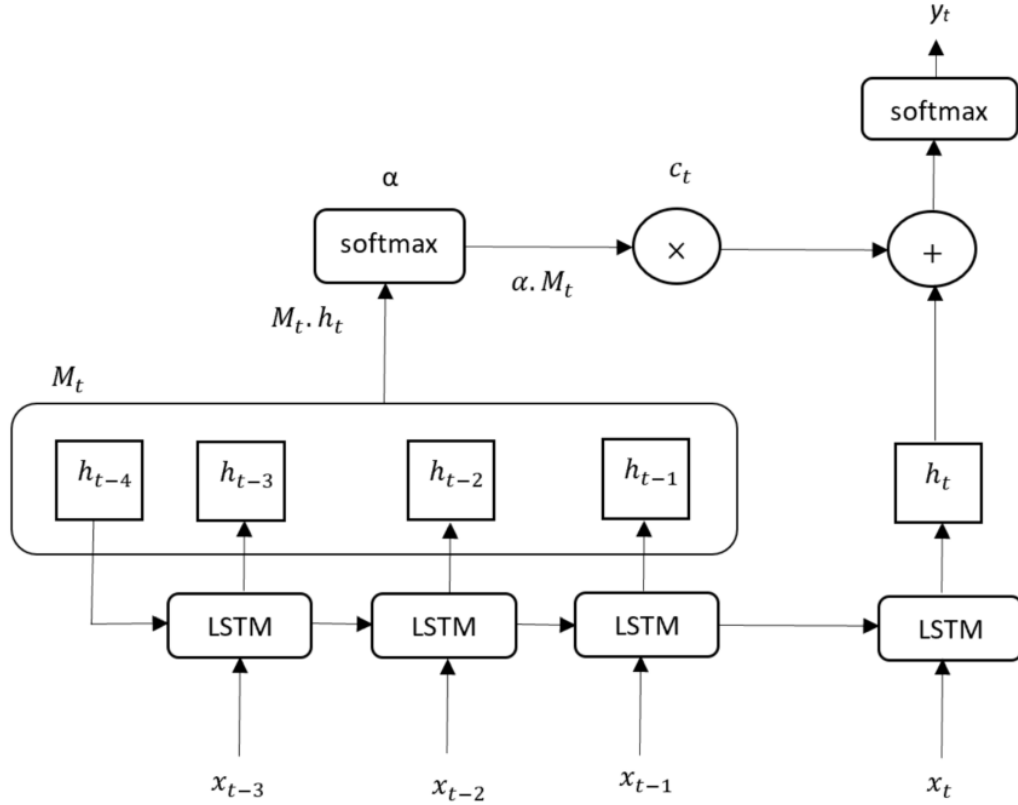


Figure 3.4: An architecture of the proposed LSTM-AM network

For our attention mechanism, we took the external memory of E for the previous hidden states, which is denoted as $M_t = [h_{t-E} \cdots h_{t-1}] \in \mathbb{R}^{k \times E}$. The proposed model used attention layer by considering h_t and M_t at the time t , attention weight α_t , and context vector C_t .

$$A_t = M_t \cdot h_t \quad (3.22)$$

$$\alpha_t = \text{softmax}(A_t) \quad (3.23)$$

$$C_t = M_t \alpha_t^T \quad (3.24)$$

To predict the next word at time step t , judgment is based not only on current hidden states h_t but also on context vector C_t . At that point, focus turns to the vocabulary spaces to produce the final probability $y_t \in \mathbb{R}^v$ via softmax function. Here, G_t is an output vector.

$$G_t = \text{tanh}(w^g[w^h(h_t) + w^m(c_t)]) \quad (3.25)$$

$$y_t = \text{softmax}(w^v G_t + b^v) \quad (3.26)$$

where $w^g \in \mathbb{R}^{k \times 2k}$ and $w^v \in \mathbb{R}^{v \times k}$ are trainable projection matrices, and $b^v \in \mathbb{R}^v$ is a bias and v is a vocabulary/dictionary size.

Based on the above aspects, we can see that the use of an attention mechanism helps to effectively extract the exact features from input sequences. As such, the use of LSTM-AM can increase the capability of the proposed model.

3.5 Data Collection and Problem Description

An OJ system is a web-based programming environment that compiles and executes submitted solution codes and returns judgments based on test data sets. OJ system is an open platform for programming practice as well as competition. To conduct the experimental work, source codes are collected from the AOJ system [18, 19]. Currently, the AOJ system is effortlessly performing for various programming competitions and academies. As of May 2021, about 100,000 users are regularly playing their programming activities on the AOJ platform, with 2500 autonomous problem sets. All problem sets are classified based on different algorithms and branches of computer science [63]. As a result, about 5.5 million solution source codes have been archived on the AOJ platform, encouraging better research in the field of programming. All the experimental solution codes are collected from the AOJ system for training, validation, and testing purposes. For model training, we selected all of the correct solution codes written in C language of the three problems such as Greatest Common Divisor (GCD), Insertion Sort (IS), and Prime Numbers (PN). There are a total of 2285 correct source codes for the IS problem and the overall solution success rate is 35.16%. The total number of correct source codes for the GCD problem is 1821 and the overall solution success rate is 49.86%. Considering the GCD problem, we see that there are two inputs (a and b) given in a line, after which the greatest common divisor of a and b will be output, as shown in Figure 3.5(a).

The total number of correct source codes for the PN problem is 1538 and the overall solution success rate is 30.8%. In the PN problem description, the first line contains an integer N . The code needs to count the number of prime numbers in the following list of N elements, as shown in Figure 3.5(b).

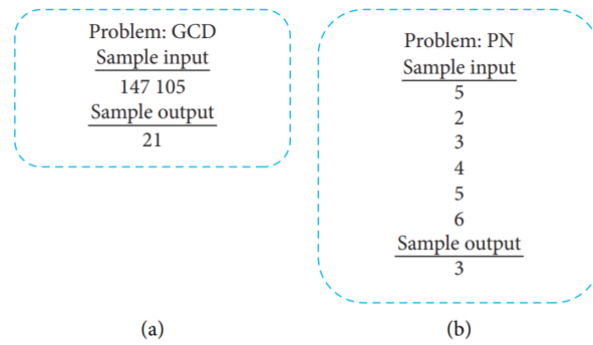


Figure 3.5: Input and output sample of GCD and PN problems

3.5.1 Data Preprocessing and Training

Before the model training, raw source codes were filtered by removing unnecessary elements. To accomplish this, we followed the procedure applied to [63] for source code embedding and tokenization. First, we removed all irrelevant elements from codes like lines (`\n`), comments, tabs (`\t`). After that, all the remaining elements of the code were converted into word sequences where numbers, functions, tokens, keywords, variables, classes, and characters were treated as simple words. The whole code transformation process is called tokenization and vocabulary creation. Then, each word was encoded with IDs in which the function names, keywords, variable names, and characters were encoded as listed in Table 3.1.

The flowchart of the training and evaluation process of the proposed model is shown in Figure 3.6.

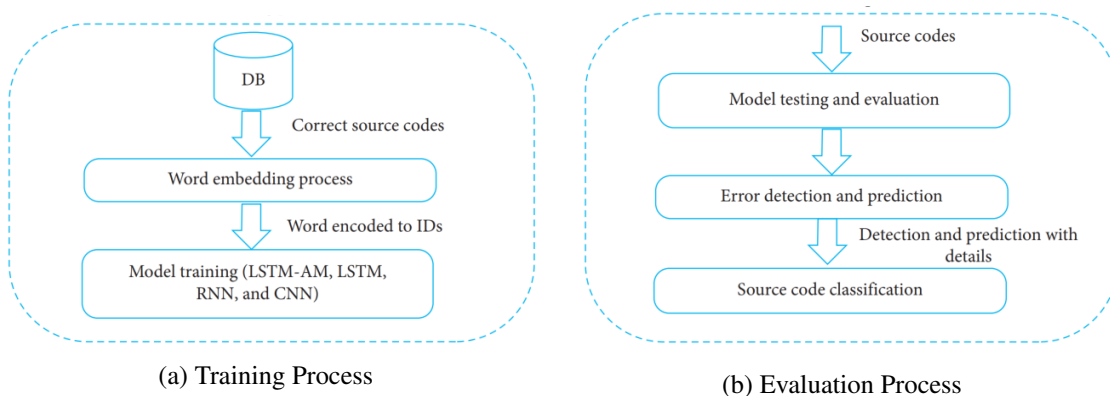


Figure 3.6: The flowchart of the training and evaluation process of the proposed model

At the very beginning of the training phase, the source codes were first converted into word sequences, and then encoded into token IDs as shown in Figure 3.7. This conversion process is called word embedding and tokenization.

Table 3.1: A partial list of Ids for characters, special characters, numbers, keywords

| ID | Word | ID | Word | ID | Word | ID | Word |
|-----|----------|-----|----------|-----|------|-----|------|
| 30 | auto | 46 | int | 62 | | 78 | . |
| 31 | break | 47 | long | 63 | ! | 79 | / |
| 32 | case | 48 | register | 64 | ? | 80 | 0 |
| 33 | char | 49 | return | 65 | - | 81 | 1 |
| 34 | const | 50 | short | 66 | “ | 82 | 2 |
| 35 | continue | 51 | signed | 67 | # | 83 | 3 |
| 36 | default | 52 | sizeof | 68 | \$ | 84 | 4 |
| 37 | do | 53 | static | 69 | % | 85 | 5 |
| 38 | double | 54 | struct | 70 | & | 86 | 6 |
| 39 | else | 55 | switch | 71 | ‘ | 87 | 7 |
| 40 | enum | 56 | typedef | 72 | (| 88 | 8 |
| 41 | extern | 57 | union | 73 |) | 89 | 9 |
| 42 | float | 58 | unsigned | 74 | * | 90 | ; |
| 43 | for | 59 | void | 75 | + | 91 | : |
| 44 | goto | 60 | volatile | 76 | , | 92 | < |
| 45 | if | 61 | while | 77 | ~ | 93 | > |
| 94 | = | 110 | O | 126 | ‘ | 142 | p |
| 95 | @ | 111 | P | 127 | a | 143 | q |
| 96 | A | 112 | Q | 128 | b | 144 | r |
| 97 | B | 113 | R | 129 | c | 145 | s |
| 98 | C | 114 | S | 130 | d | 146 | t |
| 99 | D | 115 | T | 131 | e | 147 | u |
| 100 | E | 116 | U | 132 | f | 148 | v |
| 101 | F | 117 | V | 133 | g | 149 | w |
| 102 | G | 118 | W | 134 | h | 150 | x |
| 103 | H | 119 | X | 135 | i | 151 | y |
| 104 | I | 120 | Y | 136 | j | 152 | z |
| 105 | J | 121 | Z | 137 | k | 153 | { |
| 106 | K | 122 | [| 138 | l | 154 | |
| 107 | L | 123 | \ | 139 | m | 155 | } |
| 108 | M | 124 |] | 140 | n | | |
| 109 | N | 125 | ^ | 141 | o | | |

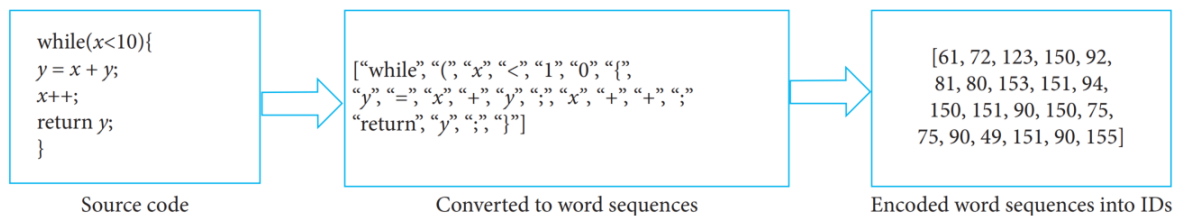


Figure 3.7: Source code word embedding and tokenization process

Upon completion of the embedding and tokenization process, we trained the proposed model and other related state-of-the-art models with the correct source codes of IS, GCD, and PN problems. The simple training process of an LSTM-based language model is shown in Figure 3.8.

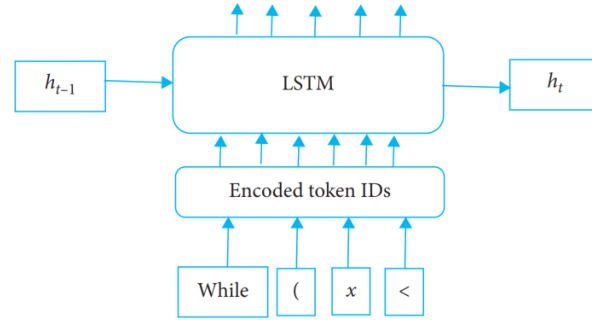


Figure 3.8: Training process of an LSTM language model

At the end of the training process, the next step is to check the performance of the model for the source code assessment task. How accurately identify errors and predict corrections? The proposed model created the probability for each word. A word is considered as an error candidate if its probability is below 0.1 [63]. Additionally, the cross-entropy for each epoch at the softmax layer is calculated to test the model loss function. Cross-entropy is defined as the difference between actual and predicted results. Softmax is an activation function that creates probabilities. Typically, softmax is used as the last layer of neural networks. The output range of the softmax function is between 0 and 1. The softmax layer received $x = [x_1, x_2, x_3, \dots, x_n]$ and returns probability $p = [p_1, p_2, p_3, \dots, p_n]$, as defined in equation (3.27). Cross-entropy is an effective performance measurement indicator for the probability-based model. Cross-entropy is calculated by the equation (3.6). Low-valued cross-entropy indicates a better model.

$$P_i = \frac{\exp(x_i)}{\sum_{j=1}^k \exp(x_j)} \quad (3.27)$$

A simple example of the prediction process used by the proposed model is shown in Figure 3.9. An example of input sequence is {"=", "x", "+", "y"} then model calculates the next probable correct word based on the source code corpus. Finally, the word with the highest probability is the winner of the next predicted word. Based on the input sequence in the example above, the correct predicted word is {";"}.

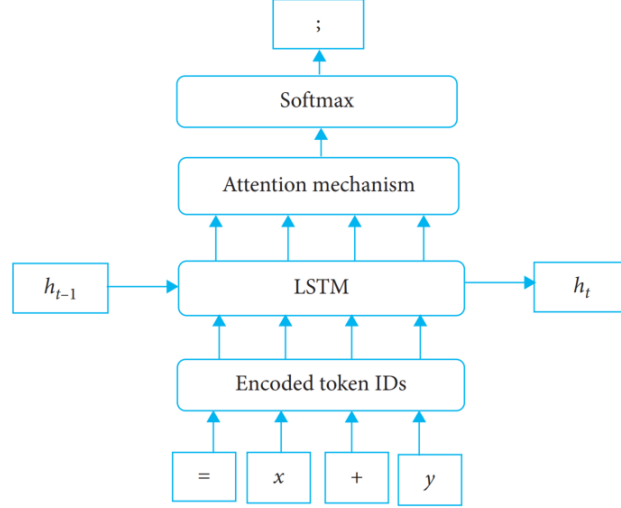


Figure 3.9: LSTM-AM network prediction process

3.6 Evaluation Metrics

Our primary goal is to evaluate the performance of the proposed model in terms of how accurately it assesses and detects errors in source code as well as classification. For that, we adopted three evaluation indices such as correction words accuracy (CWA), error detection accuracy (EDA), and model accuracy (MA), shown in equations 3.28, 3.29, and 3.30. In particular evaluated the proposed model and other benchmark models using Equation (3.30).

$$EDA = \frac{\text{True Errors (TE)}}{\text{Total Detected Errors (TDE)}} * 100\% \quad (3.28)$$

$$CWA = \frac{\text{True Correct Words (TCW)}}{\text{Total Predicted Words (TPW)}} * 100\% \quad (3.29)$$

$$MA = \frac{EDA + CWA}{2} \quad (3.30)$$

In most cases, the proposed model detects potential errors in the codes. Among these errors, there are a few actual errors, which are called True Errors (TE). Similarly, among the total predicted words, there are a few actual correct words, which are called True Correct Words (TCW).

The precision, recall, and f-measure scores are used to evaluate the classification performance. The formulas for precision, recall, and f-measure are shown in Equations (3.31), (3.32), and (3.33), respectively.

$$Precision(P) = \frac{tp_{e \rightarrow e}}{tp_{e \rightarrow e} + fp_{c \rightarrow e}} \quad (3.31)$$

$$Recall(R) = \frac{tp_{e \rightarrow e}}{tp_{e \rightarrow e} + fn_{e \rightarrow c}} \quad (3.32)$$

$$F - measure = \frac{2 * P * R}{P + R} \quad (3.33)$$

where $tp_{e \rightarrow e}$ is called true positive, the case $e \rightarrow e$ means defective source code classifies as erroneous, $fp_{c \rightarrow e}$ is called false positive, the case $c \rightarrow e$ means clean source code classifies as erroneous. The term $fn_{e \rightarrow c}$ is called false negative where $e \rightarrow c$ means erroneous source code classifies as a clean source code. F-measure is called the harmonic mean of recall and precision. Generally, it is difficult to achieve optimal results simultaneously for recall and precision. For example, if all the source codes are classified as defective, the resulted recall score will be 100% where the precision score will be small. Therefore, F-measure is a trade-off between recall and precision. The range of the F-measure score is between 1 and 0, the higher score implies to the better classification model.

3.7 Experimental Results

The proposed intelligent support model can be useful for source code assessment. Moreover, it is a general model and can be adapted to different types of programming language-based source codes for model training and testing. In the proposed model, we defined a minimum probability value by which the model can identify error candidate words based on the training corpus. Accordingly, some incorrect source codes (IS, GCD, and PN) are used to evaluate the model performance. Here, we should note that all of our research work and language model training were performed on an Intel® Core(TM) i7-5600U central processing unit (CPU) personal computer clocked at 2.60GHz with 8GB of RAM in a 64-bit Windows 10 operating system.

3.7.1 Hyperparameters

For the proposed model, we defined several experimental hyperparameters in order to obtain better results. To avoid overfitting, a dropout ratio (0.3-0.5) was used for the proposed model.

The LSTM network was optimized using Adam, which is a stochastic optimization method [133, 134]. The learning rate is an important factor for neural network training because the value of the learning rate can control the learning speed of the model. Network learning becomes faster and slower on the basis of higher and lower value of learning rates, respectively. For the proposed model, we determine the learning rate $l = 0.002$ and the network weights during training are updated by the value of l . β_1 is the exponential decay rate for the first-moment estimate and the second-moment estimate of the exponential decay rate is β_2 . The values of the β_1 and β_2 are 0.001 and 0.999 respectively. The value of ϵ is chosen to avoid any division by zero which is $\epsilon = 1e^{-8}$. We trained the proposed network in 50, 100, 150 and 200 hidden units. Each model type was named in reference to the number of units, such as the 100-unit model, 200-unit model, and so on. After training, we assessed the ability of the proposed LSTM-AM model to pick the best number of hidden units from the created models.

3.7.2 Selection of Hidden Units and Cross-entropy Measurement

Various number of hidden units including 50, 100, 150, and 200 are used to train the proposed LSTM-AM and other state-of-the-art models. In training, the source codes of IS, GCD, and PN problems are used separately as well as all the source codes of IS, GCD, and PN are used combinedly. The number of source codes of each type of problem is listed in Table 3.2.

Table 3.2: Number of source codes of each problem

| Problem Type | Number of Source Codes |
|-------------------------------|-------------------------------|
| Greatest Common Divisor (GCD) | 964 |
| Insertion Sort (IS) | 1518 |
| Prime Numbers (PN) | 972 |
| Total | 3454 |

We trained the proposed LSTM-AM and different state-of-the-art models using the source codes. Table 3.3 is presented the cross-entropy in 30 epochs during training using PN source codes. The 50-, 100-, 150- and 200-unit models took total 11483, 20909, 38043, 59065 seconds to train the LSTM-AM model using the PN problem, respectively.

Tables 3.4 and 3.5 are presented cross-entropy of different models during training using GCD and IS source codes respectively. The 50-, 100-, 150- and 200-unit models took total 19005, 24110, 24273, 30420 seconds to train the LSTM-AM model using the GCD problem, respectively and for the IS problem it took total 39643, 62756, 80100, 100803 seconds respectively. In contrast, other models such as LSTM and RNN took relatively less time for training.

Table 3.3: Cross-entropy comparison using PN source codes

| Model | Units (Neurons) | | | |
|---------|-----------------|------|------|------|
| | 50 | 100 | 150 | 200 |
| RNN | 6.35 | 4.72 | 4.21 | 3.95 |
| LSTM | 4.75 | 3.31 | 2.37 | 2.02 |
| LSTM-AM | 4.35 | 3.90 | 2.87 | 2.23 |

Table 3.4: Cross-entropy comparison using GCD source codes

| Model | Units (Neurons) | | | |
|---------|-----------------|------|------|------|
| | 50 | 100 | 150 | 200 |
| RNN | 5.11 | 4.36 | 3.50 | 3.23 |
| LSTM | 2.56 | 1.91 | 1.80 | 1.39 |
| LSTM-AM | 2.22 | 1.80 | 1.75 | 1.30 |

To evaluate the efficiency of the proposed model, epoch wise cross-entropy during the training periods using 200-unit was calculated which is depicted in Figure 3.10.

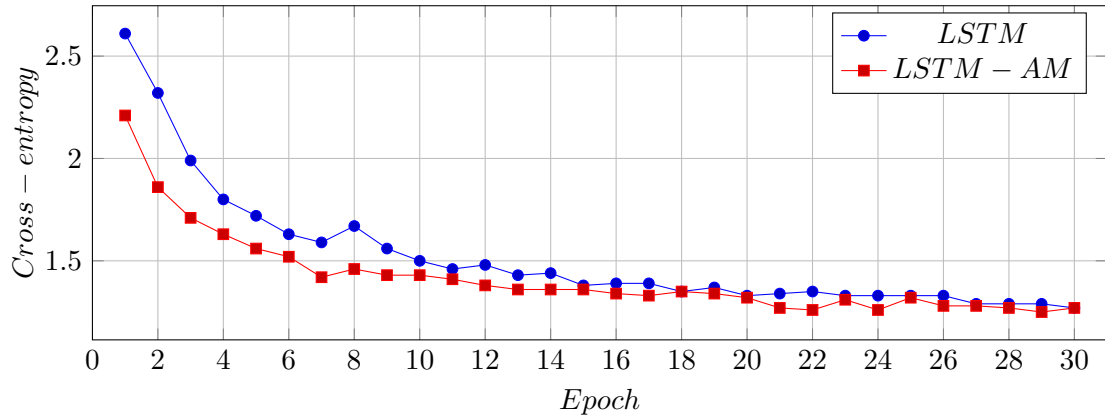
As mentioned above, the efficiency of a model strongly depends upon the value of cross-entropy. During training, the 200-unit model produced the lowest cross-entropy using each type of problem set. The cross-entropy of the 200-unit model using IS, PN and GCD problems are shown in Figure 3.11.

We aimed to find the best-suited hidden units for the LSTM-AM model and other state-of-the-art models. In this regard, we put together all the source codes (about 3442) to train the proposed and other state-of-the-art models. The cross-entropies and total times are recorded at the last epoch of all the models as presented in Table 3.6. The cross-entropy of the 200-unit model is lower than other models.

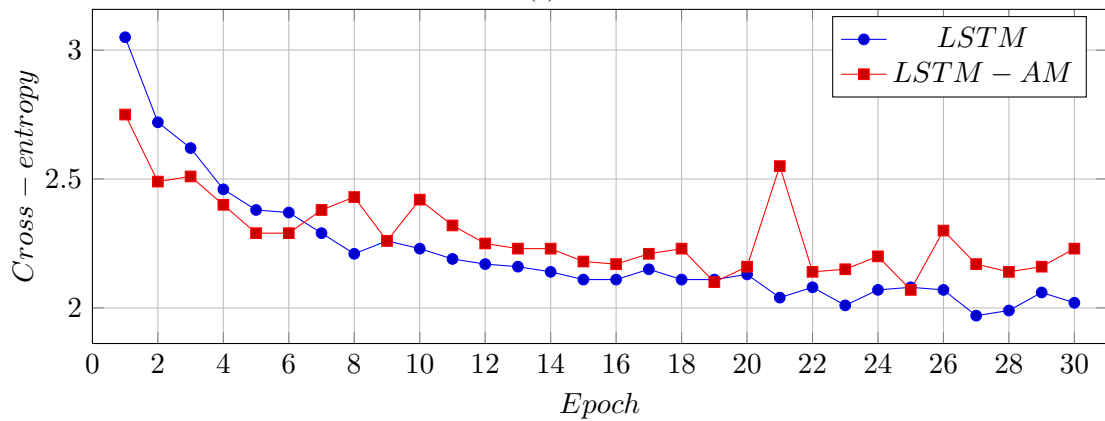
Based on the above aspect, it is ensured that the 200-unit model provides the best results because its cross-entropy is the lowest value among all the units, thus we selected a 200-unit for the LSTM-AM and other state-of-the-art models.

Table 3.5: Cross-entropy comparison using IS source codes

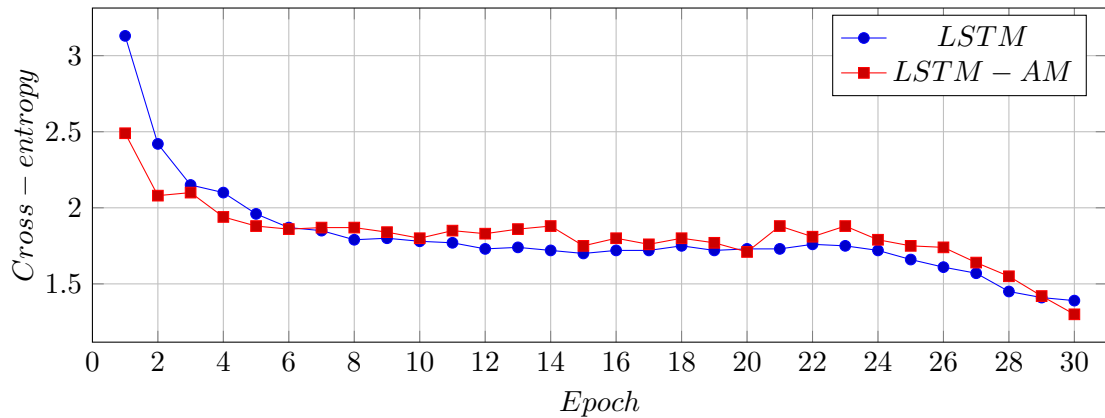
| Model | Units (Neurons) | | | |
|---------|-----------------|------|------|------|
| | 50 | 100 | 150 | 200 |
| RNN | 4.99 | 3.78 | 2.89 | 3.11 |
| LSTM | 3.26 | 1.63 | 1.48 | 1.26 |
| LSTM-AM | 3.12 | 1.55 | 1.40 | 1.27 |



(a) IS



(b) PN



(c) GCD

Figure 3.10: Epoch-wise cross-entropies of 200-unit model using a) IS, b) PN, and c) GCD source codes

Table 3.6: Cross-entropy comparison of different models using all source codes

| Model | Units (Neurons) | | | |
|---------|-----------------|------|-------|------|
| | 50 | 100 | 150 | 200 |
| RNN | 5.11 | 4.36 | 3.87 | 3.53 |
| LSTM | 3.89 | 3.26 | 2.500 | 2.31 |
| LSTM-AM | 3.96 | 2.99 | 2.75 | 2.17 |

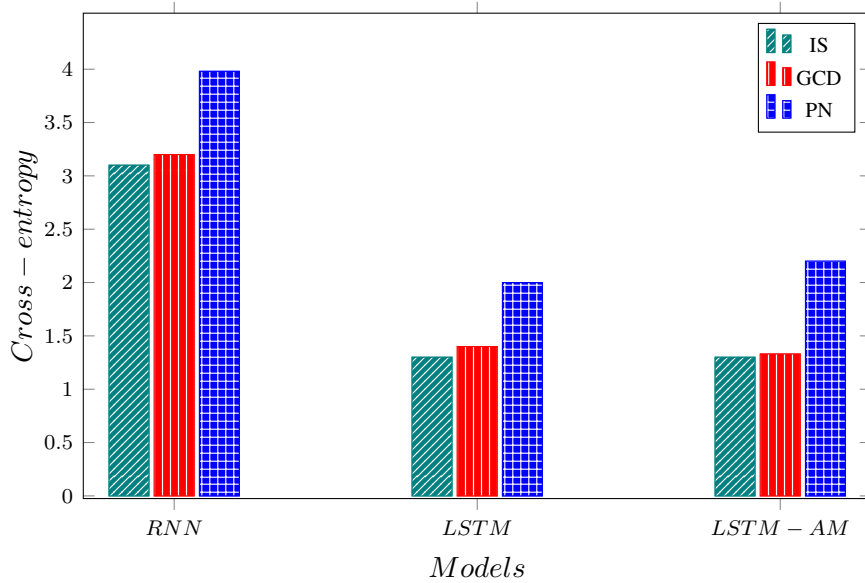


Figure 3.11: Cross-entropies of the 200-unit model using IS, GCD, and PN problems

3.7.3 Error Detection and Correction Word Prediction

In the evaluation process, we tested LSTM-AM and other state-of-the-art models using erroneous source codes. Probable error locations were marked by changing the text color and underlining the suspected erroneous portions. Also, the proposed model generates error candidate words and predicted words' probability. Since both the standard LSTM and the LSTM-AM networks identified source code errors quite well than the RNN and other networks when the 200-unit model was used. Therefore, 200-unit model was used in all the empirical experiments.

```

1 #include <stdio.h>                               11 }
2 int gcd(a,b){                                     12 int main()
3   if (b==0){                                       13 {
4     return a;                                       14   int a,b,c;
5   }                                                 15   scanf("%d",&a);
6   if (a<b){                                         16   scanf("%d",&c);
7     return gcd(b,a%b);                               17   printf("%d\n",gcd(a,b));
8   }else{                                           18
9     return gcd(a,b%a);                               19   return 0;
10  }                                                 20 }

```

Figure 3.12: Erroneous source code evaluated by the standard LSTM

An erroneous source code sequence evaluated by the standard LSTM network is shown in Figure 3.12. Here, it can be seen that errors were detected in lines 2, 6, 15, and 16. In line 2, the word “a” in the “gcd” function was detected as an error candidate, after which the correct

word was predicted to be “)” with a probability 0.62435395. The model decided that the “gcd” function might be without arguments, the word “)” was predicted instead of the word “a”. In line 6, the error word is “if” and the predicted word is “else” with a probability 0.5808078. Additionally, in line 15 the predicted word is “blank space” in the place of “double quotation”. Finally, in line 16, the model detected “c” as an error element and suggested with a high rate of probability that it be replaced by the word “b”. The word “c” is irrelevant within the context of the program, it can be confirmed that the standard LSTM model successfully detected the error candidates shown in Figure 3.12, as listed in Table 3.7.

Table 3.7: List of detected errors and predicted words in Figure 3.12 by the LSTM model

| Erroneous words | Erroneous word's probability | Location | Predicted words | Probability |
|-------------------------|------------------------------|----------|--------------------|-------------|
| <i>a</i> | 0.000496462 | 2 |) | 0.6243539 |
| <i>if</i> | 0.014852316 | 6 | <i>else</i> | 0.5808078 |
| <i>double quotation</i> | 0.029112648 | 15 | <i>blank space</i> | 0.6583209 |
| <i>c</i> | $8.5894303e^{-10}$ | 16 | <i>b</i> | 0.9261215 |

```

1 #include <stdio.h>                                11 }
2 int gcd(a,b){                                     12 int main()
3 if (b==0){                                        13 {
4 return a;                                         14 int a,b,c;
5 }                                                 15 scanf("%d",&a);
6 if (a<b){                                         16 scanf("%d",&c);
7 return gcd(b,a%b);                                17 printf("%d\n",gcd(a,b));
8 }else{                                           18
9 return gcd(a,b%a);                                19 return 0;
10 }                                                20 }

```

Figure 3.13: Erroneous source code evaluated by the LSTM-AM

The same incorrect source code was then evaluated by the LSTM-AM network, as shown in Figure 3.13. The error locations are in lines 2, 15, and 16. The word “a” in the “gcd” function was detected as an error candidate and the predicted word “)” was suggested. In line 15, the word “double quotation” was identified as a bug and the predicted word “blank space” was suggested. The word “c” in line 16 was recognized as an error and the corresponding predicted word “b” was suggested with a probability of 0.9863272, as shown in Table 3.8.

Another erroneous source code, which exist some logical errors (*WA*), was evaluated by the standard LSTM network, as shown in Figure 3.14. All the detected error words and their corresponding predicted words of Figure 3.14 are listed in Table 3.9.

Table 3.8: List of detected errors and predicted words in Figure 3.13 by the LSTM-AM model

| Erroneous words | Erroneous word's probability | Location | Predicted words | Probability |
|-------------------------|------------------------------|----------|--------------------|-------------|
| <i>a</i> | 0.000309510 | 2 |) | 0.5722154 |
| <i>double quotation</i> | 0.045484796 | 15 | <i>blank space</i> | 0.7051629 |
| <i>c</i> | $2.838025e^{-07}$ | 16 | <i>b</i> | 0.9863272 |

```

1 #include <stdio.h>          22
2                               23 rmd = y % x;
3 int    main(void)          24 if( rmd == 0 )
4 {                               25 {
5     int    x, y;           26     printf("%d\n",y);
6     int    a, b;           27 }
7                               28 else
8     int    rmd;           29 {
9     int    i;             30
10    int    rst;           31     for(i=1; i<=rmd; i++)
11                               32     {
12     scanf("%d %d",&a, &b); 33         if((( y % i) == 0) && (( rmd % i) == 0))
13     if( a >= b )          34         {
14     {                               35             rst = i;
15         x = a;           36         }
16         y = b;           37     }
17     }else                 38     printf("%d\n",rst);
18     {                               39     }
19         x = b;           40     return 0;
20         y = a;           41 }
21     }

```

Figure 3.14: Erroneous source code evaluated by the LSTM

Similarly, the same erroneous source code was tested by the LSTM-AM network, as shown in Figure 3.15. The detailed error descriptions of Figure 3.15 are listed in Table 3.10, where it can be seen that the LSTM-AM network detected all of the potential error candidates, including the true logical error (*WA*) candidate, successfully.

For further experiments, we trained the proposed model with additional datasets including IS, GCD, PN, Bubble Sort (BS) and Selection Sort (SS). Here, we tested the accuracy of error detection in source codes by the proposed model according to equation (3.28). In this case, we used faulty source codes that received *CE* verdicts for syntax errors and *WA*, *TLE*, *MLE*, *PrE*, and *RTE* verdicts for various logical errors from the AOJ. The assessment results of detecting syntax errors (*CE*) and logic errors (*WA*, *TLE*, *MLE*, *PrE*, and *RTE*) using the

Table 3.9: List of detected errors and predicted words in Figure 3.14 by the LSTM model

| Erroneous words | Erroneous word's probability | Location | Predicted words | Probability |
|-----------------|------------------------------|----------|-----------------|-------------|
| <i>for</i> | 0.049593348 | 31 | <i>i</i> | 0.1376049 |
| <i>rst</i> | 0.02372846 | 38 | <i>rmd</i> | 0.6147145 |
| } | 0.0470908 | 39 | <i>return</i> | 0.95013565 |

```

1 #include <stdio.h>
2
3 int    main(void)
4 {
5     int    x, y;
6     int    a, b;
7
8     int    rmd;
9     int    i;
10    int    rst;
11
12    scanf("%d %d",&a, &b);
13    if( a >= b )
14    {
15        x = a;
16        y = b;
17    }else
18    {
19        x = b;
20        y = a;
21    }
22
23    rmd = y % x;
24    if( rmd == 0 )
25    {
26        printf("%d\n",y);
27    }
28    else
29    {
30
31        for(i=1; i<=rmd; i++)
32        {
33            if((( y % i) == 0) && (( rmd % i) == 0))
34            {
35                rst = i;
36            }
37        }
38        printf("%d\n",rst);
39    }
40    return 0;
41 }

```

Figure 3.15: Erroneous source code evaluated by the LSTM-AM

Table 3.10: List of detected errors and predicted words in Figure 3.15 by the LSTM-AM model

| Erroneous words | Erroneous word's probability | Location | Predicted words | Probability |
|-----------------|------------------------------|----------|-----------------|-------------|
| <i>a</i> | 0.034574546 | 12 | <i>x</i> | 0.9269715 |
| = | 0.012642788 | 13 | <i>b</i> | 0.9468921 |
| <i>rmd</i> | 0.03553478 | 23 | } | 0.6362259 |
| <i>for</i> | 0.037460152 | 31 | <i>while</i> | 0.5292723 |
| <i>rst</i> | 0.025345348 | 35 | <i>i</i> | 0.8597483 |

proposed model and other baseline models are shown in Table 3.11 and Table 3.12, respectively. The results show that the proposed model achieves better accuracy in both types of error detection.

Table 3.11: Assessment results of syntax errors (*CE*) detection for erroneous source code

| Problem | RNN | LSTM | LSTM-AM |
|-------------------------|-------------|-------------|-------------|
| Insertion Sort | 83 | 88 | 98 |
| Greatest Common Divisor | 81 | 90 | 95 |
| Prime Numbers | 74 | 85 | 93 |
| Bubble Sort | 80 | 80 | 96 |
| Selection Sort | 69 | 78 | 92 |
| Average | 77.4 | 84.2 | 94.8 |

3.7.4 Classification of Source Codes

In this section, we present the source code classification performance of the proposed LSTM-AM and existing state-of-the-art models. We considered various kinds of error in source code,

Table 3.12: Assessment results of logic errors (*WA*, *TLE*, *MLE*, *PrE*, and *RTE*) detection for erroneous source code

| Problem | RNN | LSTM | LSTM-AM |
|-------------------------|-------------|-------------|----------------|
| Insertion Sort | 60 | 75 | 95 |
| Greatest Common Divisor | 57 | 81 | 96 |
| Prime Numbers | 63 | 77 | 90 |
| Bubble Sort | 65 | 80 | 91 |
| Selection Sort | 56 | 78 | 89 |
| Average | 60.2 | 78.2 | 92.2 |

including *CE*, *RTE*, *TLE*, *MLE*, *WA* and *PrE*. We evaluated the source code classification performance of the proposed model and state-of-the-art models by considering error occurrences in the source code. The proposed model calculated the error probability of each error candidate word to classify the source code. Each variable, keyword, operator, operand, class, function, etc. in the source code are considered as a normal word. The model generates the error probability for each error candidate word followed by the softmax layer. The proposed model detects errors in source codes where all the detected errors are not True errors (TE). So, only TEs are considered for the classification process. An error is called a TE when its predicted probability is more than 0.90. We aligned the term true positive $tp_{e \rightarrow e}$ with the proposed model when the model detects TE in erroneous source codes. Again, in case of the term false positive $fp_{c \rightarrow e}$, at least a single TE is detected within correct source codes. Finally, the term false negative $fn_{e \rightarrow c}$, not a single TE is detected in erroneous source code that means classify the erroneous source code as clean code. As mentioned above, all the models are trained by using correct source codes and tested on 500 randomly chosen source codes from each problem sets (IS, GCD and PN).

To evaluate the classification performance, we compared our model with some baseline methods such as i) Random Forest (RF) [135] method, ii) Random Forest (RF) method trained with secret attributes by Restricted Boltzmann Machine (RBM) [136] and iii) Random Forest (RF) method learned with secret attributes by Deep Belief Network (DBN) [137]. The classification results are listed in Table 3.13, Table 3.14 and Table 3.15 for the IS, GCD and PN source codes respectively.

3.8 Discussion

In this section, we discuss the results presented in Section 3.7. During the evaluation process, a few models obtained high cross-entropy, while the standard LSTM achieved very low

Table 3.13: Classification performance comparison using Insertion Sort (IS) source codes

| Models | Precision (P) | Recall (R) | F-measure |
|----------|---------------|------------|-----------|
| LSTM-AM | 0.99 | 0.97 | 0.97 |
| LSTM | 0.90 | 0.88 | 0.88 |
| RNN | 0.82 | 0.79 | 0.80 |
| RF | 0.62 | 0.55 | 0.58 |
| RF + RBM | 0.66 | 0.65 | 0.65 |
| RF + DBN | 0.71 | 0.66 | 0.68 |

Table 3.14: Classification performance comparison using Greatest Common Divisor (GCD) source codes

| Models | Precision (P) | Recall (R) | F-measure |
|----------|---------------|------------|-----------|
| LSTM-AM | 0.98 | 0.95 | 0.96 |
| LSTM | 0.87 | 0.89 | 0.87 |
| RNN | 0.80 | 0.81 | 0.80 |
| RF | 0.64 | 0.59 | 0.61 |
| RF + RBM | 0.70 | 0.63 | 0.66 |
| RF + DBN | 0.75 | 0.80 | 0.77 |

cross-entropy, thus we discarded the results. Therefore, we validated both the standard LSTM and LSTM-AM models using several randomly chosen erroneous source codes. Figure 3.12 and Table 3.7 are presented detail of error detection and prediction by standard LSTM. The standard LSTM detected errors in lines 2, 6, 15, and 16, and provided the corresponding candidate words “*a*”, “*if*”, “*double quotation*”, and “*c*”, respectively. The predicted correct words are “*)*”, “*else*”, “*blank space*”, and “*b*”. Although these results show the standard LSTM had detected the most probable erroneous words and locations, not all of the candidate errors are TEs. In line 2, the model detects “*a*” as an error candidate by guessing that “*gcd*” is a function without arguments. Then, as a consequence, it predicts a close parenthesis “*)*” as the correct word. Similarly, in line 6, the model detected “*if*” as a candidate error word and predicted “*else*” as a corresponding correction. In this case, the model calculated that the word “*if*” started at line 3 and ended at line 5, and that the word after line 5 should be “*else*”. As a result, the standard

Table 3.15: Classification performance comparison using Prime numbers (PN) source codes

| Models | Precision (P) | Recall (R) | F-measure |
|----------|---------------|------------|-----------|
| LSTM-AM | 0.95 | 0.94 | 0.94 |
| LSTM | 0.88 | 0.86 | 0.86 |
| RNN | 0.76 | 0.79 | 0.77 |
| RF | 0.59 | 0.60 | 0.59 |
| RF + RBM | 0.63 | 0.62 | 0.62 |
| RF + DBN | 0.65 | 0.66 | 0.65 |

LSTM predicted the word “*else*” in line 6 instead of the word “*if*”. However, both the error predictions in lines 2 and 6 were incorrect, even though they were hypothetically reasonable. It should be noted that the error candidate word “*c*” in line 16 is a TE and the predicted word “*b*” is correct. The evaluation results using the standard LSTM for the erroneous source code in Figure 3.12 are presented in Table 3.16.

Table 3.16: The evaluation results based on Figure 3.12 using standard LSTM

| Evaluation Indices | Results (%) | Descriptions |
|---------------------------|--------------------|------------------------|
| EDA | 25 | $TE=1, TDE=4$ |
| CWA | 25 | $TCW=1, TPW=4$ |
| MA | 25 | $EDA =25 \%, CWA=25\%$ |

In Figure 3.13, the LSTM-AM model detected a total of three errors in lines 2, 15, and 16, with the predicted corresponding correct words are “*)*”, “*blank space*”, and “*b*” respectively, as shown in Table 3.8. The evaluation results using the LSTM-AM for the erroneous source code in Figure 3.13 are presented in Table 3.17.

Table 3.17: The evaluation results based on Figure 3.13 using LSTM-AM

| Evaluation Indices | Results (%) | Descriptions |
|---------------------------|--------------------|------------------------------|
| EDA | 33.33 | $TE=1, TDE=3$ |
| CWA | 33.33 | $TCW=1, TPW=3$ |
| MA | 33.33 | $EDA =33.33 \%, CWA=33.33\%$ |

To further evaluate the performance of the proposed model, we then took another erroneous source code and verified it using both the standard LSTM and LSTM-AM networks, as shown in Figures 3.14 and 3.15, respectively. The erroneous source code contains a logical error in line 23. In this source code, two inputs were taken from the keyboard as “*a*” and “*b*” variables. The higher value was assigned to variable “*x*” and the lower value was assigned to variable “*y*”. Initially, variable “*x*” was thought to be a dividend and variable “*y*” was designated as a divisor. However, line 23 was checked to find the initial greatest common divisor used by the modular arithmetic operator where the small valued variable “*y*” was considered to be a dividend and higher valued variable “*x*” was considered to be a divisor. By following the code sequence, the correct logic would be $x \% y$. Based on that aspect, the LSTM-AM network identified the logical error correctly by considering the previous source code sequence, whereas the standard LSTM could not detect the logical error in line 23. The evaluation results for erroneous source codes in Figures 3.14 and 3.15 are listed in Table 3.18, where it can be seen that the LSTM-AM network performance was even better in the case of the long source codes and complex codes

with logical or other errors.

Table 3.18: Evaluation results by the standard LSTM and LSTM-AM

| Models | EDA | CWA | MA |
|---------|-----|-----|-----|
| LSTM | 66% | 30% | 48% |
| LSTM-AM | 90% | 72% | 81% |

In addition to the above-mentioned source code evaluations and examples, we evaluated about 300 randomly chosen erroneous source codes using the LSTM and LSTM-AM models and found that their average accuracy values were approximately 31% and 62% respectively. Those detailed statistics are shown in Table 3.19.

Table 3.19: Overview of the average evaluation statistical results using various erroneous source codes

| Name | Codes | LSTM | | | LSTM-AM | | |
|-----------|-------|-------------|-------|-------|--------------|-------|-------|
| | | EDA | CWA | MA | EDA | CWA | MA |
| GCD | 100 | 34.27 | 32.13 | 33.20 | 65.47 | 59.04 | 62.26 |
| PN | 100 | 28.00 | 31.00 | 29.50 | 64.60 | 57.30 | 60.95 |
| IS | 100 | 31.40 | 29.80 | 30.60 | 63.60 | 61.00 | 62.30 |
| MA | | 31.1 | | | 61.84 | | |

Additionally, some syntax and logical errors in source codes cannot be identified by traditional compilers. In such cases, the proposed LSTM-AM-based language model can provide meaningful responses to learners and professionals that can be used for the source code debugging and refactoring process. This can be expected to save time when working to detect errors from thousands of lines of source code, as well as to limit the area that must be searched to find the errors. Furthermore, use of this intelligent support model can assist learners and professionals in more easily find the logical and other critical errors in their source codes. Moreover, the classification accuracy of the proposed model is much better than the other state-of-the-art models. The average precision, recall and f-measure scores of the LSTM-AM model are 97%, 96%, and 96% respectively that outperformed other state-of-the-art models.

3.9 Summary

In this Chapter, we proposed a neural network-based source code assessment model to assist students, novice and professional programmers. The proposed model is expected to be effective in providing end-to-end solutions for the programming learners and professionals in the SE fields. The experimental results obtained in this Chapter show that the accuracy of error

detection and prediction using the proposed LSTM-AM model is approximately 62%, whereas standard LSTM model accuracy is approximately 31%. In addition, the proposed model provides the location numbers for the predicted errors, which effectively limits the area that must be searched to find errors. Thereby reducing the time required to fix large source code sequences. Furthermore, the proposed model predicted correction words for each error location and detects logical errors (*WA, TLE, PrE, MLE, RTE*) that cannot be recognized by conventional compilers. Also, the LSTM-AM model shows great success in source code binary classification (correct or incorrect) than other state-of-the-art models. As a result, it is particularly suitable for application to long source code sequences and can be expected to contribute significantly to source code debugging and refactoring process. Despite the above-mentioned advantages, the proposed model also has some limitations. For example, error detections and predictions are not always perfect, and the model sometimes cannot understand the semantic meaning of the source code because of the incorrect detections and predictions that have been produced.

Chapter 4

Code Assessment and Classification

Using Bidirectional LSTM

4.1 Introduction

Programming is among the most critical skills in the field of computing and software engineering. As a consequence, programming education has received an ever-increasing level of attention. Many educational institutions (universities, colleges, and professional schools) offer extensive programming education options to enhance the programming skills of their students. Indeed, programming has become recognized as a core literacy [53]. Programming skills are developed primarily through repetitive practice, and many universities [17, 18, 21, 22] have created their own programming learning platforms to facilitate such practice for their students. These platforms are often used for programming competitions and serve as automated assessment tools for programming courses [60]. Novice programmers tend to have difficulty developing and debugging source code due to the presence of various types of errors (i.e., *TLE*, *MLE*, *WA*, *PrE*, *OLE*) and the insufficiency of conventional compilers to detect these errors [138, 139].

Consider a simple program that takes an integer input n from the keyboard and generates an output sum s that repetitively adds integers from 1 through n . The solution code is written in C programming language to implement the procedure and is compiled by a conventional compiler. After compiling, the user inputs $n = 6$ and the program correctly produces sum $s = 21$ as the output; similarly, input $n = 7$ produces an output sum $s = 28$.

```
#include <stdio.h>
int main(){
int j, l, totalsum=0;
printf("Give a number: ");
scanf("%d", &l);
for (j=1; j<=l; j++){
totalsum= totalsum +j;
}
printf("Total sum of 1 to %d is: %d", l, totalsum);
return (0);
}
```

Now consider the code below in which a novice programmer has made a mistake (a small logic error) but the compiler executes the program normally and generates output, which, in this case, is incorrect. Specifically, the program has taken input $n = 6$ and produced output sum $s = 15$; similarly, input $n = 7$ produces output sum $s = 21$.

```
#include <stdio.h>
int main(){
int j, l, totalsum=0;
printf("Give a number: ");
scanf("%d", &l);
for (j=1; j<l; j++){
totalsum= totalsum +j;
}
printf("Total sum of 1 to %d is: %d", l, totalsum);
return (0);
}
```

No compiler has the ability to detect the coding error here. In more complex examples, such logic errors can be difficult to resolve. Environment-dependent logic errors, such as forgetting to include “= 0” for *totalsum* in the above example, are not uncommon, and even experienced programmers can make errors in source code [107]. It is widely accepted that many known and

unknown errors go unrecognized by conventional compilers, which means that programmers often spend valuable time identifying and fixing these errors. To help programmers, especially novice programmers, deal with such source code errors quickly and efficiently, research seeking to shed light on the issue is being actively conducted in programming education [39, 106].

A variety of methods have been proposed, such as source code classification [140, 141], code clone detection [142, 143], defect prediction [144], program repair [36, 113], and code completion [64, 145] to address source code related problems. Recently, NLP has been used in a number of domains, including speech recognition, language processing, and machine translation. The most commonly used language models, including bi-gram, GloVe [108], tri-gram, and skip-gram, are examples of NLP-based language models. However, while these models may be useful for relatively short, simple codes, they are considerably less effective for long, complex codes. Today, deep neural network models are being used for language modeling due to their ability to consider long input sequences, and DNN-based language models are being developed for source code bug detection, logic error detection, and code completion [38, 63, 64, 114, 146]. RNNs have been used but are less effective due to gradient vanishing or exploding [147]. LSTM has overcome gradient vanishing or exploding problem.

LSTM neural networks consider previous input sequences for prediction or output. However, the functions, classes, methods, and variables of a source code may depend on both previous and subsequent code sections or lines. In such cases, LSTM may not produce optimal results. To fill this gap, we propose a BiLSTM language model to evaluate and repair source codes. A BiLSTM neural network can combine both past and future code sequences to produce output [148, 149]. In constructing and applying BiLSTM model, we first perform a series of pre-processing tasks on the source code, then encode the code with a sequence of IDs. Next, we train the BiLSTM neural network using the encoded IDs. Finally, the trained BiLSTM model is used for source code evaluation and repair. The proposed model can be used for different systems (i.e., online judge type, or program/software development where specifications and input/output are well defined) where problems (questions), submission forms (editors), and automatic assessments are involved. We can use the proposed model for an intelligent coding environment (ICE) [150] via API (Application Programming Interface). ICE is one of the examples of many services. On the other hand, there are many powerful and intelligent IDEs (i.e., grammatical support) available, but the proposed BiLSTM model (which can be applied for online judge type systems) can provide much smarter feedback by identifying errors (i.e.,

WA, *TLE*, *RTE*, *MLE*, etc.) than conventional IDEs.

Over the past few years, programmers have improved their programming skills and can write code in many different programming languages to solve problems. As a result, a huge amount of source code written in various languages is regularly pushed to the Cloud repositories [151]. However, due to the large size of source code repositories, manual classification of source code is quite expensive and time-consuming [151]. In a study [152], a CNN was used to classify the source codes based on the algorithms used in codes. In [151], experiments conducted using source codes were written in the C++ programming language. Rahman et al. [102] proposed a BiLSTM neural network to classify the source codes written in C programming language. Similarly, an attention-based LSTM language model has been applied to classify the programming codes [63, 64].

In addition to programming, multilingual applications are spreading in a variety of fields, including social media, audio, video, text, business, and medicine. ML algorithms are increasingly being applied to classify multilingual data resources, such as sentiment analysis [153] and texts related to medicine [154]. In [155], the single-layer BiLSTM model used for sentiment classification, and three different movie review databases consisting of public opinions about movies were used in their experiment. The experimental results show that the single-layer BiLSTM model was better than the other models compared. Furthermore, programmers have been upgrading themselves in terms of skills, applications, and adaptation to new programming environments and languages. As a result, the nature of source code archives has become more heterogeneous and challenging for researchers. Nowadays, novice programmers spend a lot of time identifying problems in source code that are solved using MPLs (e.g., C, C++, Java, Python, Python 3, etc.). This is a hindrance to learn programming, not only for novice programmers but also for experienced programmers.

In this Chapter, we also present a stacked BiLSTM model for classifying source codes built/written in MPLs. Since methods, classes, variables, tokens, and keywords have both short-term and long-term dependencies, the stacked BiLSTM layers make the model deeper and provide a better understanding of the context of the codes. Using a large number of real-world source codes, we trained LSTM, BiLSTM, and stacked BiLSTM models. We tweak various hyperparameters and observe the performance of the models. The main contributions of this Chapter are summarized below:

- The proposed BiLSTM language model for code assessment can effectively detect errors

including logical errors (*TLE*, *MLE*, *OLE*, *WA*, etc.) and provide corrections for erroneous codes.

- The BiLSTM model can be helpful to students, programmers (especially novice programmers), and professionals who often struggle to resolve code errors.
- The model can be used for different real-world programming learning and software engineering platforms and services.
- The stacked BiLSTM model classifies source codes based on the algorithms. The model can help students and programmers to identify the algorithms used in the source code. Programmers can understand the code better if they know the written algorithm in the code.
- The stacked BiLSTM model can be deployed in the field of software engineering to recognize code in the large code archives.

The remainder of this Chapter is organized as follows. In particular, the research background and related work are described in Section 4.2. The architecture and mathematical background of the proposed BiLSTM model are presented in Section 4.3. In Section 4.3.4, we present the experimental results of code assessments and classification using BiLSTM. Section 4.4 presents stacked BiLSTM model for multi-class classification. In Section 4.4.6, we present the experimental results of multi-class classification using stacked BiLSTM. Section 4.5 concludes the Chapter.

4.2 Background and Related Works

The wide range of application domains and the functionality of DNNs make them powerful and appealing. Recently, ML techniques have been used to solve complex programming-related problems. Accordingly, researchers have begun to focus on the development and application of DNN-based models in programming education and software engineering. In [138], logic errors (LEs) are a type of error that persists after compilation, whereas typical compilers can only detect syntax and semantic errors in codes. They proposed a practical approach to identify and discover logic errors in codes for object-oriented-based environments (i.e., C#.Net Framework). Their proposed Object Behavior Environment (OBEnvironment) can help programmers to avoid

LEs based on predefined behaviors by using Alsing, Xceed, and Mind Fusion Components. That approach is not similar to the proposed BiLSTM model, as their model is developed for the C# programming language in the .Net framework. Al-Ashwal et al. [139] introduced a CASE (computer-aided software engineering) tool to identify LEs in Java programs using both dynamic and static methods. Programmers faced difficulties in identifying LEs in the codes during testing; sometimes it is necessary to manually check the whole code, which also takes a large amount of time, effort, and cost. They used PMD and Junit tools to identify LEs on the basis of a list of some common LEs related to Java. Their study is only suitable for identifying LEs in Java programs, but not for other programming languages.

In [156], an automated LE detection technique is proposed for functional programming assignments. A large amount of manual and hand-made efforts are required to identify LEs in test cases. Their proposed technique uses a reference solution for each assignment (written in OCaml programming language) of students to create a counter-example that contains all the semantic differences between the two programs. That method identified 88 more LEs that were not identified by the mature test cases. Moreover, the technique can be effective for automatic code repair. The disadvantage of this method is that a reference program is needed to identify LEs for each incorrect code. In [157], the authors studied a large number of research papers on programming languages and natural languages that were implemented using probabilistic models. They also described how researchers adapted these models to various application domains.

Raychev et al. [145] addressed code completion by adopting an n -gram language model and RNN. Their model was quick and effective in code completion tasks. Allamanis et al. [158] proposed a neural stochastic language model to suggest methods and class names in source codes. The model analyzed the meaning of code tokens before making its suggestions and produced notable success in performing method, class, and variable naming tasks. S. Wang et al. [159] proposed a model for predicting defective regions in source codes on the basis of the code's semantics. The DBN was trained to learn the semantic features of the code using token vectors derived from the code's AST, as every source code contains method, class, and variable names that provide important information. On the basis of the semantic meaning, Pradel et al. [160] introduced a name-based bug detection model for codes.

Song et al. [161] proposed a BiLSTM model to detect malicious JavaScript. In order to obtain semantic information from the code, the authors first constructed a program dependency graph (PDG) for generating *semantic slices*. The PDG stores semantic information that is later

used to create vectors. The approach was shown to have 97.71% accuracy, with an F1-score of 98.29%. In articles [38, 114], the authors proposed an LSTM-based model for source code bug detection, code completion, and classification. Both methods were used to develop the programming skills of novice programmers. Experimental results, obtained by tuning the various hyper parameters and settings of the network, showed that both models achieved better results for bug detection and code completion in comparison with other related models. In [63, 64], the authors proposed error detection, LE detection, and the classification of source codes on the basis of an LSTM model. Both approaches used an attention mechanism that enhanced model scalability. On the basis of various performance scales, both models achieved significant success compared to more sophisticated models. As noted earlier, however, an LSTM-based model considers only previous input sequences for prediction but is unable to consider future sequences. The proposed BiLSTM model has the ability to consider both past and future sequences for output prediction. In brief, there have been a number of novel and effective neural network and probabilistic models proposed by researchers to solve problems related to source codes. The proposed BiLSTM model is unlike from other models in that it considers both the previous and subsequent context of codes to detect errors and offer suggestions that enable programmers and professionals to make the needed repairs efficiently.

Furthermore, various research methodologies have been introduced and proposed for classification in different domains. In addition to traditional classification methods, ML-based models have recently been effectively employed for classification tasks. In [162], three well-known classifiers such as KNN, relevance feedback, and Bayesian independence were combined to classify medical texts. Different combinations of these classifiers yielded better results than the single classifier. On the other hand, the uni-gram language model was used to classify text [163]. Recently, many supervised and unsupervised classification models such as ANNs, support vector machines (SVMs), and RF trees have been applied to various classification tasks. Meanwhile, in [164], a large number of source codes were classified based on SVM, the model correctly classified the source code into topics and languages .

Despite the use of traditional classification methods, DNN-based models have recently been used moderately in code-related classification tasks. In the last few years, techniques based on ML have shown promising results for text processing and classification, especially DNN techniques (e.g., RNN, CNN, LSTM, BiLSTM). A single-layered BiLSTM model used for binary (0 or 1) sentiment classification, BiLSTM model was reported to be computationally

efficient than other models [155]. In [42], an LSTM-based model was proposed for automatic classification of source code archives based on programming languages. The RNN model gave more significant results than the Naive Bayes classifier. In [63, 64, 102, 152], CNN, LSTM, and BiLSTM based models were used to classify source code, and these models yielded significant results compared to supervised classification methods, but all of them used source codes written on a specific programming language (such as C or C++). Since it is difficult to explain which model is better, but the proposed multi-class source code classification model using BiLSTM is different from the existing models.

4.3 Proposed BiLSTM Model for Code Assessment and Classification

In this section, we describe the proposed BiLSTM model. First, Figure 4.1 shows the workflow of the model, proceeding from source code collection to code evaluation by the trained BiLSTM model.

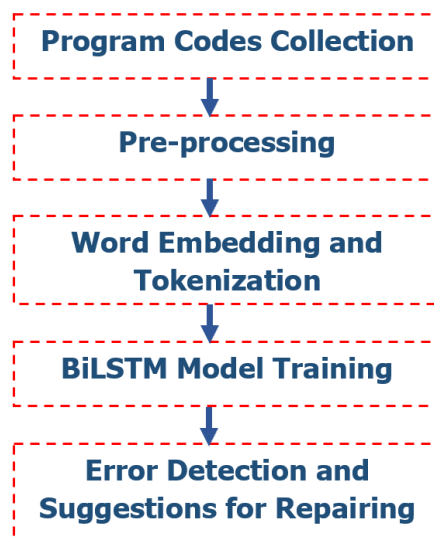


Figure 4.1: Workflow of the proposed BiLSTM model

4.3.1 BiLSTM Model Architecture

Let $I = \{i_1, i_2, i_3, \dots, \dots, i_t\}$ be the set of encoded IDs of source codes. An RNN then executes for each encoded ID i_t for $t = 1$ to n . The output vector of RNN y_t can be expressed by

the following equations:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \quad (4.1)$$

$$y_t = W_{hy}h_t + b_y \quad (4.2)$$

where h_t is the hidden state output, W is a weight matrix (W_{xh} is a weight connecting input (x) to hidden layer (h)), b is a bias vector, and \tanh is an activation function of the hidden layer. Equation (4.1) is used to calculate the hidden state output, where the hidden state receives the results of the previous state.

However, due to the problem of *gradient vanishing/exploding* [147], not all input sequences are used effectively in an RNN. To avoid the problem and produce a better result, the RNN is extended to LSTM. Conceptually, an LSTM network is similar to an RNN, but the hidden layer updating process is replaced by a special unit called a memory cell. LSTM is implemented by applying the following equations:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \quad (4.3)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad (4.4)$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (4.5)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \quad (4.6)$$

$$h_t = o_t \tanh(c_t) \quad (4.7)$$

where σ is a sigmoid function; c , f , i , and o are the cell state, forget gate, input, and output, respectively; and all b are biases. However, there is still a shortcoming in LSTM insofar as it considers only the previous context of the input but cannot consider any future (i.e., subsequent) context.

To overcome this limitation, we adopted the BiLSTM model [165], which enables us to consider both the past and future context of source codes, as shown in Figure 4.2. Here, there are two distinct hidden layers, called the forward hidden layer and backward hidden layer. The forward hidden layer h_t^f considers the input in ascending order, i.e., $t = 1, 2, 3, \dots, T$. On the other hand, the backward hidden layer h_t^b considers the input in descending order, i.e., $t = T, \dots, 3, 2, 1$.

Finally, h_t^f and h_t^b are concatenated to generate output y_t . The BiLSTM model is implemented with the following equations:

$$h_t^f = \tan h \left(W_{xh}^f x_t + W_{hh}^f h_{t-1}^f + b_h^f \right) \quad (4.8)$$

$$h_t^b = \tan h \left(W_{xh}^b x_t + W_{hh}^b h_{t+1}^b + b_h^b \right) \quad (4.9)$$

$$y_t = W_{hy}^f h_t^f + W_{hy}^b h_t^b + b_y \quad (4.10)$$

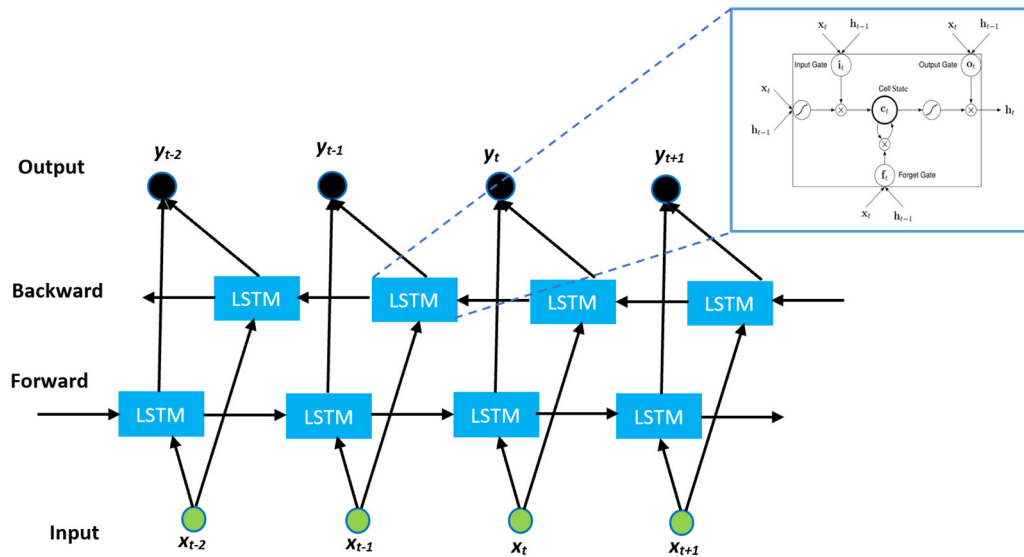


Figure 4.2: Architecture of the BiLSTM neural network

The training and evaluation processes of the proposed model are shown in Figure 4.3. The BiLSTM network is used as the core processing unit for training and code evaluation. The figure shows the typical input and output pattern of the model.

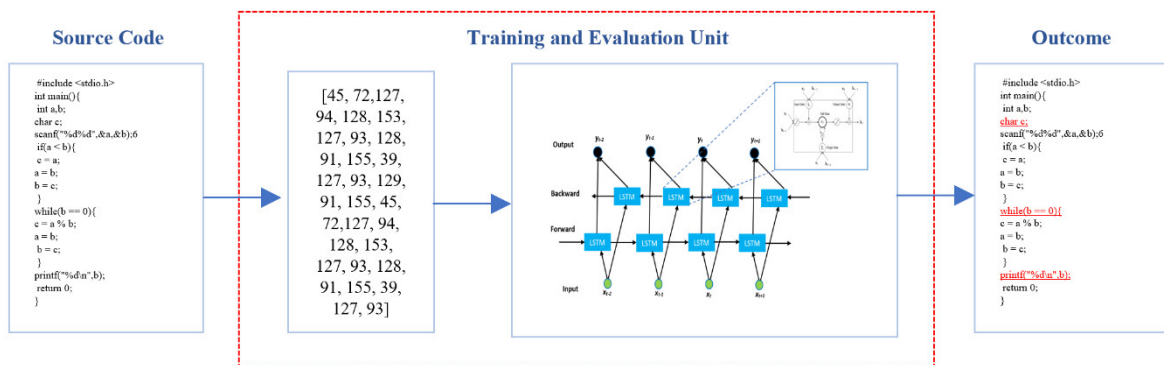


Figure 4.3: Typical input and output prototype of the proposed BiLSTM model

4.3.2 Dataset and Experimental Setup

In this Chapter, all real-world source codes (data) are collected from the AOJ system. The data preprocessing and model training steps mentioned in Section 3.5.1 are performed. All experiments are conducted using the source codes of GCD and IS problems. A total of 2482 codes were included in the experiments: 90% for model training, 5% for model validation, and 5% for testing. The average length or number of lines in the GCD and IS solution codes were 18.9 and 30.91, respectively. Moreover, the average sizes of GCD and IS solution codes were 262.45 and 532.28 bytes, respectively. The difficulty or complexity level of solution codes was moderate. To balance the evaluation of the experimental results, we selected an equal number of correct (50%) and incorrect (50%) source codes from each type of problem (GCD and IS). The nature of the error was heterogeneous in the incorrect source codes. We did not select similar or common error typed source codes for training, validation, and testing. Instead, we randomly selected a variety of faulty source codes. To obtain the best results, we tuned the network configurations using hidden neurons of different sizes (e.g., 100, 200, 300, and 400) for the BiLSTM and other models. Training data were saved as .npz format for each type of hidden neuron (200, 300, 400, etc.). Similarly, for the output (error identification and providing suggestions), the model used the same number of hidden neurons. A value of 0.50 was used for the dropout [133] layers to avoid network overfitting. The Adam optimization algorithm [134] was adopted during model training. Particularly in deep learning, Adam optimizer is effectively used for the purpose of model learning. It balances model parameters and loss functions to efficiently update network weights.

$$S(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^n \exp(z_j)} \quad (4.11)$$

Our proposed model is a seq2seq language model that predicts next words in incorrect codes on the basis of probability. The Softmax activation layer (as defined in Equation (4.11)) is used to transform the output vector to probability where Softmax takes vector $Z = [z_1, z_2, z_3, \dots, z_n]$ and produces a vector $S(z) = [s_1, s_2, s_3, \dots, s_n]$ for probabilities. The Softmax layer generates the probability for each word (token or ID) if the probability is too low (less than 0.1), which is considered as an error candidate and immediately mark the entire line. At the same time, the model generates a possible correct word instead of the error. To predict the correct word (token or ID), the model (BiLSTM) calculates the code sequences (both forward and backward) to find

the best possible word on the basis of the highest probability.

4.3.3 Evaluation Metrics for Code Assessment

In this section, we have defined terms "Error identification accuracy (EIA)", "Suggestion accuracy (SA)", and "Correctness of the model (CoM)" as model evaluation metrics for code assessment. Furthermore, we used precision, recall, and F-score to evaluate the classification results of the BiLSTM model. For this purpose, we adopted equations 3.31, 3.32, and 3.33 for precision, recall, and F-score, respectively.

Definition 5 (*Error identification accuracy*) *The model identifies erroneous candidates (words) in the solution codes; the number of correct or actual error candidates (words) out of the total identified error candidates (words) is called error identification accuracy (EIA). The "Number of correctly detected errors" are the errors that actually exist in the code and the "Total number of detected errors" are the errors (may exist or not in the code) detected by the model.*

$$EIA = \frac{\text{Number of correctly detected errors in code}}{\text{Total number of detected errors in code}} \times 100\% \quad (4.12)$$

Example 4 *If a model m identifies a total of 11 error candidates (words) in solution code s_1 , and only 5 of the identified candidates (words) are actually present in the code, the EIA of model m for s_1 is approximately 45.45%.*

Definition 6 (*Suggestion accuracy*) *The model generates suggestions for each identified error candidate; the number of correct or actual code repair suggestions out of the total suggestions for error candidates is called suggestion accuracy (SA).*

$$SA = \frac{\text{actual suggestion for error candidates}}{\text{total suggestions}} \times 100\% \quad (4.13)$$

Example 5 *If model m generates a total of 20 suggestions in solution code s_2 , and only 13 of the total suggestions are correct or true, the SA of model m for s_2 is 65%.*

Definition 7 (*Correctness of the model*) *Correctness of the model is calculated as the average of EIA and SA values.*

$$\text{Correctness of Model (CoM)} = \frac{EIA + SA}{2} \quad (4.14)$$

Example 6 If model m has an EIA value of 45.45% and an SA value of 65%, the correctness of model m will be approximately 55.23%.

4.3.4 Experimental Results

In this section, we present the hidden units and epochs selection, code assessment and classification results based on the BiLSTM model.

4.3.4.1 Selection of the Number of Epochs and Hidden Units

An epoch is a complete cycle with the full training dataset. Using the optimal number of epochs can improve the performance of the model as well as save time for model training. Our training dataset consists of the solution codes of the two problems GCD and IS. We trained the model separately for each of the two types using different hidden units.

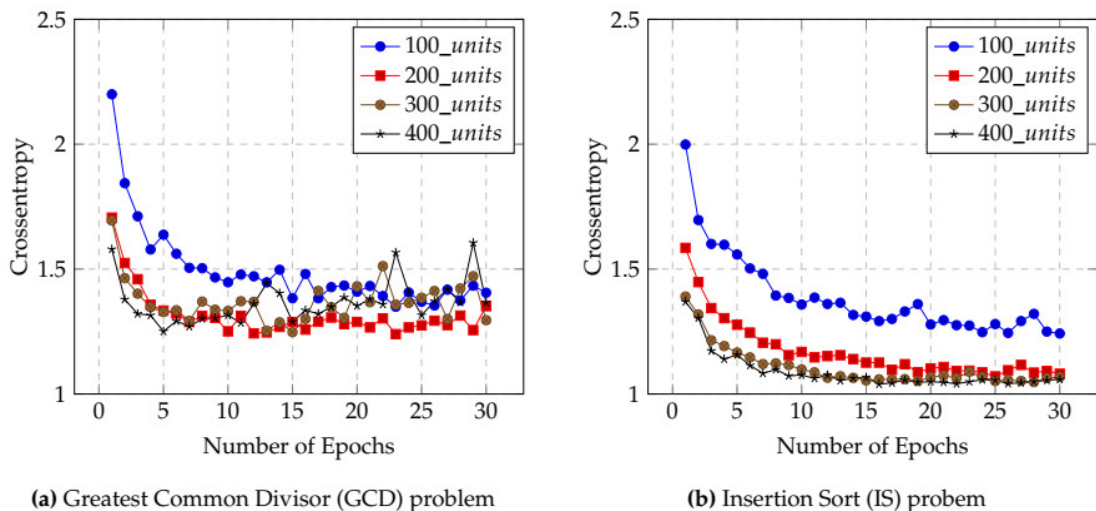


Figure 4.4: Effect of cross-entropy on selecting epochs and hidden units for BiLSTM. (a) GCD problem set, (b) IS problem set.

Figure 4.4 shows the results of the cross-entropy calculations used to select the optimal number of epochs and hidden units (neurons) for the BiLSTM model. Figure 4.4(a) gives the results for the GCD case. First, we identified the appropriate number of hidden units for model training. In this case, 200 hidden units produced the lowest cross-entropy. Moreover, cross-entropy was lowest when the number of epochs was between 20 and 25. The indication is that, for the GCD case, the model produced its best performance when the number of hidden units was 200 and the number of epochs was between 20 and 25, and that using these values saves model training time. Similarly, Figure 4.4(b) shows that, in the IS case, cross-entropy for the

BiLSTM model was lowest when 400 hidden units and between 25 and 30 epochs are used.

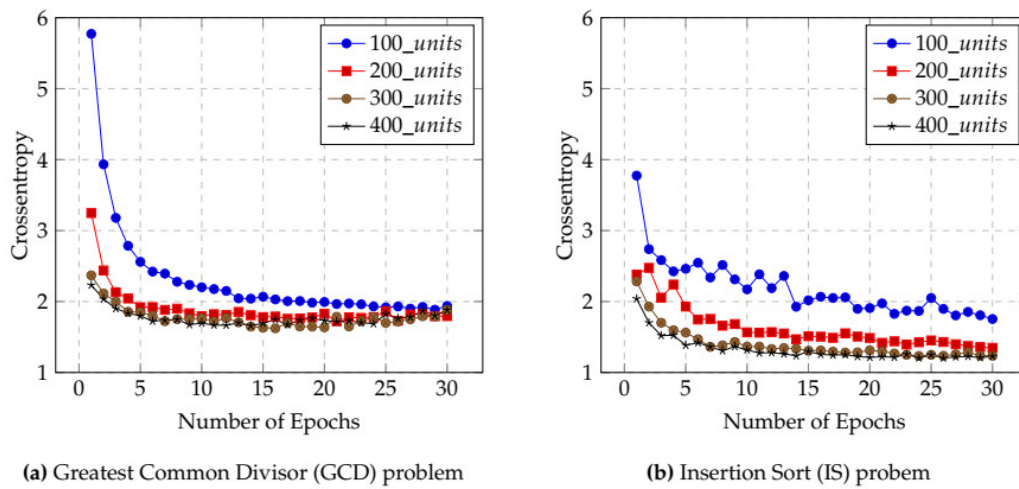


Figure 4.5: Effect of cross-entropy on selecting epochs and hidden units for LSTM. (a) GCD problem set, (b) IS problem set.

Figure 4.5 shows the cross-entropy results for the LSTM model. Figure 4.5(a) indicates that, in the GCD case, the cross-entropy of the LSTM model reached its lowest level when 300 hidden units and between 22 and 25 epochs were used. Figure 4.5(b) provides the results of when IS codes were used for model training. Here, the LSTM model had minimum cross-entropy when 400 hidden units and between 25 and 30 epochs were used.

Comparative statistics for the lowest cross-entropy of the LSTM and BiLSTM models are given in Table 4.1. For the LSTM model, 300 and 400 hidden units produced the minimum cross-entropy for the GCD and IS solution codes, respectively. On the other hand, for the BiLSTM model, 200 and 400 hidden units, respectively, produced the lowest cross-entropy for the GCD and IS solution codes. Therefore, we chose these numbers of hidden units (i.e., those producing the minimum cross-entropy) for model training and code assessment.

Table 4.1: Comparative lowest crossentropy of LSTM and BiLSTM models for different hidden units

| SL | Hidden Units (neurons) | LSTM | | BiLSTM | |
|----|------------------------|-------------|-------------|-------------|-------------|
| | | GCD | IS | GCD | IS |
| 1 | 100 | 1.89 | 1.75 | 1.35 | 1.24 |
| 2 | 200 | 1.72 | 1.35 | 1.24 | 1.07 |
| 3 | 300 | 1.62 | 1.24 | 1.25 | 1.05 |
| 4 | 400 | 1.66 | 1.20 | 1.25 | 1.04 |

4.3.4.2 Erroneous Source Code Assessment

We evaluated erroneous source codes using the LSTM and BiLSTM models and compared the performance of the two models. In Figure 4.6(a), an incorrect GCD solution code was evaluated by the LSTM model. Errors were identified in lines 13, 15, 16, and 18 of the code. According to the context of the code, an error occurred in line 14, which was supposed to be $l = m \% n$; however, the LSTM model was unable to accurately detect the error. Figure 4.6(b) shows an erroneous solution code to an IS problem assessed by LSTM. Most of the errors (logical and syntactic) were identified by the model, but an irrelevant error (actually, no error at all) was identified in line 3.

| | | | |
|---|---|--|--|
| <pre> 1 #include<stdio.h> 2 int main(void){ 3 int a, b, m, n, l; 4 scanf("%d %d", &a, &b); 5 if(a >= b){ 6 m = a; 7 n = b; 8 } 9 else{ 10 m = b; 11 n = a; 12 } 13 while(n == 0){ 14 l = n % m; </pre> | <pre> 15 m = n; 16 n = l 17 } 18 printf("%d\n", m); 19 return 0; </pre> | <pre> 1 #include <stdio.h> 2 #define N 100 3 main(){ 4 int n, a[N], i, j, key; 5 scanf("%d", &n); 6 for(i = 0; i <= n; i++){ 7 scanf("%d", &a[i]); 8 } 9 for(i = 1; i < n; i++){ 10 for(j = 0; j < n-1; j++){ 11 printf("%d ", a[j]); 12 } 13 printf("%d\n", a[n-1]); 14 key = a[i]; </pre> | <pre> 15 j = i - 1; 16 while(j >= 0 && a[j] < key){ 17 a[j+1] = a[j]; 18 j++; 19 a[j+1] = key; 20 } 21 } 22 for(j = 0; j < n-1; j++){ 23 printf("%d ", a[j]); 24 } 25 printf("%d\n", a[n-1]); 26 return 0; </pre> |
| (a) Erroneous solution code of GCD problem | (b) Erroneous solution of IS problem | | |

Figure 4.6: Source codes evaluation by the LSTM model. (a) Erroneous solution code of GCD problem, (b) Erroneous solution of IS problem.

To compare the error assessment efficiency of the two models (LSTM and BiLSTM), we assessed the same erroneous codes by the BiLSTM model. In Figure 4.7(a), a solution to the GCD problem is evaluated by the BiLSTM model. The model detected errors in lines 13, 14, and 16. On the basis of the context of the code, the model considered the output statement in line 18 to identify the errors. As a result, the BiLSTM model correctly identified logical errors in lines 13 and 14 on the basis of the output details in line 18. In Figure 4.7(b), where an incorrect solution to the IS problem was assessed by the BiLSTM model, the BiLSTM model was able to identify all the errors (logical and syntax) in the code. Errors are identified in lines 6, 16, 17, and 18, considering the full context of the erroneous code. In contrast, it is all but impossible to determine logical errors using a conventional compiler or to even consider the later context of the code. Figures 4.6 and 4.7 show that the BiLSTM model properly evaluated the erroneous codes on the basis of later context. On the other hand, the LSTM model was slightly less efficient of considering later context to detect errors.

```

1 #include<stdio.h>
2 int main(void){
3 int a, b, m, n, l;
4 scanf("%d %d %d", &a, &b);
5 if( a >= b){
6 m = a;
7 n = b;
8 }
9 else{
10 m = b;
11 n = a;
12 }
13 while( n == 0 ){
14 l = n % m;
15 m = n;
16 n = l
17 }
18 printf("%d\n", m);
19 return 0;

```

```

1 #include <stdio.h>
2 #define N 100
3 main(){
4 int n, a[N], i, j, key;
5 scanf("%d", &n);
6 for(i = 0; i <= n; i++){
7 scanf("%d", &a[i]);
8 }
9 for(i = 1; i < n; i++){
10 for(j = 0; j < n-1; j++){
11 printf("%d ", a[j]);
12 }
13 printf("%d\n", a[n-1]);
14 key = a[i];
15 j = i - 1;
16 while(j >= 0 && a[j] < key){
17 a[j+1] = a[j];
18 j++;
19 a[j+1] = key;
20 }
21 }
22 for(j = 0; j < n-1; j++){
23 printf("%d ", a[j]);
24 }
25 printf("%d\n", a[n-1]);
26 return 0;

```

(a) Erroneous solution code of GCD problem

(b) Erroneous solution of IS problem

Figure 4.7: Source codes evaluation by the BiLSTM model. (a) Erroneous solution code of GCD problem, (b) Erroneous solution of IS problem.

4.3.4.3 Suggestions for Code Repair

The two models provided suggestions for code repair for each detected error location. Table 4.2 lists the suggestions based on the GCD problem evaluation described above.

Table 4.2: Suggestions for GCD problem evaluated in Figures 4.6(a) and 4.7(a)

| Location in Code | LSTM | | BiLSTM | |
|------------------|---------------|-------------|-------------|-------------|
| | Detected | Suggested | Detected | Suggested |
| 13 | = | ! | = | ! |
| 14 | | | <i>n, m</i> | <i>m, n</i> |
| 15 | <i>m, n</i> | <i>n, m</i> | | |
| 16 | <i>l</i> | <i>l;</i> | <i>l</i> | <i>l;</i> |
| 18 | <i>printf</i> | } | | |

Similarly, Table 4.3 lists the suggestions based on the IS problem evaluations by the LSTM and BiLSTM models.

Table 4.3: Suggestions for IS problem evaluated in Figures 4.6(b) and 4.7(b)

| Location in Code | LSTM | | BiLSTM | |
|------------------|-------------|-----------------|-----------------|-----------------|
| | Detected | Suggested | Detected | Suggested |
| 3 | <i>main</i> | <i>int main</i> | | |
| 6 | = | < | <i><= n</i> | <i>< n</i> |
| 16 | < | > | <i>< key</i> | <i>> key</i> |
| 17 | <i>i</i> | <i>j</i> | <i>a[i]</i> | <i>a[j]</i> |
| 18 | <i>+, +</i> | <i>-, -</i> | <i>j++</i> | <i>j--</i> |

Both the LSTM and BiLSTM models provided relevant suggestions for correcting the detected errors. Such suggestions can be useful to programmers, especially novice programmers, to help them quickly repair erroneous codes. However, not all errors are straightforward; some may rely on previous or subsequent lines in the code. In such cases, the BiLSTM model is more efficient than the LSTM model.

4.3.4.4 Overall Model Performance for Error Detection and Binary Classification

Table 4.4 shows the comparative performances of the BiLSTM, LSTM, and RNN models for the GCD source code dataset. The CoM of the BiLSTM model was approximately 52.4%, which was much higher than the CoM of the other two models. Note that CoM (as per Equation (4.14)) was determined on the basis of the correctness of error detection and the suggestions provided for erroneous code. We calculated standard deviation (σ) on the basis of model performance in terms of error identification and suggestion accuracy. The BiLSTM model achieved the lowest deviation (σ : 4.55) compared to other models, which determines that the performance distribution of the model was consistent. Although the BiLSTM model also had the highest precision rate of 98%, there were very few correct codes classified as incorrect. The 95.5% recall rate indicated that there were relatively few FP. As noted earlier, the F-score is the harmonic mean of the recall and precision ratios and is an important metric to describe model performance. As Table 4.4 shows, the F-score of the BiLSTM model was highest among the three models, at 96.7%, indicating that the proposed model produced superior TP with a low rate of FP.

Table 4.4: Performance of the models based on the GCD dataset

| Model | EIA | CoM | σ | Precision (P) | Recall (R) | F-Score |
|-----------|-----|-------|----------|---------------|------------|---------|
| BiLSTM | 66 | 52.4% | 4.55 | 98% | 95.5% | 96.7% |
| LSTM [64] | 45 | 33.2% | 7.67 | 87% | 89% | 87.0% |
| RNN [64] | 33 | 25% | 7.76 | 80% | 81% | 80.0% |

Table 4.5 shows the comparative results when the models were applied to the IS dataset. In this case, the BiLSTM model outperformed the LSTM and RNN models (CoM: 49.35%; σ : 3.12; precision: 97%; recall: 97%; F-score: 97.0%).

Table 4.5: Performance of the models based on the IS dataset

| Model | EIA | CoM | σ | Precision (P) | Recall (R) | F-Score |
|-----------|-----|--------|----------|---------------|------------|---------|
| BiLSTM | 62 | 49.35% | 3.12 | 97% | 97% | 97.0% |
| LSTM [64] | 41 | 30.6% | 6.09 | 90% | 88% | 88.0% |
| RNN [64] | 29 | 22% | 7.15 | 82% | 79% | 80.0% |

4.4 Proposed Stacked BiLSTM Model for Multilingual Source Code Classification

One variation of the BiLSTM model is stacked BiLSTM, which uses several BiLSTM layers to train the neural network with better contextual information. Stacked BiLSTM uses the same

update formula as BiLSTM that described in Section 4.3.1.

4.4.1 Stacked BiLSTM Model Architecture

In stacked BiLSTM model, we used a two-layer BiLSTM neural network. In this model, the output of hidden state of each BiLSTM layer is given as input to the next BiLSTM layer. The stacked BiLSTM architecture improves the ability of neural networks to learn more contextual information and has been used effectively in various applications [166]. The stacked RNN had achieved state-of-the-art performance for a language modeling tasks [167]. Fig. 4.8 shows the overall architecture using the stacked BiLSTM that employs for the multi-class code classification.

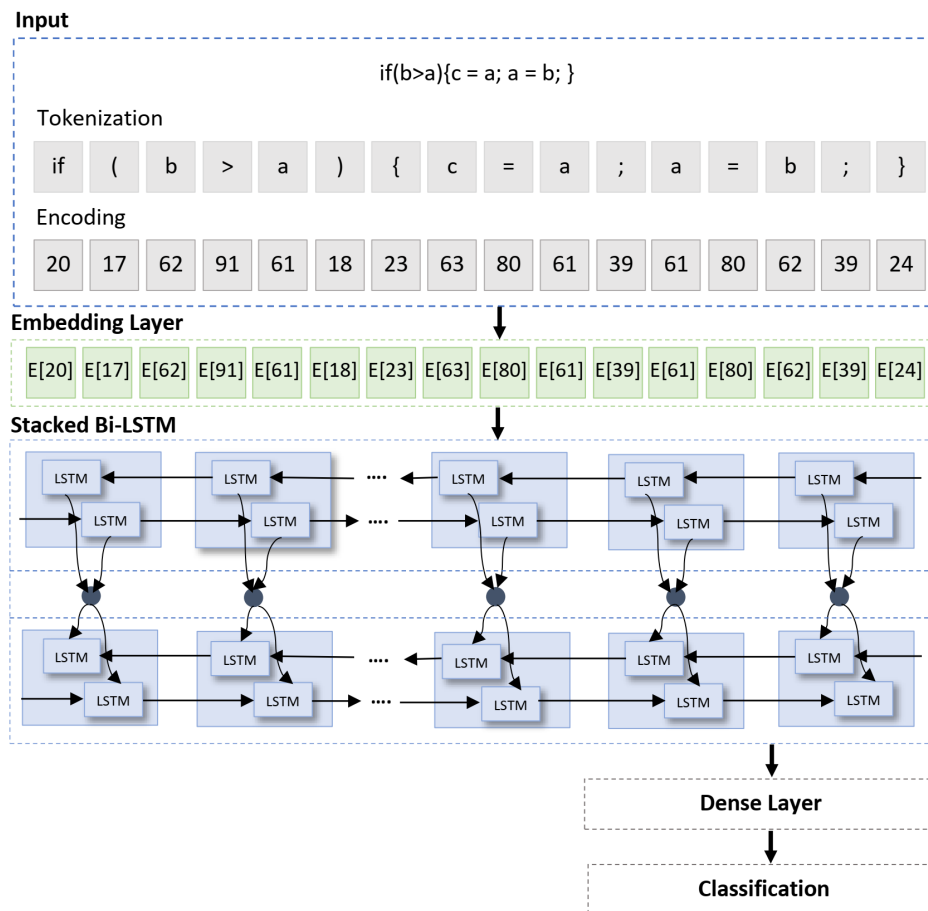


Figure 4.8: Architectural overview of the stacked BiLSTM model

4.4.2 Dataset and Preprocessing

For this multiclass classification model, we collected a real-world dataset from the AOJ system [18, 19]. We collected about 35,000 solution codes written in MPLs for this experiments.

Each solution code belong to a specific problem title (e.g., insertion sort, bubble sort, stack, 15-puzzle, graph, Shell Sort, etc.). A summary of source codes with problem's title/name shown in Table 4.6. According to Table 4.6, the solution codes are distributed among 25 unique classes.

Table 4.6: A summary of solution codes with problem title/class name

| Sl. | Problem title/class name | Number of codes |
|-----|--------------------------------|-----------------|
| 1 | Insertion Sort | 4208 |
| 2 | Greatest Common Divisor | 3054 |
| 3 | Prime Numbers | 2907 |
| 4 | Bubble Sort | 3381 |
| 5 | Selection Sort | 2945 |
| 6 | Stable Sort | 1545 |
| 7 | Fibonacci Number | 1750 |
| 8 | Longest Common Subsequence | 1257 |
| 9 | Matrix Chain Multiplication | 1090 |
| 10 | Graph | 1410 |
| 11 | Breadth First Search | 1006 |
| 12 | Depth First Search | 1300 |
| 13 | Minimum Spanning Tree | 1077 |
| 14 | Single Source Shortest Path I | 957 |
| 15 | Single Source Shortest Path II | 615 |
| 16 | Maximum Profit | 3255 |
| 17 | Shell Sort | 1309 |
| 18 | 8 Puzzle | 198 |
| 19 | Connected Components | 613 |
| 20 | 8 Queens Problem | 309 |
| 21 | 15 Puzzle | 146 |
| 22 | Naive String Search | 263 |
| 23 | String Search | 206 |
| 24 | Pattern Search | 67 |
| 25 | Multiple String Matching | 131 |

The statistics for each programming language in the dataset are shown in Fig. 4.9. In the experiment, source codes written in about 14 different languages are used. The diversity of the dataset is shown by these statistics.

Data preprocessing is one of the most important steps in machine learning. The irrelevant and irrational information can cause the poor results in any machine learning model. In order to get better performance from the machine learning model, data preprocessing is considered as a major step. However, when writing code, irrelevant information such as comments, line breaks, tabs, and spaces are included in the code. Therefore, we have removed all unnecessary information from the source codes. The Natural Language Toolkit (NLTK) [168] is used to

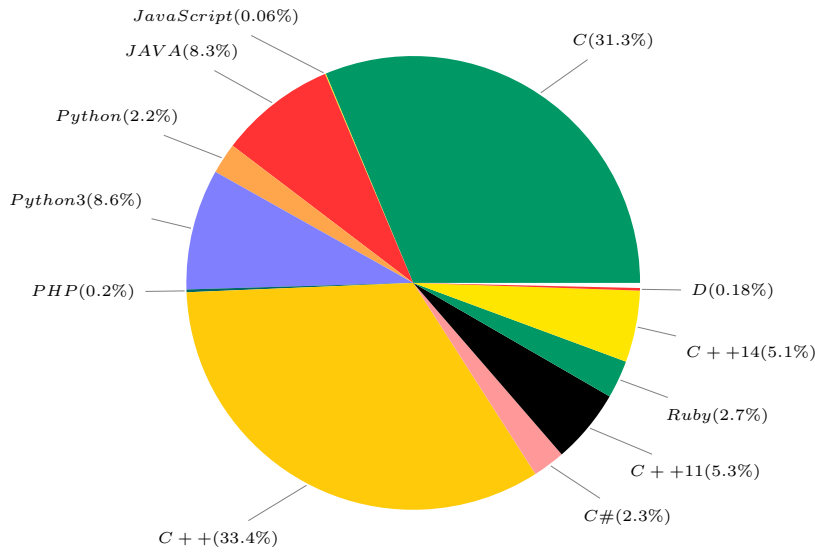


Figure 4.9: Statistical overview of the programming languages based on solution codes

preprocess the data. We considered each keyword, token, variable, function, class, operator, operand, etc. in the source code as a normal word. First, we created a vocabulary list containing the unique words and assigned a natural number to each unique word. This process is called tokenization and encoding, as shown in Fig. 4.10.

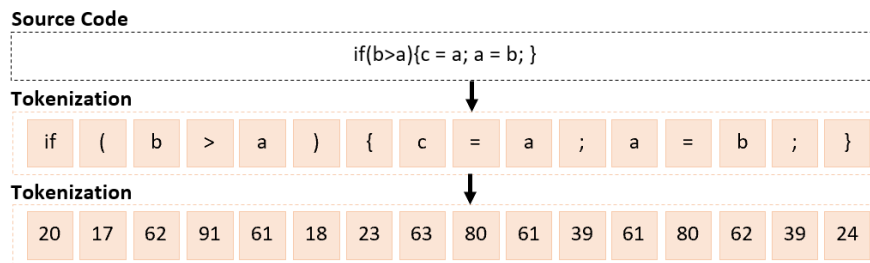


Figure 4.10: Tokenization and encoding

4.4.3 Hyperparameters

Source code is a complex collection of statements such as variables, keywords, tokens, functions, classes, and mathematical operations. These statements are highly dependent on each other. Therefore, neural networks, especially DNN, have the ability to learn the complex context of the source code and solve various tasks associated with the source code. DNN can also learn the complex relationships between the inputs and outputs of source code [169]. However, the training and performance of a neural network is highly dependent on the selection of optimal hyperparameters. We have fine-tuned the hyperparameters for the proposed classification model to obtain better performance. The values of the hyperparameters are shown in Table 4.7.

Table 4.7: List of the hyperparameters and settings used in the proposed model

| Name of the hyperparameter | Values |
|--|---------------------------------|
| Vocabulary size (v) | 10000 |
| Maximum sequence length (m) | 500 |
| Embedding Size (e) | 64 |
| Truncating type (<i>trunc_type</i>) | post |
| Padding type (<i>padding_type</i>) | post |
| Out of vocabulary token (<i>oov_tok</i>) | < OOV > |
| Optimizer | adam |
| Loss function (<i>cross_entropy</i>) | sparse_categorical_crossentropy |
| Activation functions | ReLU and Tanh |
| Epochs (<i>epoch</i>) | 25, 50, 100, and 150 |
| BiLSTM nodes | 128 |
| Training portion of the dataset | 80% |
| Validation portion of the dataset | 15% |
| Testing portion of the dataset | 5% |

4.4.4 Activation Functions

Activation functions are used to add non-linearity to a NN. The activation function expands the learning opportunities of the NN and ensures that the output of the NN is not reproduced from similar combinations of inputs. The activation function plays an important role in improving the overall performance of the network [170]. In the absence of activation functions, each layer of the NN behaves like a single layer perceptron or simple linear regression model; the activation functions of the NN include Linear, Sigmoid, Tanh, Rectified linear unit (ReLU), Leaky ReLU, and so on [170]. In this experiment, we used tanh and ReLU activation functions separately and the performance of both functions is investigated.

Hyperbolic Tangent or Tanh: The Tanh function takes any real number from NN as input and produces an output in the range of -1 to 1 [170]. The Tanh function outputs 1 for large values and -1 for small values according to the following equation (4.15).

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad (4.15)$$

ReLU: In DNNs, the ReLU activation layer has a positive impact on the performance of the network. The ReLU activation function is used to avoid the gradient vanishing problem in DNNs. If the value of the input is less than 0, ReLU generates 0, and if it is greater than 0, it generates the same value [170]. In this way, ReLU speeds up the network compared to other

activation functions. The following Equation 4.16 is used in the ReLU layer. Figure 4.11 shows the activation function curves for Tanh and ReLU.

$$f(x) = \max(0, x) \quad (4.16)$$

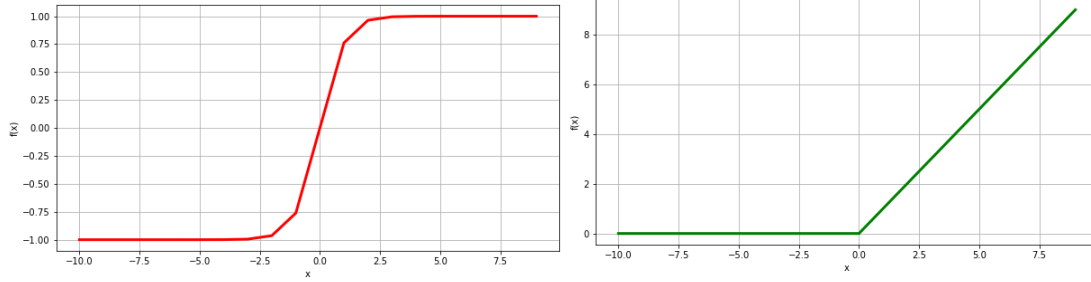


Figure 4.11: Tanh and ReLU activation functions

Softmax: The Softmax function is used for multi-class classification. The Softmax function takes a vector of real values from the NN and converts it into a vector of probabilities that sum to 1. This is used as the output layer for multi-class classification of the DNN model [170]. To calculate the probability, the following Softmax formula (4.17) is used.

$$\sigma(\vec{X})_i = \frac{e^{X_i}}{\sum_{j=1}^Z e^{X_j}} \quad (4.17)$$

where X is the input vector received from the NN, X_i is the elements of the input vector X , e^{X_i} is the exponential function applied to each element of X , and the $\sum_{j=1}^Z e^{X_j}$ term ensures that the probability of each element is in the range of 0 to 1, and the sum of the probabilities of all elements is 1, and Z is the number of class.

4.4.5 Evaluation Metrics

The performance of a classification model using NN depends on the elements of the confusion matrix. The confusion matrix has four elements: precision, recall, F1-score, and accuracy. Each element is evaluated based on four terms: true positive (T_P), false positive (F_P), true negative (T_N), and false negative (F_N). For example, c is the true class for a given problem p , and \bar{c} is the opposite of the true class of c . T_P classifies problems in class c as class c , ($c \rightarrow c$), while F_P classifies problems in class \bar{c} as incorrect class c , ($\bar{c} \rightarrow c$). On the other hand, T_N classifies problems in class \bar{c} as class \bar{c} , ($\bar{c} \rightarrow \bar{c}$), and F_N classifies problems in class c as incorrect class

\bar{c} , ($c \rightarrow \bar{c}$). *Precision*(P :) It is used to measure the correctness of a classification model. It indicates how many of the positive classifications are correct. Equation 3.31 is used for the precision. *Recall*(R :) This is used to measure the completeness of a classification model and is calculated as the ratio of T_P to the total number of actual classes ($T_P + F_N$). Equation 3.32 is used for the recall. *F1 – score* : It is calculated using the scores of P and R and is the harmonic mean of P and R . The F1-score is useful when the distribution of F_P and F_N is uneven. Equation 3.33 is used for the F1-score. *Accuracy* : It is an important metric to measure the performance of a model. The accuracy of a model is evaluated by the ratio of the total number of correct predictions to the total number of predictions.

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \quad (4.18)$$

4.4.6 Experimental Results

In this section, we show the experimental results generated based on different hyperparameter settings. In addition to the proposed stacked BiLSTM model, several state-of-the-art models such as LSMT and BiLSTM were trained and validated to compare the performance of the proposed model.

Figures 4.12, 4.13, and 4.14 show the accuracy and loss per epoch for the LSTM, BiLSTM, and stacked BiLSTM model during training and validation, respectively. We trained the models with different numbers of epochs and activation functions such as ReLU and tanh, and most of the models performed better when 150 epochs and ReLU activation function were used.

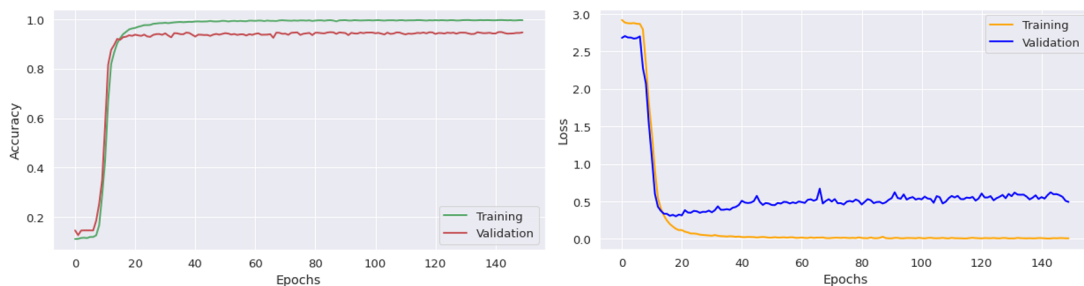


Figure 4.12: Accuracy and loss per epoch during training and validation for the LSTM model

The following observations can be drawn from these above figures: (i) the proposed model obtained higher training and validation accuracy than the LSTM and BiLSTM models and (ii) the proposed model has less loss than other two comparative models.

Considering these evaluation metrics, i.e., precision, recall, and F1 score, we compared

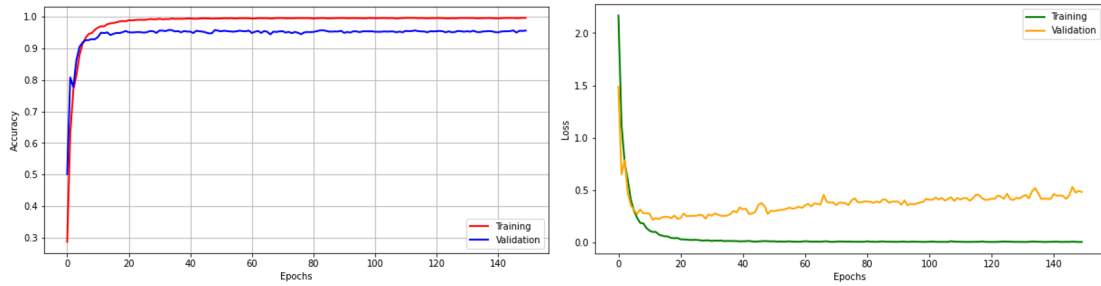


Figure 4.13: Accuracy and loss per epoch during training and validation for the BiLSTM model

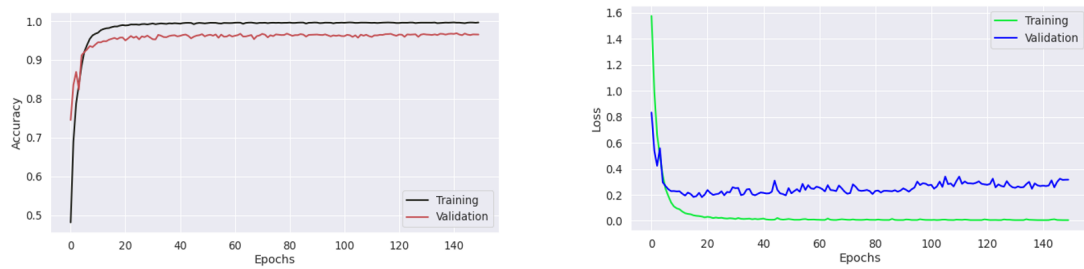


Figure 4.14: Accuracy and loss per epoch during training and validation for the stacked BiLSTM model

the classification results with other state-of-the-art models. The results show that the proposed stacked BiLSTM model significantly outperforms the other models. Table 4.8 shows the average precision, recall, and F1 scores of all the models considering the epoch number 150 and the activation function ReLU. It can be seen that the proposed stacked BiLSTM model outperforms the other state-of-the-art models in all three evaluation metrics.

Table 4.8: Average precision, recall, and F1-score of all models

| Model | Precision | Recall | F1-score |
|----------------|-----------|--------|----------|
| LSTM | 0.8248 | 0.8380 | 0.8116 |
| BiLSTM | 0.8456 | 0.8508 | 0.8348 |
| Stacked BiLSTM | 0.9012 | 0.8948 | 0.8924 |

The accuracy of all the models was also calculated considering the different number of epochs and activation functions as shown in Table 4.9 and Table 4.10, respectively. The proposed stacked BiLSTM model achieved better accuracy with both activation functions at different epochs.

Figure 4.15 shows a confusion matrix to visualize the prediction results of the proposed model. In the confusion matrix, the x -axis and y -axis represent the predicted label and the true label, respectively. The proposed model predicted the correct class for most of the test data,

Table 4.9: Average accuracy of the models using ReLU activation function

| Epochs | Models | | |
|--------|--------|--------|----------------|
| | LSTM | BiLSTM | Stacked BiLSTM |
| 25 | 0.89 | 0.89 | 0.91 |
| 50 | 0.90 | 0.90 | 0.91 |
| 100 | 0.88 | 0.92 | 0.93 |
| 150 | 0.91 | 0.91 | 0.93 |

Table 4.10: Average accuracy of the models using Tanh activation function

| Epochs | Models | | |
|--------|--------|--------|----------------|
| | LSTM | BiLSTM | Stacked BiLSTM |
| 25 | 0.88 | 0.87 | 0.92 |
| 50 | 0.89 | 0.91 | 0.92 |
| 100 | 0.89 | 0.91 | 0.93 |
| 150 | 0.90 | 0.90 | 0.93 |

but misclassification occurred for a very small amount of data.

In brief, the proposed BiLSTM model achieves better results than LSTM and other models for code evaluation and providing suggestions for code repair. We used two datasets, GCD and IS, for model training and evaluation. We presented the results of code assessments, suggestions, and overall performance of the model based on various evaluation metrics. In addition, we used stacked BiLSTM, a variation of BiLSTM, to classify multilingual source codes. In this classification task, the source codes used belonged to 25 different classes and were written in 14 different programming languages. The proposed stacked BiLSTM model produced satisfactory results compared to other state-of-the-art models. The model is applied to classify multilingual real-world source codes using various hyperparameters. The model outperforms the other models in each evaluation matrix such as precision, recall, F1 score, and accuracy. These results indicate that the proposed model has the potential to learn complex contexts of source codes of different programming languages. All the experimental results suggest that the stacked BiLSTM model can also be applied to other purposes of complex task classification.

4.5 Summary

It is generally recognized that conventional compilers and other code evaluation systems are unable to reliably detect logic errors (*WA*, *TLE*, *MLE*, *OLE*, *PrE*) and provide proper

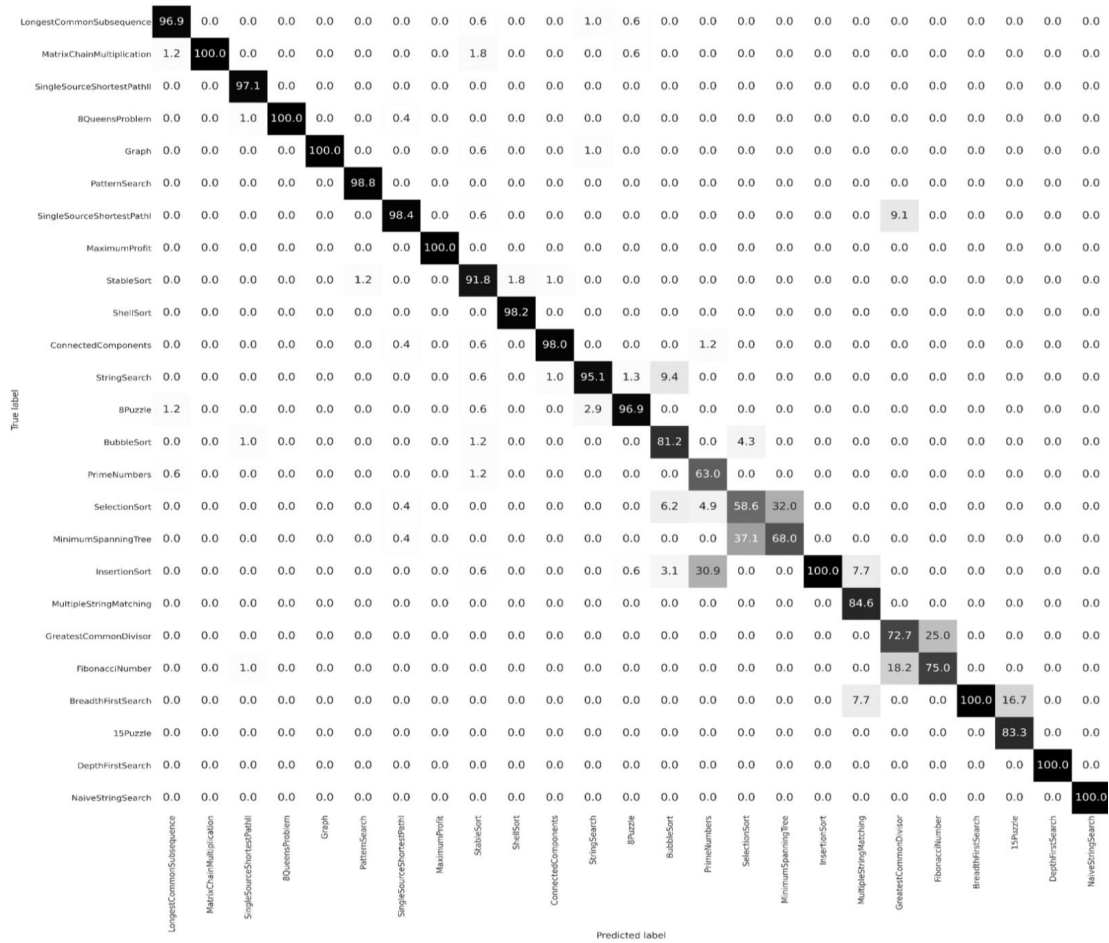


Figure 4.15: Confusion matrix for all classes using the stacked BiLSTM model

suggestions for code repair. While neural network-based language models can be effective in identifying errors, standard FNN or unidirectional RNNs have proven insufficient for effective source code evaluation. There are many reasons for this, including code length and the fact that some errors depend on both previous and subsequent code lines. In this chapter, we proposed an efficient BiLSTM neural network model for code assessment, repair, and binary classification (correct or incorrect). Importantly, the BiLSTM model has the ability to consider both the previous and subsequent context of the code under evaluation. In developing the model, we first trained the BiLSTM model using a large number of source codes. After that, we used the trained BiLSTM model for error detection and to provide suggestions for code repair. Experimental results showed that the BiLSTM model outperformed existing unidirectional LSTM and RNN neural network-based models. The CoM value of the BiLSTM model was approximately 50.88%, and F-score for binary classification was approximately 97%. The proposed BiLSTM model thus appears to be effective for detecting errors and providing relevant suggestions for code repair.

Furthermore, we proposed a multi-class classification model based on a stacked BiLSTM neural network. The architecture of the stacked BiLSTM neural network enables learning more complex features from the source codes for the multi-class classification task. For this experiment, we collected 35,000 real-world source codes written in different languages and classified them into about 25 classes. The proposed model and other state-of-the-art models were trained using these codes for the source code classification task. In the proposed model, the hyperparameters of the network were fine-tuned to achieve optimal results. The proposed model showed relatively good performance in terms of various evaluation metrics for multi-class classification. The average precision, recall, F1-score, and accuracy of the proposed model are 90.12%, 89.48%, 89.24%, and 93%, respectively, which are better than other state-of-the-art models such as LSTM and BiLSTM. Moreover, the performance of the proposed stacked BiLSTM model was evaluated for each class, and significant classification results were achieved.

Chapter 5

Conclusion and Future Research

5.1 Conclusion

OJ systems have recently become an important academic tool for automated programming assessment in educational institutions. Although traditional e-learning platforms offer many services, including online classes and lectures, various theory-based tests, assessments, and grading. However, the services of these traditional e-learning platforms are limited when it comes to assess the computer programming or other exercises. In this area, OJ systems are playing a key role in automated programming assessment. Since the OJ system regularly stores a large number of codes and a variety of information, these valuable data resources open up opportunities for educational research. In this dissertation, we focus on exploiting the real-world data resources collected from an OJ system to extract valuable features, hidden relationships, and flaws through educational data mining and learning analytics. Furthermore, we developed machine learning model for evaluating and classifying programming codes to improve programming learning.

In *Chapter 2*, a novel framework for exploring the effects of practical skills on academic performance is proposed. Subsequently, a programming course is selected as a sample course for experiments and analyses. By employing the framework, many meaningful and significant features are extracted from the dataset. The extracted features are deeply correlated to the students' behavior. The analytical results showed that better practical (e.g., programming) skills have a positive effect on academic performance. Moreover, the interaction and interdependence between programming skills and academic performance are presented based on the experimental results. Thus, we have concluded that if a student of an ICT or engineering discipline performs well in practical assignments (e.g., programming, logical implementation, PL/SQL), then they

are likely to perform well in other academic activities. The overall approach of Chapter 2 can be applicable to other fields such as education, big data analytics, and behavior analysis.

In *Chapter 3*, we proposed a neural network-based source code assessment model to assist students and programmers. The experimental results obtained in Chapter 3 show that the accuracy of error detection and prediction using the proposed LSTM-AM model is approximately 62%, whereas standard LSTM model accuracy is approximately 31%. In addition, the proposed model provides the location numbers for the predicted errors, which effectively limits the area that must be searched to find errors. Thereby reducing the time required to fix large source code sequences. Furthermore, the proposed model predicted correction words for each error location and detects logical errors (*WA, TLE, PrE, MLE, RTE, OLE*) that cannot be recognized by conventional compilers. Also, the LSTM-AM model shows great success in source code binary classification (correct or incorrect) than other state-of-the-art models. As a result, it is particularly suitable for application to long source code sequences and can be expected to contribute significantly to source code debugging and refactoring process. So, the proposed model is expected to be effective in providing end-to-end solutions for the programming learners and professionals. Despite the above-mentioned advantages, the proposed model also has some limitations. For example, error detections and predictions are not always perfect, and the model sometimes cannot understand the semantic meaning of the source code because of the incorrect detection and prediction that have been produced.

Lastly in *Chapter 4*, we proposed an efficient BiLSTM neural network model for code assessment, repair, and binary classification (correct or incorrect). Importantly, the BiLSTM model has the ability to consider both the previous and subsequent context of the code under evaluation. In developing the model, we first trained the BiLSTM model using a large number of source codes. After that, we used the trained BiLSTM model for error detection and to provide suggestions for code repair. Experimental results showed that the BiLSTM model outperformed existing unidirectional LSTM and RNN neural network-based models. The CoM value of the BiLSTM model was approximately 50.88%, and F-score for binary classification was approximately 97%. The proposed BiLSTM model thus appears to be effective for detecting errors and providing relevant suggestions for code repair.

Furthermore, we proposed a multi-class classification model based on a stacked BiLSTM neural network in Chapter 4. The architecture of the stacked BiLSTM neural network enables learning more complex features from the source codes for the multi-class classification task.

For this experiment, we collected 35,000 real-world source codes written in different languages and classified them into about 25 classes. The proposed model and other state-of-the-art models were trained using these codes for the source code classification task. In the proposed model, the hyperparameters of the network were fine-tuned to achieve optimal results. The proposed model showed relatively good performance in terms of various evaluation metrics for multi-class classification. The average precision, recall, F1-score, and accuracy of the proposed model are 90.12%, 89.48%, 89.24%, and 93%, respectively, which are better than other state-of-the-art models such as LSTM and BiLSTM. Moreover, the performance of the proposed stacked BiLSTM model was evaluated for each class, and significant classification results were achieved.

To summarize, we first proposed (in *Chapter 2*) an educational data mining framework to analyze the data collected from an OJ system. By employing the framework, many hidden features, patterns, and association rules are extracted from the data. The extracted features are deeply correlated to the students' behavior. Furthermore, the experimental results show that students made many errors (*WA, TLE, PrE, MLE, RTE, OLE*) in their solution codes that were not detected by the conventional compilers. In order to reduce the error rate in solution codes that cannot be detected by conventional compilers, we proposed (in *Chapters 3 and 4*) a machine learning-based source code assessment and classification model. The proposed machine learning-based models achieved significant results that can help students to reduce the error in their solution codes as well as improve programming learning.

5.2 Future Research

Although we have achieved significant results, there is still room for improvement in this dissertation for future research. First, the data analysis in *Chapter 2* was done using a programming course (ALDS1). We believe that this alone is not sufficient for an effective recommendation model for broader area. Therefore, in the future, we can focus on collecting data from a large number of programming and exercise courses for comprehensive analysis and recommendations. In addition, we plan to develop such a model that can be integrated with typical e-learning platforms (i.e., OJs and APAs) to provide more accurate recommendations that can help to improve the overall learning process.

On the other hand, we developed source code assessment and classification models using RNNs (i.e., LSTM, BiLSTM, and Stacked BiLSTM). In particular, we leveraged some specific

types of source code (i.e., C and $C++$) for training and testing to assess source codes. However, the ML-based source code assessment and classification models proposed in *Chapters 3 and 4* have achieved significant results and have provided a direction for future research in this area. Nevertheless, we believe that the current model is insufficient as a general-purpose model for evaluating all types of source codes (written in multi programming languages). Therefore, we plan to develop such a model for source code evaluation using a large amount of source codes written in different programming languages. Further, we consider to integrate that model with programming learning platforms (i.e., OJs and APAs) to provide learners with a more human-like assessment experience.

However, in this dissertation, we only focus on RNN-based models for code evaluation and classification. In recent years, Transformer-based models have achieved benchmark results in various language modeling tasks, such as programming code generation, Pseudo-code generation, code summarization, error detection, and classification. Therefore, as future works, we consider to develop a Transformer-based model for source code assessment and classification. Moreover, the benchmark dataset with real-world source codes is still sparse for AI research in the code domain. Since the AOJ system has a large number of real-world source codes, we are considering to build several benchmark datasets with these source codes, which will provide opportunities for various AI researches in this domain.

References

- [1] R. Rietsche, K. Duss, J. M. Persch, and M. Soellner, “Design and evaluation of an it-based formative feedback tool to foster student performance,” in *Proceedings of the 39th international conference on information systems (ICIS)*, 2018, pp. 1–17.
- [2] S. Oeste, K. Lehmann, A. Janson, M. Sollner, and J. M. Leimeister, “Redesigning university large scale lectures: How to activate the learner,” in *75th academy of management annual meeting*, 2015.
- [3] M. M. Rahman, Y. Watanobe, R. U. Kiran, T. C. Thang, and I. Paik, “Impact of practical skills on academic performance: A data-driven analysis,” *IEEE Access*, vol. 9, pp. 139 975–139 993, 2021.
- [4] R. Knote, A. Janson, S. Matthias, and J. M. Leimeister, “Value co-creation in smart services: A functional affordances perspective on smart personal assistants,” *Journal of the Association for Information Systems*, vol. 22, no. 2, pp. 418–458, March 2021.
- [5] S. Perez, “Voice-enabled smart speakers to reach 55% of u.s. households by 2022,” November 2017. [Online]. Available: <https://techcrunch.com/2017/11/08/voice-enabled-smart-speakers-to-reach-55-of-u-s-households-by-2022-says-report/>
- [6] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, “Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers,” *ACM SIGPLAN notices*, vol. 50, no. 4, pp. 223–238, April 2015.
- [7] H. Chung, M. Iorga, J. Voas, and S. Lee, “Alexa, can i trust you?” *Computer*, vol. 50, no. 9, pp. 100–104, September 2017. [Online]. Available: <https://doi.org/10.1109/MC.2017.3571053>
- [8] J. A. Kulik and J. D. Fletcher, “Effectiveness of intelligent tutoring systems,” *Review of Educational Research*, vol. 86, no. 1, pp. 42–78, March 2016.
- [9] W. Ma, O. O. Adesope, J. C. Nesbit, and Q. Liu, “Intelligent tutoring systems and learning outcomes: A meta-analysis,” *Journal of Educational Psychology*, vol. 106, no. 4, pp. 901–918, 2014.
- [10] J. C. Nesbit, O. O. Adesope, Q. Liu, and W. Ma, “How effective are intelligent tutoring systems in computer science education?” in *Proceedings of the IEEE 14th international conference on advanced learning technologies*, 2014, pp. 99–103.
- [11] K. Vanlehn, “The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems,” *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011.
- [12] A. Janson, M. Sollner, and J. M. Leimeister, “Ladders for learning: Is scaffolding the key to teaching problem-solving in technology mediated learning contexts?” *Academy of Management Learning Education*, vol. 19, no. 4, pp. 439–468, December 2020.

- [13] M. C. Kim and M. J. Hannafin, "Scaffolding problem solving in technology-enhanced learning environments (teles): Bridging research and theory with practice," *Computers & Education*, vol. 56, no. 2, pp. 403–417, February 2011.
- [14] T. A. Brush and J. W. Saye, "A summary of research exploring hard and soft scaffolding for teachers and students using a multimedia supported learning environment," *The Journal of Interactive Online Learning*, vol. 1, no. 2, pp. 1–12, 2002.
- [15] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–34, April 2018.
- [16] M. M. Rahman, S. Kawabayashi, and Y. Watanobe, "Categorization of frequent errors in solution codes created by novice programmers," *SHS Web of Conference*, vol. 102 (2021), May 2021.
- [17] M. A. Revilla, S. Manzoor, and R. Liu, "Competitive learning in informatics: The uva online judge experience," *Olympiads in Informatics*, vol. 2, pp. 131–148, 2008.
- [18] Y. Watanobe, "Aizu online judge," 2018. [Online]. Available: <https://onlinejudge.u-aizu.ac.jp>
- [19] "Aizu online judge: Developers site (api)," 2004. [Online]. Available: <http://developers.u-aizu.ac.jp/index>
- [20] Y. Watanobe, M. M. Rahman, T. Matsumoto, R. U. Kiran, and R. Penugonda, "Online judge system: Requirements, architecture, and experiences," *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2022.
- [21] J. L. Bez, N. A. Tonin, and P. R. Rodegheri, "Uri online judge academic: A tool for algorithms and programming classes," in *Proceedings of the 2014 9th International Conference on Computer Science Education*, 2014, pp. 149–152.
- [22] J. Petit, S. Roura, J. Carmona, J. Cortadella, J. Duch, O. Gimnez, A. Mani, J. Mas, E. Rodriguez-Carbonell, E. Rubio, E. d. S. Pedro, and D. Venkataramani, "Judge.org: Characteristics and experiences," *IEEE Transactions on Learning Technologies*, vol. 11, no. 3, pp. 321–333, July 2018. [Online]. Available: <https://doi.org/10.1109/TLT.2017.2723389>
- [23] A. C. Graesser, X. Hu, and R. Sottolare, *Intelligent tutoring systems*. Routledge, 2018.
- [24] S. Sweta, "Educational data mining in e-learning system," in *Modern Approach to Educational Data Mining and Its Applications*. Springer, January 2021.
- [25] C. Vaitsis, V. Hervatis, and N. Zary, "Introduction to big data in education and its contribution to the quality improvement processes," in *Big Data on Real-World Applications*. IntechOpen, July 2016.
- [26] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, pp. 165–193, 2015.
- [27] M. M. Rahman, Y. Watanobe, T. Matsumoto, R. U. Kiran, and K. Nakamura, "Educational data mining to support programming learning using problem-solving data," *IEEE Access*, vol. 10, pp. 26 186–26 202, 2022.
- [28] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Databases*, September 1994, pp. 487–499.

- [29] B. Kamsu-Foguem, F. Rigal, and F. Mauget, “Mining association rules for the quality improvement of the production process,” *Expert Systems with Applications*, vol. 40, no. 4, pp. 1034–1045, 2013.
- [30] M. M. Rahman, Y. Watanobe, U. K. Rage, and K. Nakamura, “A novel rule-based online judge recommender system to promote computer programming education,” in *Advances and Trends in Artificial Intelligence. From Theory to Practice*, H. Fujita, A. Selamat, J. C.-W. Lin, and M. Ali, Eds. Cham: Springer International Publishing, 2021, pp. 15–27.
- [31] J. Yuan and S. Ding, “Research and improvement on association rule algorithm based on fp-growth,” in *Proceedings of the International Conference on Web Information Systems and Mining*, 2012, pp. 306–313.
- [32] P. Wang, C. An, and L. Wang, “An improved algorithm for mining association rule in relational database,” in *Proceedings of the International Conference on Machine Learning and Cybernetics*, 2014, pp. 247–252.
- [33] S. P. Perumal, G. Sannasi, and K. Arputharaj, “An intelligent fuzzy rule-based e-learning recommendation system for dynamic user interests,” *The Journal of Supercomputing*, vol. 75, pp. 5145–5160, 2019.
- [34] M. Jiang, P. Cui, R. Liu, Q. Yang, F. Wang, W. Zhu, and S. Yang, “Social contextual recommendation,” in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, 2012, pp. 45–54.
- [35] S. S. Khanal, P. W. C. Prasad, A. Alsadoon, and A. Maag, “A systematic review: machine learning based recommendation systems for e-learning,” *Education and Information Technologies*, vol. 25, no. 4, pp. 2635–2664, July 2020.
- [36] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, “Sk_p: A neural program corrector for mooc,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 39–40.
- [37] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR ‘15)*, 2015, pp. 334–345.
- [38] K. Terada and Y. Watanobe, “Code completion for programming education based on recurrent neural network,” in *Proceedings of the 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*, 2019, pp. 109–114.
- [39] L. L. Minku, E. Mendes, and B. Turhan, “Data mining for software engineering and humans in the loop,” *Progress in Artificial Intelligence*, vol. 5, pp. 307–314, 2016.
- [40] A. Ram and M. Nagappan, “Supervised sentiment classification with cnns for diverse se datasets,” *arXiv*, 2018.
- [41] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015, pp. 1556–1566.

- [42] J. Reyes, D. Ramírez, and J. Paciello, “Automatic classification of source code archives by programming language: A deep learning approach,” in *Proceedings of the 2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 514–519.
- [43] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Software defect prediction via attention-based recurrent neural network,” *Scientific Programming*, vol. 2019, pp. 1–14, April 2019.
- [44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS’17)*, December 2017, pp. 6000–6010.
- [45] A. Radford and K. Narasimhan, “Improving language understanding by generative pre-training,” 2018.
- [46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv*, vol. arXiv:1810.04805, pp. 1–16, November 2018.
- [47] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.
- [48] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv*, vol. arXiv:1907.11692, pp. 1–13, July 2019.
- [49] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020, p. 1433–1443.
- [50] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, “Treegen: A tree-based transformer architecture for code generation,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, April 2020, pp. 8984–8991.
- [51] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” in *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020*, November 2020, pp. 1536–1547.
- [52] W. Gad, A. Alokla, W. Nazih, M. Aref, and A.-b. Salem, “Dlbt: Deep learning-based transformer to generate pseudo-code from source code,” *Computers, Materials & Continua*, vol. 70, no. 2, pp. 3117–3132, 2022.
- [53] A. Vee, “Understanding computer programming as a literacy,” *Literacy in Composition Studies*, vol. 1, no. 2, pp. 42–64, November 2013.
- [54] L. E. Margulieux, B. B. Morrison, and A. Decker, “Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples,” *International Journal of STEM Education*, vol. 7, no. 1, pp. 1–16, May 2020.
- [55] R. Yera and L. Martínez, “A recommendation approach for programming online judges supported by data preprocessing techniques,” *Applied Intelligence*, vol. 47, pp. 277–290, March 2017.

- [56] S. Manzoor, “Common mistakes in online and real-time contests,” *The ACM Magazine for Students*, vol. 14, no. 4, pp. 10–16, June 2008.
- [57] F. Okubo, T. Yamashita, and A. Shimada, “Students’ performance prediction using data of multiple courses by recurrent neural network,” in *Proceedings of the 25th International Conference on Computers in Education (ICCE)*, December 2017, pp. 439–444.
- [58] R. Romli, S. Sulaiman, and K. Z. Zamli, “Improving automated programming assessments: User experience evaluation using fast-generator,” *Procedia Computer Science*, vol. 72, pp. 186–193, January 2015.
- [59] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, “Automated assessment and experiences of teaching programming,” *Journal on Educational Resources in Computing*, vol. 5, no. 3, pp. 1–21, September 2005.
- [60] I. Mekterović, L. Brkić, B. Milašinović, and M. Baranović, “Building a comprehensive automated programming assessment system,” *IEEE Acces*, vol. 8, pp. 81 154–81 172, April 2020.
- [61] A. Kosowski, M. Małafiejski, and T. Noiński, “Application of an online judge & tester system in academic tuition,” in *Proceedings of the 6th international conference on Advances in web based learning (ICWL07)*, August 2007, pp. 343–354.
- [62] N. A. Rashid, L. W. Lim, O. S. Eng, T. H. Ping, Z. Zainol, and O. Majid, “A framework of an automatic assessment system for learning programming,” *Advanced Computer and Communication Engineering Technology (Lecture Notes in Electrical Engineering)*, vol. 362, pp. 967–977, December 2016.
- [63] M. M. Rahman, Y. Watanobe, and K. Nakamura, “Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education,” *Applied Sciences*, vol. 10, no. 8, p. 2973, April 2020.
- [64] R. Md Mostafizer, Y. Watanobe, and K. Nakamura, “A neural network based intelligent support model for program code completion,” *Scientific Programming*, vol. 2020, pp. 1–18, July 2020.
- [65] V. Hegde and S. Rao H.S., “A framework to analyze performance of student’s in programming language using educational data mining,” in *Proceedings of the 2017 IEEE International Conference on Computational Intelligence and Computing Research (IC-CIC)*, December 2017, pp. 1–4.
- [66] K. L. Ang, F. L. Ge, and K. P. Seng, “Big educational data & analytics: Survey, architecture and challenges,” *IEEE Access*, vol. 8, pp. 116 392–116 414, May 2020.
- [67] J. Knobbout and E. Van Der Stappen, “Where is the learning in learning analytics? a systematic literature review on the operationalization of learning-related constructs in the evaluation of learning analytics interventions,” *IEEE Transactions on Learning Technologies*, vol. 13, no. 3, pp. 631–645, July 2020.
- [68] Y. Maher, S. M. Moussa, and M. E. Khalifa, “Learners on focus: Visualizing analytics through an integrated model for learning analytics in adaptive gamified e-learning,” *IEEE Access*, vol. 8, pp. 197 597–197 616, October 2020.
- [69] L. F. Aleman, “Automated assessment in a programming tools course,” *IEEE Transactions on Education*, vol. 54, no. 4, pp. 576–581, November 2011.

- [70] R. E. Francisco and A. P. Ambrosio, "Mining an online judge system to support introductory computer programming teaching," in *Proceedings of the 8th International Conference on Educational Data Mining*, June 2015, pp. 1–6.
- [71] F. Restrepo-Calle, J. J. R. Echeverry, and F. A. González, "Continuous assessment in a computer programming course supported by a software tool," *Computer Application in Engineering Education*, vol. 27, no. 1, pp. 80–89, September 2018.
- [72] X. Lu, D. Zheng, and L. L., "Data driven analysis on the effect of online judge system," in *Proceedings of the 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, June 2017, pp. 573–577.
- [73] R. Y. Toledo, Y. C. Mota, and L. Martínez, "A recommender system for programming online judges using fuzzy information modeling," *Informatics*, vol. 5, no. 2, pp. 1–17, April 2018.
- [74] C. M. Intisar, Y. Watanobe, M. Poudel, and S. Bhalla, "Classification of programming problems based on topic modeling," in *Proceedings of the 7th International Conference on Information and Education Technology*, March 2019, pp. 275–283.
- [75] A. R. Anaya and J. Boticario, "A data mining approach to reveal representative collaboration indicators in open collaboration frameworks," in *Proceedings of the 2nd International Conference on Educational Data Mining (EDM)*, July 2009, pp. 210–219.
- [76] S. Kausar, X. Huahu, I. Hussain, Z. Wenhao, and M. Zahid, "Integration of data mining clustering approach in the personalized e-learning system," *IEEE Access*, vol. 6, pp. 72 724–72 734, November 2018.
- [77] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice java programmers through the analysis of online protocols," in *Proceedings of the 7th International Workshop on Computing Education Research*, August 2011, pp. 58–92.
- [78] M. M. Rahman, Y. Watanobe, and K. Nakamura, "An efficient approach for selecting initial centroid and outlier detection of data clustering," in *Proceedings of the 18th International Conference on Intelligent Software Methodologies, Tools, and Techniques (SOMET_19)*, September 2019, pp. 616–628.
- [79] H. Toivonen, "Sampling large databases for association rules," in *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996, pp. 134–145.
- [80] J. Park, M.-S. Chen, and P. Yu, "An effective hash-based algorithm for mining association rules," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 175–186, May 1995.
- [81] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 255–264, June 1997.
- [82] A. Savasere, E. Omiecinski, and S. B. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995, pp. 432–444.
- [83] D. W. Cheung, H. Jiawei, V. T. Ng, and C. Y. Wong, "Maintenance of discovered association rules in large databases: an incremental updating technique," in *Proceedings of the 12th International Conference on Data Engineering*, March 1996, pp. 106–114.

- [84] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD'00*, May 2000, pp. 1–12.
- [85] D. Ai, H. Pan, X. Li, Y. Gao, and D. He, "Association rule mining algorithms on high-dimensional datasets," *Artificial Life and Robotics*, vol. 23, no. 3, pp. 420–427, May 2018.
- [86] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-mine: Fast and space-preserving frequent pattern mining in large databases," *IIE Transactions*, vol. 39, no. 6, pp. 593–605, March 2007.
- [87] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, "A tree projection algorithm for generation of frequent item sets," *Journal of Parallel and Distributed Computing*, vol. 61, no. 3, pp. 350–371, March 2001.
- [88] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining frequent item sets by opportunistic projection," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002, pp. 229–238.
- [89] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 10, pp. 1347–1362, October 2005.
- [90] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 372–390, May 2000.
- [91] N. Yusof, N. A. M. Zin, and N. S. Adnan, "Java programming assessment tool for assignment module in moodle e-learning system," *Procedia - Social and Behavioral Sciences*, vol. 56, pp. 767–773, 2012.
- [92] A. Lile, "Analyzing e-learning systems using educational data mining techniques," *Mediterranean Journal of Social Sciences*, vol. 2, no. 3, pp. 403–419, September 2011.
- [93] E. Fernandes, M. Holanda, M. Victorino, V. Borges, E. Carvalho, and G. V. Erven, "Educational data mining: Predictive analysis of academic performance of public school students in the capital of brazil," *Journal of Business Research*, vol. 94, pp. 335–343, January 2019.
- [94] I. E. Livieris, K. Drakopoulou, V. T. Tampakas, T. A. Mikropoulos, and P. Pintelas, "Predicting secondary school students' performance utilizing a semi-supervised learning approach," *Journal of Educational Computing Research*, vol. 57, no. 2, pp. 448–470, January 2018.
- [95] S. A. Salloum, M. Alshurideh, A. Elnagar, and K. Shaalan, "Mining in educational data: Review and future directions," in *Proceedings of the International Conference on Artificial Intelligence and Computer Vision (AICV2020)*, vol. 1153. In: Hassanien AE., Azar A., Gaber T., Oliva D., Tolba F. (eds), Springer, March 2020, pp. 92–102.
- [96] T. P. Tran and D. Meacheam, "Enhancing learners' experience through extending learning systems," *IEEE Transactions on Learning Technologies*, vol. 13, no. 3, pp. 540–551, July 2020.
- [97] O. Viberg, M. Hatakka, O. Bälter, and A. Mavroudi, "The current landscape of learning analytics in higher education," *Computers in Human Behavior*, vol. 89, pp. 98–110, December 2018.

- [98] L.-K. Lee, S. K. S. Cheung, and L.-F. Kwok, "Learning analytics: current trends and innovative practices," *Journal of Computers in Education*, vol. 7, no. 1, pp. 1–6, February 2020.
- [99] F. Dunke and S. Nickel, "A data-driven methodology for the automated configuration of online algorithms," *Decision Support Systems*, vol. 137, p. 113343, June 2020.
- [100] T. Saito and Y. Watanobe, "Learning path recommendation system for programming education based on neural networks," *International Journal of Distance Education Technologies (IJDET)*, vol. 18, no. 1, pp. 36–64, 2019.
- [101] Y. Watanobe, C. M. Intisar, R. Cortez, and A. Vazhenin, "Next-generation programming learning platform: Architecture and challenges," in *Proceedings of the 2nd ACM Chapter Conference on Educational Technology, Language and Technical Communication*, November 2020, pp. 1–11.
- [102] M. M. Rahman, Y. Watanobe, and K. Nakamura, "A bidirectional lstm language model for code evaluation and repair," *Symmetry*, vol. 13, no. 2, p. 247, February 2021.
- [103] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv.org*, vol. arXiv:2105.12655, pp. 1–22, August 2021.
- [104] I. T. Jolliffe and C. Jorge, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society*, vol. 374, no. 2065, pp. 1–16, April 2016.
- [105] K. H. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv.org*, vol. arXiv:1608.02715, pp. 1–4, August 2016.
- [106] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–24, April 2018.
- [107] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (a google)," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014, pp. 724–734.
- [108] J. Pennington, R. Socher, and C. D. Manning, "Glove: global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, October 2014, pp. 1532–1543.
- [109] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 2014, pp. 269–280.
- [110] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8624–8628.
- [111] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, May 2018, pp. 60–70.
- [112] M. Pedroni and B. Meyer, "Compiler error messages: what can help novices?" in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, March 2008, pp. 168–172.

-
- [113] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, “Deepfix: fixing common c language errors by deep learning,” in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, February 2017, pp. 1345–1351.
- [114] Y. Teshima and Y. Watanobe, “Bug detection based on lstm networks and solution codes,” in *Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, October 2018, pp. 3541–3546.
- [115] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, May 2015, pp. 1–15.
- [116] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI’18)*, July 2018, pp. 4159–4165.
- [117] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 318–328.
- [118] Y. Watanobe, R. Md Mostafizer, R. Kabir, and A. Md Faizul Ibne, “Identifying algorithm in program code based on structural features using cnn classification model,” *Applied Intelligence*, 2022.
- [119] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, “Lessons learned from using a deep tree-based model for software defect prediction in practice,” in *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 46–57.
- [120] N.-Q. Pham, K. German, and G. Boleda, “Convolutional neural network language models,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, November 2016, pp. 1153–1162.
- [121] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL’96)*, June 1996, pp. 310–318.
- [122] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR’13)*, May 2013, pp. 207–216.
- [123] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, June 2013, pp. 1310–1318.
- [124] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, “Recurrent models of visual attention,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*, December 2014, pp. 2204–2212.
- [125] T. Matsumoto, Y. Watanobe, K. Nakamura, and Y. Teshima, “Logic error detection algorithm based on RNN with threshold selection,” vol. 327. IOS Press, 2020, pp. 76–87. [Online]. Available: <https://doi.org/10.3233/FAIA200554>
- [126] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, September 2015, pp. 1412–1421.

- [127] J. Chen, H. Zhang, X. He, L. Nie, W. Liu, and T.-S. Chua, "Attentive collaborative filtering: multimedia recommendation with item- and component-level attention," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'17)*, August 2017, pp. 335–344.
- [128] X. Ran, Z. Shan, Y. Fang, and C. Lin, "An lstm-based method with attention mechanism for travel time prediction," *Sensors*, vol. 19, no. 4, p. 861, February 2019.
- [129] Y. Yoshizawa and Y. Watanobe, "Logic error detection system based on structure pattern and error degree," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, no. 5, pp. 1–15, September 2019.
- [130] T. Matsumoto and Y. Watanobe, "Towards hybrid intelligence for logic error detection," *Advancing Technology Industrialization rough Intelligent Software Methodologies, Tools and Techniques*, vol. 318, pp. 120–131, September 2019.
- [131] T. Matsumoto, Y. Watanobe, and K. Nakamura, "A model with iterative trials for correcting logic errors in source code," *Applied Sciences*, vol. 11, no. 11, 2021.
- [132] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory networks for machine reading," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, November 2016, pp. 551–561.
- [133] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.
- [134] D. P. Kingma and B. Jimmy, "Adam: a method for stochastic optimization," in *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, May 2015, pp. 1–13.
- [135] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and qsar modeling," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 6, pp. 1947–1958, November 2003.
- [136] I. Sutskever, G. E. Hinton, and G. W. Taylor, "The recurrent temporal restricted boltzmann machine," in *Proceedings of the Advances in Neural Information Processing Systems*, December 2009, pp. 1601–1608.
- [137] G. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.
- [138] G. Samara, "A practical approach for detecting logical error in object oriented environment," *ArXiv*, vol. abs/1712.04189, 2017.
- [139] D. Al-Ashwal, E. Al-Sewari, and A. Al-Shargabi, "A case tool for java programs logical errors detection: Static and dynamic testing," in *Proceedings of the 2018 International Arab Conference on Information Technology (ACIT)*, November 2018, pp. 1–6.
- [140] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," *Journal of Systems and Software*, vol. 81, no. 3, pp. 447–460, March 2008.
- [141] G. Mou, L. and Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, February 2016, pp. 1287–1293.

-
- [142] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, September 2016, pp. 87–98.
- [143] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, August 2017, pp. 3034–3040.
- [144] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: A benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, pp. 531–577, 2012.
- [145] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI’14*, June 2014, pp. 419–428.
- [146] M. Rahman, Y. Watanobe, and K. Nakamura, “Evaluation of source codes using bidirectional lstm neural network,” in *Proceedings of the 3rd IEEE International Conference on Knowledge Innovation and Invention (ICKII)*, August 2020, pp. 140–143.
- [147] P. Le and W. Zuidema, “Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive lstms,” in *Proceedings of the 1st Workshop on Representation Learning for NLP*, August 2016, pp. 87–93.
- [148] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, November 1997.
- [149] M. M. Rahman, Y. Watanobe, R. U. Kiran, and R. Kabir, “A stacked bidirectional lstm model for classifying source codes built in mpls,” in *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Cham: Springer International Publishing, 2021, pp. 75–89.
- [150] “Intelligent coding environment (ice),” 2020. [Online]. Available: <https://onlinejudge.u-aizu.ac.jp/services/ice/>
- [151] S. Gilda, “Source code classification using neural networks,” in *Proceedings of the 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, July 2017, pp. 1–6.
- [152] H. Ohashi and Y. Watanobe, “Convolutional neural network for classification of source codes,” in *Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, October 2019, pp. 194–200.
- [153] M. Attia, Y. Samih, A. Elkahky, and L. Kallmeyer, “Multilingual multi-class sentiment classification using convolutional neural networks,” in *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC 2018)*, May 2018.
- [154] S. Mutuvi, E. Boros, A. Doucet, A. Jatowt, G. Lejeune, and M. Odeo, “Multilingual epidemiological text classification: A comparative study,” in *Proceedings of the 28th International Conference on Computational Linguistics*, December 2020, p. 6172–6183.
- [155] Z. Hameed and B. Garcia-Zapirain, “Sentiment classification using a single-layered bilstm model,” *IEEE Access*, vol. 8, pp. 73 992–74 001, April 2020.
- [156] D. Song, M. Lee, and H. Oh, “Automatic and scalable detection of logical errors in functional programming assignments,” in *Proceedings of the ACM Programming Languages*, October 2019, p. 1–30.

- [157] M. Allamanis, E. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, September 2018.
- [158] M. Allamanis, E. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, September 2015, p. 38–49.
- [159] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE’16)*, May 2016, p. 297–308.
- [160] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” in *Proceedings of the ACM Programming Languages*, November 2018, p. 1–25.
- [161] X. Song, C. Chen, B. Cui, and J. Fu, “Malicious javascript detection based on bidirectional lstm model,” *Applied Sciences*, vol. 10, no. 10, p. 3440, May 2020.
- [162] L. S. Larkey and W. B. Croft, “Combining classifiers in text categorization,” in *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1996, p. 289–297.
- [163] D. D. Lewis and W. A. Gale, “A sequential algorithm for training text classifiers,” in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1994, p. 3–12.
- [164] S. Ugurel, R. Krovetz, and C. L. Giles, “What’s the code?: Automatic classification of source code archives,” in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002, p. 632–638.
- [165] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional lstm and other neural network architectures,” *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, July 2005.
- [166] Z. Cui, R. Ke, Z. Pu, and Y. Wang, “Stacked bidirectional and unidirectional lstm recurrent neural network for forecasting network-wide traffic state with missing values,” *arXiv*, vol. arXiv:2005.11627, pp. 1–11, 2020.
- [167] M. Hermans and B. Schrauwen, “Training and analyzing deep recurrent neural networks,” in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, December 2013, p. 190–198.
- [168] E. Loper and S. Bird, “Nltk: The natural language toolkit,” in *Proceedings of the CL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, July 2002, p. 63–70.
- [169] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
- [170] T. Szandała, “Review and comparison of commonly used activation functions for deep neural networks,” In: Bhoi A., Mallick P., Liu CM., Balas V. (eds) *Bio-inspired Neurocomputing. Studies in Computational Intelligence*, vol. 903, pp. 203–224, July 2020.