Towards Deep Learning Models for Automatic Computer Program Grading

by

Peter Nagy

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Science (MSc) in Computer Science

Faculty of Science
University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

April 2023

THESIS EXAMINATION INFORMATION

Submitted by: **Peter Nagy**

Master of Science in Computer Science

Thesis Title: Towards Deep Learning Models for Automatic Computer Program Grading

An oral defense of this thesis took place on April 14, 2023 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Ramiro Liscano
Research Supervisor	Dr. Heidar Davoudi
Examining Committee Member	Dr. Ken Pu
Thesis Examiner	Dr. Faisal Qureshi

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Automatic grading of computer programs has a great impact on both computer science education and the software industry as it saves human evaluators a tremendous amount of time required for assessing programs. However, to date, this problem lacks extensive research from the machine learning/deep learning perspective. Currently, the existing auto-grading systems are mostly based on test-case execution results. However, these approaches lack insight into the syntax and semantics of the codes, and therefore, are far from human-level evaluation. In this study, we leverage the power of language models pre-trained on programming languages. We introduce two simple deep architectures and show that they consistently outperform the shallow models built upon extensive feature engineering approaches by a high margin. We also develop an incremental transductive learning algorithm that only requires a single reference solution to a problem and takes advantage of the correct implementations in the set of programs to be evaluated. Furthermore, our human evaluation results show that the proposed approaches provide partial marks having a strong correlation with the marks given by human graders. We prepare and share a dataset of C++ and Python programs for future research. Finally, we provide some interpretations and explainability of the deep-learning models as well as insights to the decisions and potential feedback to programming submissions in real-world applications.

Keywords: Automatic program grading; Pre-trained language representation models; Siamese network; Incremental learning; Contrastive learning; Interpretability

Author's Declaration

I hereby declare that this submission is entirely my own work, in my own words, and that all sources used in researching it are fully acknowledged. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

_____ Peter Nagy

Statement of Contributions

Part of this work described in Chapters 4, and 5 are being prepared for publication.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Kourosh Davoudi for his immense support and assistance in the accomplishment of my research objectives. He provided great faith and academic freedom for me to discover different subjects in my field. Additionally, he always showed me the right way when in doubt.

I would like to also thank my professors, Dr. Ken Pu, Dr. Faisal Qureshi, Dr. Patrick Hung, Dr. Fletcher Lu, and so much more, for their guidance during my undergraduate and graduate studies in Ontario Tech and who have captured my interest and lead the way to my research topic.

Lastly, I'm also deeply grateful for my family who have provided me support during the development of my research. It wouldn't have been possible without their assistance.

Table of Contents

Thesis Examination Information ii

Abstract iii

Author's Declaration iv

Statement of Contributions v

Acknowledgements vi

Table of Contents x

List of Tables xii

List of Figures xiv

1 Introduction **1**

1.1 Overview 1

1.2 Contribution 5

1.3 Thesis Outline 6

1.4 Software & Source Code 7

2 Literature Review **9**

2.1 Computer Program Grading 9

2.2 Transformer Models 10

3 Background **12**

3.1 Current Grading Practices 12

3.2 Natural Language Processing 14

3.3 Tokenization 14
3.4 Feature Extraction 14
3.5 Representation Learning 15
3.6 Structures in Programming Languages 16
3.7 Transformer Architecture 19
3.7.1 CodeBERT 19
3.7.2 GraphCodeBERT 19
3.7.3 CodeT5 20
3.7.4 PLBART 21
3.7.5 Codex 21
3.7.6 UniXcoder 22
3.7.7 Code-MVP 23
3.7.8 SynCoBERT 23
3.8 Other Models 24
3.8.1 Code2vec 24
3.8.2 Code2seq 25
3.9 Shallow Models 25
3.9.1 Random Forest 25
3.9.2 Support vector machine (SVM) 26
3.9.3 LASSO (L1) 26
3.9.4 Ridge regression (L2) 26

4 Methodology **28**

4.1 Problem Definition 28
4.2 Deep Grader 30
4.3 Deep Siamese Grader 31
4.4 Incremental Learning 32
4.4.1 Majority Vote Grading 32

4.4.2 Incremental Transductive Grading 34
4.5 Interpretability / Explainability 36
4.5.1 Attention Analysis 36
4.5.2 LIME 37
4.6 Feature Engineering Approaches 37
4.6.1 Feature Extraction 39
4.6.2 Feature Invariant Transformation 43
4.6.3 Shallow Prediction Models 46

5 Experiments and Results **47**
5.1 Data Collection and Preprocessing 47
5.2 Baselines and Experimental Setup 50
5.2.1 Feature Engineering 51
5.2.2 Feature Importance 52
5.2.3 Distance Functions 54
5.3 Proposed Models 58
5.3.1 The Effect of Majority Vote Grading 58
5.3.2 The Incremental Transductive Learning Analysis 60
5.3.3 Comparing with Human Grading 62
5.3.4 Runtime Analysis 64
5.4 Evaluation Methods 64
5.4.1 Question Independent Grading 65
5.4.2 Question Dependent Grading 66
5.5 Interpretability 67
5.5.1 Attention Analysis 67
5.5.2 Case Study of using LIME 78
5.6 Summary 80

6 Conclusion and Future work **81**

6.1 Thesis Contribution Highlights 82

6.2 Limitations 83

6.3 Future Work 83

Appendix **84**

7 Appendix I **85**

7.1 Training Details 85

7.2 Applications of Deep Grader in Program Synthesis 86

7.2.1 Overview 86

7.2.2 Improved Sampling Strategy 90

Bibliography **92**

List of Tables

4.1 Summary of notations and their definitions 29

4.2 Count-based feature categories 40

5.1 Dataset Statistics 48

5.2 Performance of models with single correct solution on *Python* code . . . 59

5.3 Performance of models with single correct solution on C^{++} code 59

5.4 Performance of models with multiple correct solutions on *Python* code . 61

5.5 Performance of models with multiple correct solutions on C^{++} code . . . 61

5.6 Performance of models on Incremental Transductive Grading on *Python* code 62

5.7 Performance of models on Incremental Transductive Grading on C^{++} code 63

5.8 Grading Rubric 64

5.9 Performance of models using single correct solution with a question dependent setting on *Python* code 66

5.10 Performance of models using single correct solution with a question dependent setting on C^{++} code 67

7.1 Training details 85

7.2 Example of a generated program with a correctness score of 81% and its reference correct solution. 90

7.3 Example of a generated program with a correctness score of 100% and its reference correct solution. 91

List of Figures

1.1 This program is only missing the "`cout`" statement, but it does not pass any test-cases. The human evaluators grade the program 3 and 4 out of 5. The proposed model predicts the grade of 3.32 out of 5. 2

3.1 A Python code snippet example. 17

3.2 An example of Abstract Syntax Tree. 18

3.3 An example of Flow Graphs. (a) Control Flow Graph, (b) Data Flow Graph of the expression x = a - b (c) Data Flow Graph of the expression x = a + b 27

4.1 Deep Grader Architecture 31

4.2 Deep Siamese Grader Architecture 33

4.3 Feature-based Model Pipeline 38

4.4 Control-flow graph 41

4.5 Statement data dependency graph 42

4.6 Expression data dependency graph 42

4.7 Abstract Syntax Tree 43

4.8 Complete Abstract Syntax Tree 44

5.1 Histograms presenting the distance levels in features for correct and incorrect programs used for evaluation. 49

5.2 Pair of graph representation feature distances for correct and incorrect programs used for evaluation. 51
5.3 Importance of graph-based features. 52
5.4 Impurity-based feature importance 53
5.5 Comparing examples of correct and incorrect programs using control-flow. 54
5.6 Comparing examples of correct and incorrect programs using data dependency. 55
5.7 Accuracy for each distance function (evaluated with Random Forest and *Python* code) 56
5.8 Percentage of minimum distances (evaluated with Random Forest and *Python* code) 57
5.9 Models' Grading Time per Submission 65
5.10 A sample program P^i 68
5.11 A correct program P^i_+ 68
5.12 Attention for the token "for" over heads and layers (Heads 1-6) 70
5.13 Attention for the token "for" over heads and layers (Heads 7-12) 71
5.14 Self-attention plot 72
5.16 Self-attention plot (partition) 75
5.17 Self-attention plot filtered (partition) 76
5.19 Program highlighted with attention weights 78
5.20 LIME explanation of a correct implementation 79
5.21 LIME explanation of an incorrect implementation 80

7.1 Example of a generated solution for a three component problem. 88
7.2 Effect of multiple components on the performance of generative models. . 89
7.3 Example of Sampling Tree 92
7.4 Sampling methods for generated programs 93
7.5 Distribution of Correctness Scores (k=10) 93

Chapter 1

Introduction

1.1 Overview

The automatic and accurate grading of computer programs is of paramount importance due to its tremendous impact on educational programming courses. Compared to manually grading programming assignments, the use of such tools can immensely reduce the time of grading and help maintain impartial grading across all students. Moreover, this technology can help software development companies largely in the recruitment process and assessment of the sheer volume of applicants by providing instant feedback on the programming ability of applicants.

The traditional auto-grading approaches require creation of an appropriate *test suite* for each question facilitating their prompt evaluation. However, building such test suites demands ample time and effort. To ensure loophole-free test suites designers are required to perform rigorous analysis on the coverage of all possible combinations of inputs and outputs for each problem. This limits the complexity of questions that can be asked since a higher complexity question requires a larger test suite and more time to ensure its validity and completeness. Moreover, in the case that a program fails to compile/execute or for certain reasons fails to pass any test-cases, there is no way to evaluate the program

using a test suite for potential partial marks (see Figure 1.1).

```cpp
#include <bits/stdc++.h>
using namespace std;

int main ()
{
    int n, minPrice, answer = 0;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        int x, y;
        cin >> x >> y;

        if (i == 0)
            minPrice = y;
        else
            minPrice = min(minPrice, y);

        answer += minPrice * x;
    }

    return 0;
}
```

Figure 1.1: This program is only missing the "`cout`" statement, but it does not pass any test-cases. The human evaluators grade the program 3 and 4 out of 5. The proposed model predicts the grade of 3.32 out of 5.

Although effective auto-grading of computer programs is a demanding need, there are only few studies on the analytical side that make this mechanism more effective [1, 62, 64]. To date, the majority of these works in machine learning are based on shallow models (e.g. LASSO, Support Vector Machine) designed on top of handcrafted features. While designing these features is a tedious task and may vary from one programming language to another one, they are far from deep insights, which are required for human-level assessment of programs.

In educational programming courses where evaluators manually grade programming submissions, the use of such tool can immensely reduce the time of grading and help

maintain impartial grading across all students. Currently, due to workload distribution, grading strictness varies across graders, and grading time is proportional to the program's length and complexity. There is an even bigger need of such technology in Massive Open Online Courses where class sizes can reach up to thousands of students. This is not the case for our model. Our automatic grading model is idempotent and grading time is invariant to the program's length and complexity. Our proposed model does not evaluate the program based on possible inputs and corresponding outputs, but instead evaluates the code itself just like a human evaluator would do.

For software development companies, this technology would largely aid in the recruitment process. Due to the sheer volume of applicants, the evaluation of programming skills is usually done with the aid of test-suites. The interviewer could get instant feedback on the validity of the code and programming ability of the applicant with the help of our model.

In the area of code generation, to measure the quality and correctness of generated code responding to natural language specifications, researchers have used automatic metrics such as BLEU score and test-cases [30]. As they find that BLEU is a troublesome metric, they sacrifice the evaluation of the quality of generated code for the correctness since their main objective is to produce correct programs. Test-case evaluation of generated code poses security risks (since generated code can have unknown intent and potentially be malicious) and requires the time-consuming creation of test-cases, thus it is unpractical in certain environments.

Recently, advancements in Natural Language Processing (NLP) approaches based on pre-trained language models (e.g., BERT [19]) have shown significant performance improvement in many natural language understanding tasks [19, 41, 54]. Similar ideas are adapted in programming languages models such as CodeBERT [23] and GraphCodeBERT [27]. However, to the best of our knowledge, these Tranformer-based [68] models have not been investigated for automatic program grading. We aim to further discover its

capabilities in the domain of programming language processing on our task. In this work, we leverage the advantages of models pre-trained on a large set of computer programs and adapt them for the problem of computer program grading.

The problem of grading computer programs is facing several other challenges from the data mining/machine learning point of view. Fine-tuning existing models requires a large number of labeled samples (i.e., programs and their grades) which are not usually available. Furthermore, in order to provide maximal effectiveness, the proposed model should be *invariant* to questions, that is, the model trained for marking a particular question should be used for assessing other questions as well.

To that end, we propose different models and end-to-end solutions for the problem of automatic computer program grading. Our proposed models take advantage of pre-trained models built upon large collections of unlabeled programs. In particular, our models leveraging the pre-trained models CodeBERT [23] and UniXcoder [28] are fine-tuned on the task of predicting the correctness of computer programs. Additionally, we implement a Siamese network architecture with a contrastive loss function, as defined in [29], to increase the memory-bounded limitation on the maximum size of input programs and to reduce the information loss on the model's output embedding.

In order to make the models invariant to a specific question, we feed models with a question-specific correct solution (as a reference) and use appropriate loss functions (i.e., cross-entropy loss, contrastive loss). We train the proposed models by utilizing *weakly* labeled samples, where the labels (i.e., correct/incorrect) are determined by the test suite results available for each program. That is, if the program fails on any test-case then it is incorrect, otherwise, it is correct. Our extensive experiments show that the proposed models trained on weakly labeled samples significantly outperform the shallow models. Moreover, our user study shows that the proposed approaches have a strong correlation with marks given by human evaluators. As a case study, we gather a small graded dataset of programs and use that to further evaluate the correlation and mean

squared error between the model's logits (non-normalized predictions) and the given grades. Considering the small size of this graded data, we cannot use it to train deep neural networks, and thus resort to training on test-case labeled data.

We also extend the proposed methods by approaching the problem from an instructor's point of view. In a real-world academic scenario, an instructor usually only provides a single solution to a problem instead of multiple solutions. However, the use of multiple solutions as different point of views can help the model make better predictions. Thus, to address this situation, we propose an incremental transductive algorithm built upon a majority vote scheme that takes advantage of the available correct implementations in the set of programs to be evaluated. We propose two algorithms, one of which allows us to utilize multiple correct references with our proposed models and the other, resembling more to a real-world environment, requires only one correct solution, but extends the solution set, and makes use of the prior algorithm to incorporate multiple points of reference in its prediction.

1.2 Contribution

The contributions of this thesis can be summarized as follows:

- We propose end-to-end deep architectures for the problem of grading computer programs and show the possibility of training them using weakly labeled samples. To the best of our knowledge, this is the first work investigating the potential of pre-trained deep learning models for this problem.

- We provide different neural network architectures to help the model's predictive capabilities in distinct ways.

- We provide a new inference method for more realistic scenarios, where only one correct solution is available, which is usually the case in the academic field. We

propose an incremental learning approach to train our model using only a single ground truth correct solution per problem. The incremental transductive learning approach takes advantage of correct programs in the test set based on a majority voting approach.

- We design comprehensive hand-crafted features capturing the program's syntax and structure. We further conduct extensive experiments and show that while the proposed architectures need less supervision than shallow models built upon hand-crafted features, they also outperform the shallow models by a high margin.
- We conduct a user study and find that partial grades predicted by the proposed models appear to have a strong correlation with human graders' marks.
- We collect and clean a dataset of C++ and Python programs, and release it for further research in evaluating the correctness of computer programs.
- We provide an analysis on explainability and interpretability of our proposed deep learning models including an attention weights analysis and insights of the models' decisions.
- We show the potential of using our proposed models in the task of program synthesis as a useful application.

1.3 Thesis Outline

This thesis is organized in six Chapters as follows:

- Chapter 2 goes over the related work in the field of Programming Language (PL) processing followed by more specific works related to the task of computer program grading.
- Chapter 3 presents the concepts and definitions necessary to understand the field.

- Chapter 4 presents the problem definition consisting of a high level description of our task, followed by our proposed models and baselines intended to produce maximal results for the automatic grading of programs.

- Chapter 5 presents our data, possible evaluation methods, the empirical results, and a user study to better evaluate the proposed models.

- Chapter 6 concludes the thesis, provides some limitations of the work, and suggests possible future work in the area.

- Chapter 7 presents an application of our proposed methods which can bring improvements to the task of program synthesis.

1.4 Software & Source Code

Software

Our models were implemented in the Python programming language. Python is a high-level programming language (PL) used as the main PL for machine learning (ML). The ML models in this research were trained and evaluated on graphical processing units (GPUs) due to their high computational efficiency and parallelization capability. Some of the necessary Python libraries used in this thesis are:

- NumPy is a library mainly used for scientific computation including multi-dimensional data manipulation and comprising of a large collection of mathematical functions.

- Pandas is a library used for dataset analysis thanks to its DataFrame object and coupled with data manipulation capabilities similar to NumPy.

- Tree-sitter is parsing library used in this research to assure the compilability of a program and to build concrete syntax trees from source code.

- PyMinifier is a tool used to reduce the size of python source code while maintaining its functionality with methods such as removing comments, dosctrings, unnecessary whitespaces, and so on. Although it required some modifications, it was used in the preprocessing of our data.

- PyTorch is a machine learning (ML) framework used to build and train ML models with a focus on deep neural networks.

- Scikit-learn is a library used for general machine learning (ML) including different ML algorithms such as the decision tree classifier, support vector machines, k-nearest neighbors algorithm, etc.

Source Code

The code implementations necessary to perform automatic grading of computer programs, including our proposed models, and the data used to train our deep-neural network models are available at the following link.

`https://github.com/peter-nagy1/Deep-Grader`

(**Please contact to request access.** This link is restricted due to the work being prepared for an anonymous publication.)

Chapter 2

Literature Review

2.1 Computer Program Grading

The grading of computer programs using machine learning has only been tackled with a feature-engineering approach in [1, 62, 64]. In this paper, they propose a grammar of features that is composed of six categories where each category specifies the type of code occurrences that are counted as features. The categories start from simple keyword counts to more complex data dependency and control-context between programming expressions. They then transform the features to make them invariant across questions. Finally, they experiment with multiple models including linear regression with L_1 regularization (LASSO), ridge regression, support vector machines (SVM), and random forest to train a supervised model for this task. This design necessitates a test-suite to acquire a number of correct programs in order to accurately grade responses to an unseen question. The main drawback of this approach comes from the features that need to be redefined specifically to the programming language at hand. Algorithms for certain graph representations of code would need to be rewritten for every programming language as they mostly depend on specific keywords and structure of code which vary across languages. Furthermore, this design inevitably requires multiple correct solutions as reference to accurately grade

responses which may not be feasible in deployment.

Outside of machine learning, few research papers proposed grading computer programs based on the number of test-cases passed [20, 37, 70, 80]. However, this approach does not evaluate the program's efficiency and style of writing. Furthermore, a minor error in an almost correct program can make it fail all test-cases judging it as completely incorrect by this approach (see Figure 1.1). On another note, a recent paper [44] has suggested grading computer programs by comparing their execution paths to a reference solution within a problem. In the case, that the proposed system finds a semantically different execution path it deems the program incorrect. On the downside, this approach cannot provide a partial grade to a program.

Other related papers focus on finding errors and providing feedback [59, 63] or correcting them [10, 26, 51] in computer programs. The task of plagiarism detection [85] or code clone detection [74, 79], involving the detection of similar programs, can also be linked to our task, as likewise, we attempt to deduce the similarity between program pairs. Some have approached their problem with recurrent neural network (RNN) models [10, 51] using Seq2seq [66] and LSTM [34] architectures. While others used methods involving abstract syntax tree (AST) representations of programs and developed complex algorithms such as AutoGrader [63] and CLARA [26] to solve their problem.

2.2 Transformer Models

Recently, great innovations have emerged in the natural language (NL) processing field due to the Transformer [68] architecture, more specifically, by the use of pre-trained large language models such as BERT [19], BART [41], T5 [54], etc. Similar models have surfaced in the programming language (PL) field based on their NL counter-part including CodeBERT [23], GraphCodeBERT [27], CuBERT [38], UniXcoder [28], PLBART [2], CodeT5 [78], etc. Some of these models have incorporated multiple input modalities to

pre-train their model other than the PL source code such as NL in the form of code comments and graph representations of code in the form of AST [28, 78] or data flow [27]. Although these models have aided to produce better results in many downstream tasks in the PL field, they have never been utilized for computer program assessment.

To measure the capabilities of a wide spectrum of state-of-the-art PL models, we decided to leverage CodeBERT and UniXcoder into our model architectures, since these pre-trained models have been proven to perform outstandingly on many code-related downstream tasks. To the best of our knowledge, CodeBERT is the earliest implementation of a pre-trained Transformer-based model dedicated for PL data. It is then continuously used by future research in the field as a strong baseline [2, 27, 28, 78]. Following CodeBERT, researchers introduce graph representations into their model, and amongst them, UniXcoder shows to outperform the previously mentioned rivals on most downstream tasks [28].

Chapter 3

Background

3.1 Current Grading Practices

Currently, educators and recruiters mainly use test-cases to automatically evaluate the programming competency of programmers. In the case that a submitted program passes a set of test-cases, the program is deemed correct. Usually, evaluators assign partial marks for programs that only pass a fraction of test-cases.

A test-case is a problem specific evaluation of a program which compares (e.g., using exact matching) an expected output against the program's output given a set of inputs. A complete set of test-cases make up a test-suite for a given problem. A test-suite should cover all possible combination of inputs that can be provided to the program and test all requirements of the problem. Formally, each requirement should be tested with a positive test-case and a negative test-case.

There are many shortcomings to the approach of using test-cases to evaluate programs.

- Creating a test-suite is time-consuming, laborious, and harder as programs get more complex. Currently, in educational programming courses, when evaluators manually grade programming submissions, due to workload distribution, grading

strictness tends to vary across graders, and grading time is proportional to the program's length and complexity. Additionally, there is a limit to the complexity of questions that can be asked since a higher complexity question requires a larger test-suite and more time to ensure its validity and completeness.

- Test-case evaluation fails in cases where the solution is close to perfect but missing an output related element. Due to the nature of unit-testing, if the program's output format is not exactly the same as intended by the test-suite, the program will fail all test-cases. A reoccurring common example is if the program outputs an extra newline character at the end of the output; then all test-cases will fail. Another frequent example is when the programming language numerical precision varies over compilations. The output floating point values will be slightly different at different executions of the code.

- Test-cases do not evaluate the content of the code, containing characteristics such as the programming ability of the coder, just like a human evaluator would do, but instead only evaluate outputs based on given inputs.

- Test-case evaluation may cause out of memory issues if the program is too large or too complex leading to failed test-cases and therefore incorrectly labeled programs. The same is true for the elapsed time. If a test-suite runs for too long then the program is marked incorrect due to a timeout inconsiderate of its actual correctness. These two aspects further limit the solution size and complexity of programming questions that can be asked.

- Test-case evaluation is unusable if a program does not compile. There exists multiple cases causing compilation issues depending on the programming language used such as incorrect indentation, missing semi-colon, etc. However, in some scenarios, the correctness of the program should not be entirely determined by its compilation

status. For example, in educational institutions, professors believe students should get part marks on their assignments even if their code doesn't compile.

3.2 Natural Language Processing

Natural language processing (NLP) is a field of machine learning studying the interaction between computers and human language. The field covers a wide range of tasks from natural language understanding to generation. Researchers proposed many machine learning solutions such as statistical methods by computing word occurrences and making probabilistic decisions, recurrent neural networks models with LSTM [34] and gated recurrent neural networks [14], and language models using word embeddings to capture semantic properties of natural language sentences. The most recent state-of-the-art advancements were brought by the discovery of the Transformer [68] which is a neural network architecture built using attention mechanisms.

3.3 Tokenization

The preprocessing phase of NLP pipelines mainly include a tokenization process. The process of tokenization involves a tokenizer which groups a sequence of characters into tokens. This process ensures the parsing of long text into individual tokens. The tokenizer will then perform indexing by assigning an index value for each distinct token. Hence, by converting natural language text into a sequence of indices, we can compute token representation embeddings using neural network models.

3.4 Feature Extraction

There exists many ways to extract feature vectors from input text. One way is to use machine learning models to extract representations from a sequence of tokens. Another

approach is to manually extract statistical features from the text such as counts of specific keywords or keyword combinations. This is also called feature engineering, where the goal is to extract the syntactic and semantics of a given text using the most descriptive features. In the case of computer programs, designing such features requires a deep knowledge of the programming language at hand. Since each programming language works differently and uses different syntax to achieve a certain goal. For example, there are low-level and high-level programming languages. There are also multiple programming paradigms such as procedural (e.g., C, FORTRAN), object-oriented (e.g., C++, Java), functional (e.g., Clojure, Scala), logical (e.g., Absys, Prolog), mathematical (e.g., Maple, MATLAB), and reactive (e.g., RxJava, RxJS) programming languages. There are also multi paradigm languages like Python. We can also extract structural properties of programming languages with structural representations such as abstract syntax trees, control flow graphs, and data flow graphs.

3.5 Representation Learning

There has been a growing interest in the field of *representation learning* over the past decade [40, 42, 47]. In representation learning, the goal is to capture the semantic representation of instances (e.g., text documents, words, images) in a low-dimensional vector space. That is, each instance is represented by a vector representing its rich set of features. The information captured in the representation vectors (i.e., embedding) can be used in many downstream tasks, such as similarity search [56], classification [39], and clustering [84] and it reduces/eliminates the need for expert knowledge in feature engineering.

For example, in the NLP domain, *context-free* models such as word2vec [47] and GloVe [49] learn word representations without considering the context of words (e.g., the sentence in which they are appearing). However, many words have different meanings depending on the context they are used (e.g., "Apple" for instance, may be a name

of company or a fruit depending on the context). The *context-based* models such as BERT [19] learns representations based on the context in which words appear. Therefore, BERT is more powerful in learning semantically meaningful representations that captures context-dependent meanings of words.

Most often, models, such as BERT, are optimized using *supervised-training*. That is, randomly selected tokens are masked from the input sentence before going through the encoder. The encoder learns the embedding vectors for each word/token in the sentence in order to predict the most likely word/token for each masked word/token. This process does not require manually annotated data and can take advantage of large unlabelled training data, which is available in many domains.

3.6 Structures in Programming Languages

In order to understand semantics of the code, we need to learn about the basic structures in the code and the way of extracting them. This is important as multiple methods [4, 5, 27, 28, 36, 75, 76] have leveraged the code's structural representation in order to incorporate the structure of the code containing crucial code semantics into the embedding space. A lexical analyzer or tokenizer is used to convert the code to a token-based sequence in which the order of tokens follows the order of appearance in code. Then, some models opt to utilize a parser, which is also known as a syntax analyzer, that produces an abstract syntax tree (AST) from the token-based sequence based on the grammar rules. The root node in this tree is the start symbol of the grammar, the interior nodes are the non-terminals in the grammar and the leaf nodes are the terminals, which are code tokens such as programming language specific keywords, variables and identifiers defined by the programmer. Finally, a semantic analyzer can utilize the AST to generate the flow graphs that contain the semantic information of the source code. There are two common flow graphs:

- Control Flow Graph (CFG): This graph illustrates different possible execution paths of a program.
- Data Flow Graph (DFG): This graph describes the dependency relation between variables in a program and shows how the data traverses across the variables.

Furthermore, since DFGs are only capable of representing basic blocks without branches (blocks without any conditions in other words), they can be replaced by the basic blocks of a CFG resulting in a control/data flow graph. Utilizing the following piece of code snippet in Python (Figure 3.1), we constructed the AST (Figure 3.2) as well as CFG (Figure 3.3.a) and DFG (Figure 3.3.b) to illustrate their definitions.

```
def subtract(a,b):
    x = 0
    if(a > b):
        x = a - b
    else:
        x = a + b
    return x
result = subtract(2,1)
print(result)
```

Figure 3.1: A Python code snippet example.

An **encoder-only** model encodes an input sequence, in our case, a set of code tokens, into internal state vectors which can be used for program classification or regression tasks. Whereas a **decoder-only** model predicts the next tokens given some previous tokens as context which is mainly used for code completion tasks or code generation tasks. Finally, an **encoder-decoder** model will encode an input sequence and then generate a

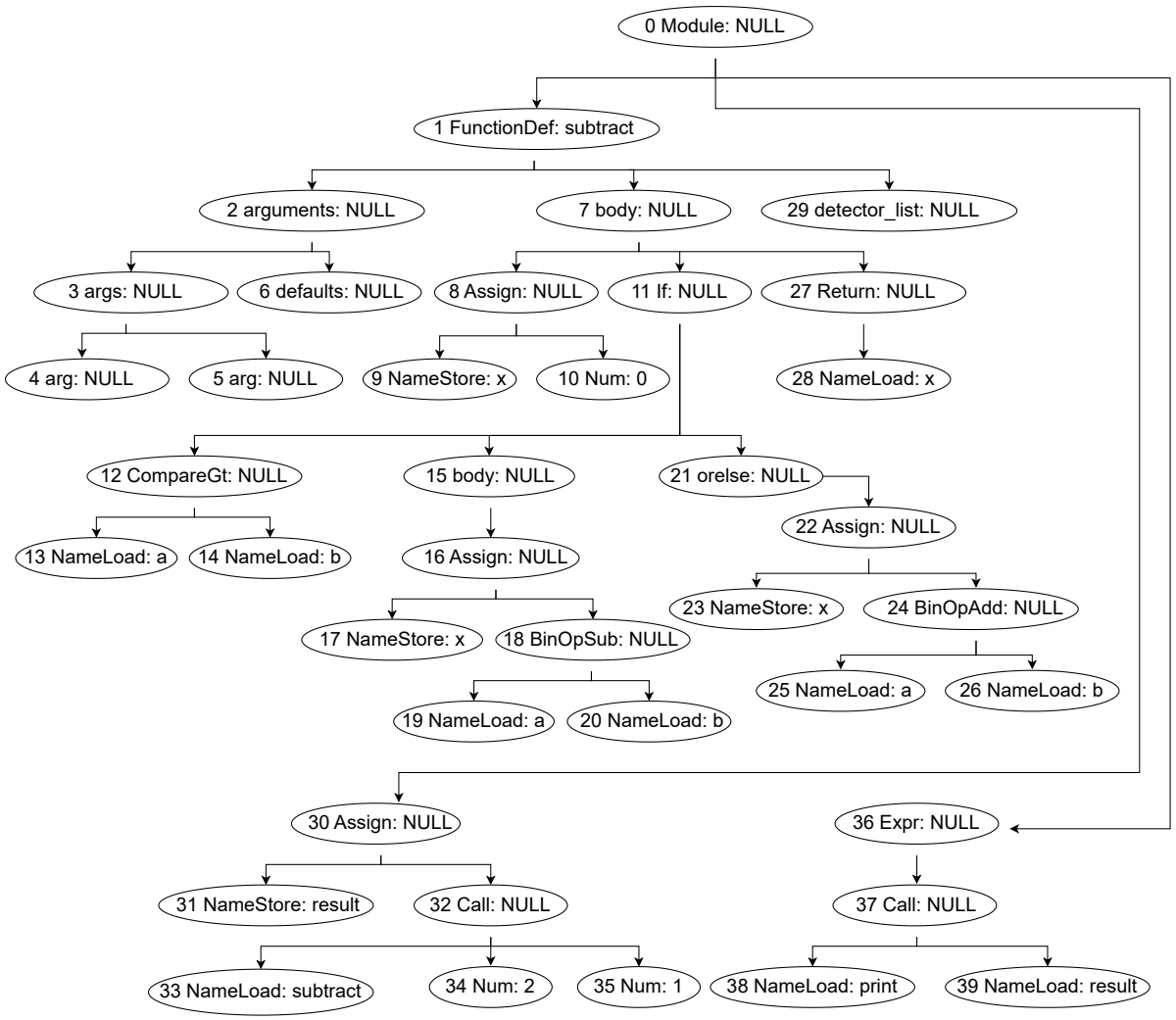


Figure 3.2: An example of Abstract Syntax Tree.

new output sequence not necessarily as a continuation of the input sequence, but as an original token sequence such as in code summarization or code translation tasks.

We mainly focus on deep-learning models developed for code understanding tasks. Some models have been developed with networks such as Recurrent Neural Networks (RNNs) [60], more specifically, Long Short-Term Memory (LSTM) [34], bidirectional LSTMs, or Graph Neural Networks (GNNs). Bidirectional LSTMs allow the input to flow in both directions through the neural network. GNNs can receive graph representation inputs such as ASTs, CFGs, DFGs and learn directly from the graphs instead of learning from a sequence representation of the graphs.

3.7 Transformer Architecture

The Transformer [68] architecture is constructed using an *attention mechanism* connecting the encoder to the decoder and assigning attention weights which inform the model about which input tokens should be more influential. The encoder associates the input tokens with each other while learning their representation through a self-attention mechanism (which is also used by the decoder when receiving the encoder's output).

3.7.1 CodeBERT

CodeBERT [23] is a pre-trained programming language model utilizing an *encoder-only* architecture identical to that of the base RoBERTa [45] model. RoBERTa is derived from BERT [19], both using a Transformer-based architecture. CodeBERT is trained using the Masked-Language Modeling (MLM) objective, consisting of predicting randomly masked code tokens, as well as the Replaced Token Detection (RTD) objective, aiming to predict whether a token at a specific location appears in the original program or it has been replaced. Both of these objectives were originally developed for natural language models and then adapted to programming language models. The adaptations mainly involve addressing both *bimodal* (i.e., training based on pairs of natural language and programming language examples) and *unimodal* (i.e., training upon pairs of two programming language instances) data. The model has a total of 125M parameters which are trained on the CodeSearchNet [35] dataset with six programming languages including Go, Java, Javascript, PHP, Python, and Ruby.

3.7.2 GraphCodeBERT

GraphCodeBERT [27] introduces data flow into the architecture with a graph-guided masked attention function. GraphCodeBERT improves on CodeBERT in programming language understanding and code representation by leveraging the code's structure with

Data Flow Graphs (DFGs). The code's structure is incorporated into the model using a graph guided masked attention function. In addition to the MLM objective, it introduces two new objectives for pre-training in a structure-aware manner. The first one consists of predicting data flow edges between variable nodes which they refer to as Edge Prediction (EP), and the second involves predicting edges between variable nodes and source code tokens with is referred to as Node Alignment (NA). This encoder-only model follows the BERT architecture using a multi-layer bidirectional Transformer design. They also utilize code comments in their pre-training data which most pre-trained models omit as it can be misleading and does not affect the execution of the program. This model contains 125M trainable parameters and was trained on the CodeSearchNet dataset using the six given programming languages.

3.7.3 CodeT5

CodeT5 [78] is an encoder-decoder pre-trained model which is able to perform with both code understanding tasks and code generation tasks. This model's architecture is based on the T5 [54] NLP model. They introduce new pre-training tasks including Masked Span Prediction (MSP) which masks arbitrary length spans of text and then attempts to predict them. They also makes use of the Identifier Tagging (IT) task where the goal is to predict if a certain code token is an identifier. Furthermore, using the Masked Identifier Prediction (MIP) task, the model learns to predict the missing identifier code tokens from the masked source code. Finally, they employ the Bimodal Dual Generation (BDG) task which consists of generating NL or PL from a NL-PL pair. This can also be viewed as a MSP where the NL or the PL is masked in one single span and the task is to predict the span. The CodeT5 model was built with two sizes: a small version consisting of 60M parameters and a base version consisting of 220M parameters. It was pretrained on two datasets: CodeSearchNet and the GitHub fraction of Google BigQuery dataset

[1]. In addition to the six programming languages present in the CodeSearchNet dataset, the model is also trained on C and C# programming languages.

3.7.4 PLBART

PLBART [2] is another encoder-decoder pre-trained model originating from the BART [41] model's architecture well established in the NLP field. This model is pre-trained with a single objective named Denoising Autoencoding (DA). The objective consists of reconstructing an input text affected by a noise function. The input is modified with noise by masking certain tokens (similar to MLM), by deleting certain tokens or by masking out spans of tokens (similar to MSP). The PLBART model has 140M parameters and was trained on the Java and Python written GitHub fraction of Google BigQuery dataset and they used data from StackOverflow to extract NL questions and answers to programming problems. They also evaluated their model on seven additional programming languages (i.e., Go, JavaScript, PHP, Ruby, C, C++, C#) which were not presented to the model during its pre-training phase. The PLBART model performed surprisingly well, even outperforming other models that were explicitly pre-trained on those languages on several tasks.

3.7.5 Codex

Codex [13] introduces a decoder-only model that generates code solutions based on a given natural language problem denoted as the context. This model's architecture was inspired by the GPT-3 [12] model family and showed performance improvements on code related tasks over GPT models which were partially trained on code. They train this model with the objective of minimizing the negative log-likelihood between the reference code and the generated code. They publish this model in multiple sizes ranging from

[1]https://console.cloud.google.com/marketplace/details/github/github-repos

12M to 12B parameters. The model is trained on a large dataset (179 GB) of repositories from GitHub using only the Python programming language. This model proves capable of generating solutions to a wide variety of introductory difficulty problems, however, it falls short when tasked with harder problems. It is presumed that the Codex model under-performs a strong student having completed an introductory computer science course.

3.7.6 UniXcoder

UniXcoder [28] is a general pre-trained model that proposes an *encoder-only*, a *decoder-only*, and an *encoder-decoder* framework for a range of code-related tasks. The different behavior modes (encoder-only, decoder-only, and encoder-decoder) are enabled using an input prefix (`[Enc]`, `[Dec]`, and `[E2D]`). The model takes advantage of code comments and Abstract Syntax Tree (AST) to enrich code representations. While code comments could be helpful for translation-related tasks (e.g., translating from one programming language to another one), abstract syntax trees retain all structural information of codes in embeddings. They utilize multiple pre-training tasks including Masked Language Modeling (MLM) in the encoder-only mode, Unidirectional Language Modeling (ULM) for the decoder-only mode, where the task consists of predicting the next token conditioned on previous tokens. They further employ a DeNoiSing (DNS) objective in an encoder-decoder mode which consists of predicting random masked spans. The model also learns semantic embeddings from multi-modal code using Multi-modal Contrastive Learning (MCL), and Cross-Modal Generation (CMG). The former employs a contrastive learning approach based on SimCSE [25] framework and the latter consists of generating code comments. They also mention that while AST representations are crucial in pre-training the model, they are not required when fine-tuning the model to specific downstream tasks. The model includes 125M trainable parameters and follows a Transformer-based model architecture. The UniXcoder model was trained on the CodeSearchNet [35] dataset and

additionally, on the C4 dataset [54], consisting of English text, with a denoising objective to help the model with various bimodal tasks.

3.7.7 Code-MVP

Code-MVP [76] presents an encoder-only pre-trained model which integrates multiple representations of a program in the model input at the same time with a contastive learning framework. Thus, they extract the natural language (NL) description/comment, the programming language (PL) source code, the abstract syntax tree (AST) representation, the control flow graph (CFG) representation, and the program transformation (PT) variant from a program and use those views as the input to the model. They utilize various functionally-invariant PT techniques to help the model understand functional semantics and also serves as a data augmentation. The model is trained on Multi-View Masked Language Modeling (MMLM) which predicts masked out tokens from data points. The model is also trained on Multi-View Contrastive Learning (MVCL) which functions differently depending on a single-view or a dual-view approach. In both cases they perform contrastive learning which requires positive and negative samples. In a single-view approach, positive samples consist of paired identical programs with different views and negative samples consist of different programs with different views. In the dual-view scenario, the setup is similar, however, they prepend the program pairs with their corresponding NL representation. Finally, the model is trained using the Fine-Grained Type Inference (FGTI) task which consists of predicting the type information of code tokens. They use a model containing 125M trainable parameters and they train on the CodeSearchNet dataset solely on Python code.

3.7.8 SynCoBERT

SynCoBERT [75] provides a pre-trained encoder-only model derived from BERT. The parameters and initializations are adopted from CodeBERT and GraphCodeBERT. The

dataset used and learned programming languages are also identical to CodeBERT. SynCoBERT, however, introduces a contrastive pre-training approach, coupled with AST representations, and two novel pre-training tasks. First, the model is trained on Multi-Modal Masked Language Modeling (MMLM), similar to Code-MVP, although, in this case, the modalities used are NL, PL, and AST. They form triplets from these modalities for each data point and predict masked tokens from the data sequences. They further pre-train using the novel Identifier Prediction (IP) task which consists of predicting whether tokens are identifiers or not as binary classification task. Furthermore, they use AST Edge Prediction (TEP) to encode structural information directly from AST into the model by predicting masked edges between tree nodes. The main goal is to predict whether two nodes have an edge between them or not. Finally, they employ Multi-Modal Contrastive Learning (MCL), similar to Code-MVP, they create positive and negative samples by pairing different combinations of NL, PL, and AST and train to maximize the similarity for positive pairs while minimizing it for negative pairs.

3.8 Other Models

Some influential models have been proposed which do not rely on a Tranformer architecture, however, they do utilize the attention mechanism. Although, they are not large models, they introduce important and successful code representation models.

3.8.1 Code2vec

Code2vec [5] provides an encoder-only trained model which is designed to aggregate syntactic paths from the AST representation of source code into a single vector. The model architecture is a path-based attention model which is trained with the objective of predicting a probability distribution of assigned tags to code snippets. The model is inspired by the NL model Doc2vec [40]. Code2vec was trained on a Github dataset

containing 14M Java methods.

3.8.2 Code2seq

Code2seq [4] presents an encoder-decoder model which is designed to convert source code snippets to natural language sequences. It was inspired by the Seq2seq [66] NL model. Similarly to Code2vec, source code snippets are represented as compositional paths in their AST representation. The encoder takes in a vector representation of each path in the AST rather than code tokens. Code2seq was trained to predict the next token based on previously generated tokens as context. The model architecture encapsulates a combination of bidirectional LSTMs coupled with an attention mechanism in the decoder. The model was trained on Java (Java-Small, Java-Med, Java-Large) and C# datasets from GitHub and has a total of 37M trainable parameters.

3.9 Shallow Models

3.9.1 Random Forest

The Random Forest [33] classifier is composed of a collection of decision tree classifiers which are fitted to distinct sub-sets of the training data. The average of the decision trees' predictions will be considered as the final prediction. The classifier can also control over-fitting by adjusting the characteristics of the decision trees in order to reduce their complexity. For example, reducing the maximum tree depth of the decision trees, reducing the number of splits at each node, and reducing the number of variables considered at each split increase the variability of the decision trees.

3.9.2 Support vector machine (SVM)

Support vector machines [17, 22] are considered a non-probabilistic binary linear classifier mapping training instances to points in an embedding space while maximizing the gap between the two classes. New points can then be classified based on which side of the separation line they fall in.

3.9.3 LASSO (L1)

The least absolute shrinkage and selection operator (LASSO) [67] is a linear regression model with L_1 regularization performing variable selection and regularization to enhance the predictive capability and interpretability of regression models.

3.9.4 Ridge regression (L2)

The ridge regression [31] method is a linear regression model with L_2 regularization which estimates the coefficients of multiple-regression models when the variables are highly correlated. The method was mainly developed to tackle multicollinearity in linear regression.

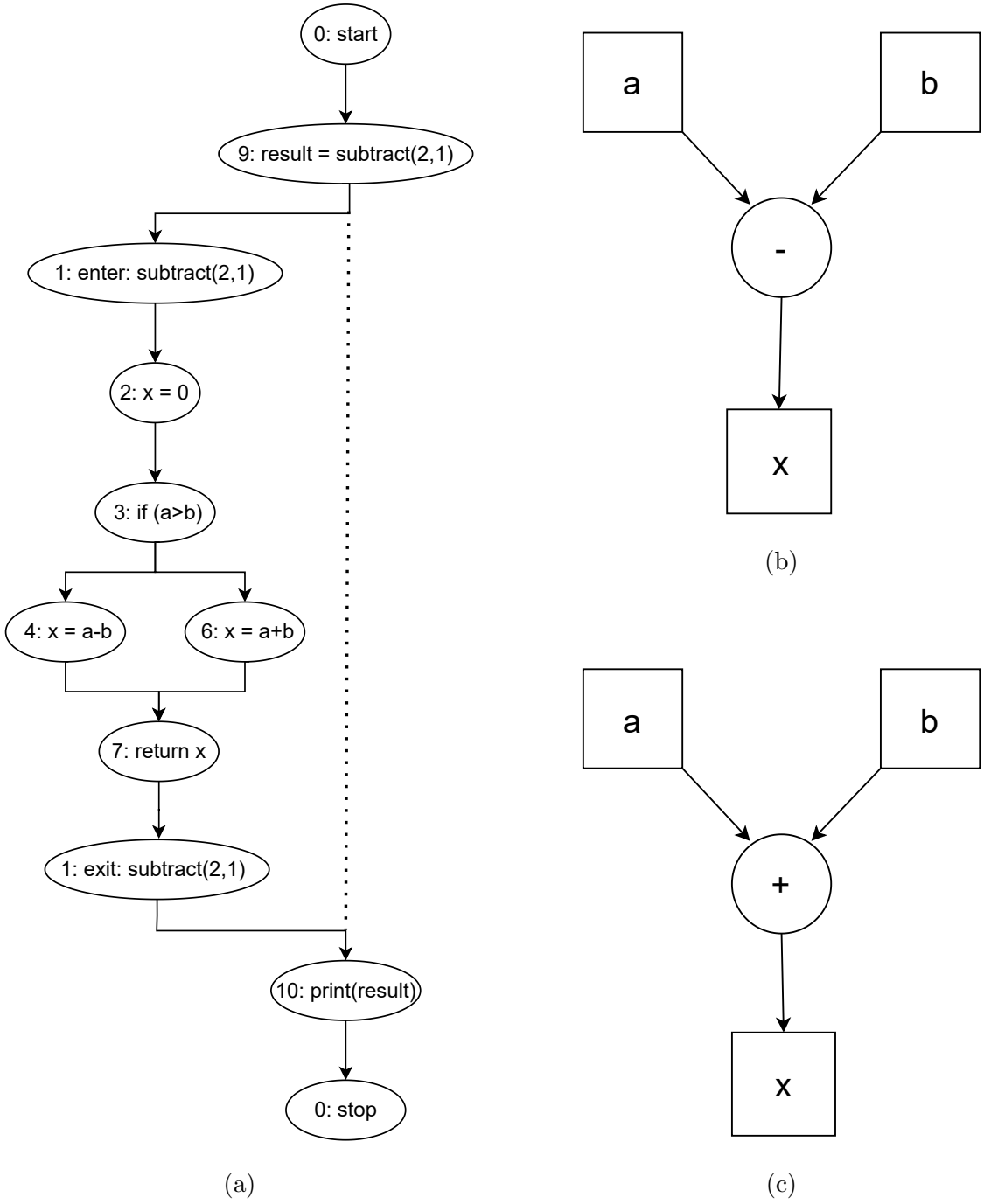


Figure 3.3: An example of Flow Graphs. (a) Control Flow Graph, (b) Data Flow Graph of the expression x = a - b (c) Data Flow Graph of the expression x = a + b

Chapter 4

Methodology

4.1 Problem Definition

The true assessment of a computer program's grade requires a thorough analysis by external expert evaluators. However, building a large dataset of programs paired with their corresponding true labels (i.e., grades) is an extremely time-consuming task. To that end, we utilize weakly-labeled samples where the label is determined by a test suite created for each given problem. A program is deemed *correct* or positive if it passes all test-cases within the test suite, otherwise, it is deemed *incorrect*. More formally, the *j'th* solution for problem i is denoted by $P^i_j =< s^i_{j,1}, s^i_{j,2}, \ldots, s^i_{j,m_j} >$, where $s^i_{j,k}$ is the *k'th* token in the *j'th* solution for problem i. The label $l^i_j = 1$ if P^i_j is correct, and $l^i_j = 0$ if P^i_j is incorrect. Our training dataset is composed of programs and corresponding labels as follows:

$$D = \{(P^i_j, l^i_j) | i = 1, 2, \ldots \; j = 1, 2, \ldots\} \tag{4.1}$$

where P^i_j is the *j'th* program for problem i and its corresponding label l^i_j. Furthermore, in the following sections, we assume P^i_+ is a correct solution for problem i, and P^i is a solution that could be correct or incorrect.

Problem Statement: Given a training set D, the goal is to build a model that takes a correct program/solution $\widehat{P}_{+}^{q} \notin D$ for problem q, and estimates the correctness of an unseen solution $\widehat{P}^{q} \notin D$ for an unseen problem/question q.

It is worth mentioning that we are considering only smaller sized programs composed of about 512 tokens. However, it would be possible to adapt a NL model like Longformer [9] that can handle large sequences of text to the PL field. We consider this as a potential future work.

A program is correct or positive if it is verified by an external party to be correct. In our data, correct programs are those that pass the test-suite created for the given problem. However, there is no need for a test-suite if a human expert evaluates the program as correct. A program is deemed incorrect or negative if it fails a test-case inside the test-suite.

Table 4.1: Summary of notations and their definitions

Notation	**Definition**
D	the dataset
P	the collection of programs
s	the collection of tokens in a program
l	the labels assigned to a collection of programs (correct/incorrect)
P^{i}	an unlabeled solution to a given problem i
P_{+}^{i}	a correct solution to a given problem i

We propose two architectures leveraging these pre-trained models for programming language grading.

4.2 Deep Grader

Figure 4.1 shows the Deep Grader architecture. We create our dataset by combining all coding submissions with a random correct submission belonging to the same question. In this way, we can achieve question-independent grading since the model only has to learn to differentiate the two input submissions irrespective of the problem. Furthermore, this architecture requires only one single correct reference to successfully assess the grade of a new program. The programs are then processed through a pre-trained *tokenizer* which joins the two programs into one sequence padded and truncated to a fixed sequence length.

The token sequence starts with the `[CLS]` special token followed by two tokenized programs separated by a `[SEP]` token and ends with the `[EOS]` token. For example, a program pair (P^i, P^i_+) is encoded as:

$$< \texttt{[CLS]}, P^i, \texttt{[SEP]}, P^i_+, \texttt{[EOS]} > \tag{4.2}$$

Note that P^i_+ is a correct solution for question i, and P^i is a solution that could be correct or incorrect. In addition to this token sequence, the tokenizer also provides the pre-trained model with an attention mask which specifies which program each token belongs to.

The output of the pre-trained models incorporates contextual vector representation for each token and an aggregate representation of the sequence is captured in the `[CLS]` token embedding, denoted by $\vec{C}$. The pre-trained model's output (i.e., $\vec{C}$) is then fed into a softmax layer to provide the final output.

The model is trained using the cross-entropy loss:

$$L\left(W, Y, P^i, P^i_+\right) = -(Y^i \log(\widehat{Y}) + (1 - Y^i) \log(1 - \widehat{Y})) \tag{4.3}$$

where W is the model parameters, $\widehat{Y}$ is the predicted probability of P^i's correctness, and Y^i is the class label, which is 1 if P^i is a correct program, and 0 otherwise.

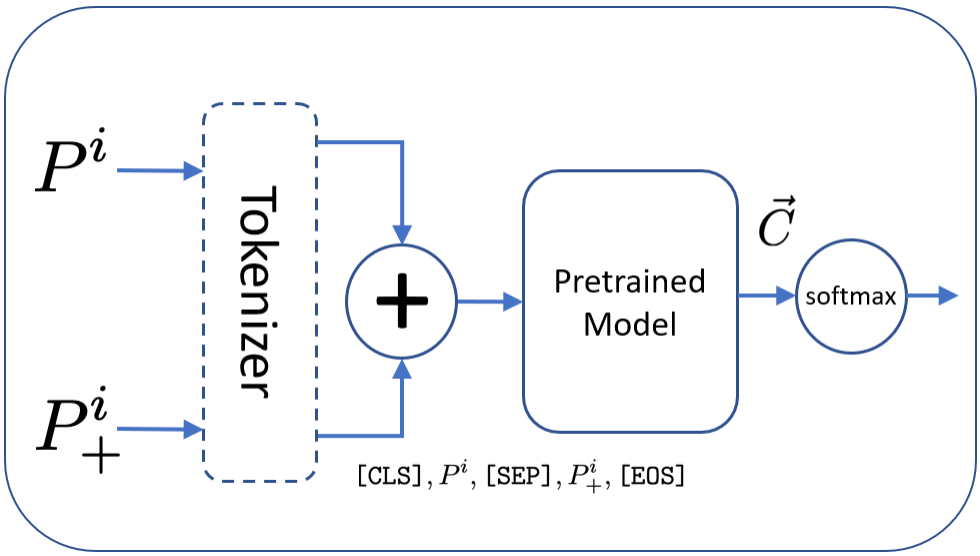


Figure 4.1: Deep Grader Architecture

A sample program P^i and a reference solution P^i_+ are tokenized using a pre-trained tokenizer (i.e., CodeBERT, UniXcoder). The tokenized programs are concatenated into one sequence separating the program tokens by a separator token `[SEP]`. This sequence is fed through the pre-trained model, and the `[CLS]` token embedding denoted by $\vec{C}$ is the model's output. The $\vec{C}$ is then passed through a softmax layer to obtain a normalized output.

4.3 Deep Siamese Grader

Figure 4.2 shows the Siamese-based architecture for grading computer programs. The anchor program P^i and correct program P^i_+ are tokenized independently and fed into two separate instances of the pre-trained models sharing weights with each other. The model uses the contrastive loss function [29] to learn the parameters. That is, the outputs of the models, which represent low-dimensional embedding of the programs P^i, and P^i_+, denoted by $\vec{P^i}$, and $\vec{P^i_+}$ respectively, are compared using the parameterized Euclidean distance function:

$$D_W\left(\vec{P^i}, \vec{P^i_+}\right) = \left\|\vec{P^i} - \vec{P^i_+}\right\|_2 \tag{4.4}$$

The distance is then combined with the labels assigned to the pair of programs to generate the loss function:

$$L\left(W, Y, P^i, P^i_+\right) = (Y)\frac{1}{2}\left(D_W\right)^2 + (1-Y)\frac{1}{2}\left\{\max\left(0, m - D_W\right)\right\}^2 \tag{4.5}$$

where Y is the label representing the anchor program P^i's correctness, 0 if incorrect and 1 if correct, W represents the model weights, D_W is the parameterized Euclidean distance function, and m is the margin or radius.

The stochastic gradient approach is used to update the weights. In the inference phase, we normalize the distance between the anchor and correct programs using a sigmoid function to generate the model's predictions.

A key advantage of the Siamese network is that we are able to extend the limit on the input program length due to the fact that the programs are tokenized into separate sequences and fed to the model separately. Thus, in our case, with this architecture each program has 512 allocated token slots when encoded as token sequences.

4.4 Incremental Learning

In incremental learning, initially, our correct set contains only one solution for each problem. As we train the model using only that one solution as the anchor, we find other correct solutions within P^i. We then continue training the model now with an extended correct set including the newly found correct solutions to the problems. We repeat this process for k iterations (2-3) until the validation performance converges. The same process is used for both the training set and the validation set simultaneously.

4.4.1 Majority Vote Grading

As we have a trained model, we can predict the grade of a new unseen program $\widehat{P}^i$ using the set of correct solutions $\mathcal{P}^i_+$ more effectively. In order to take advantage of all

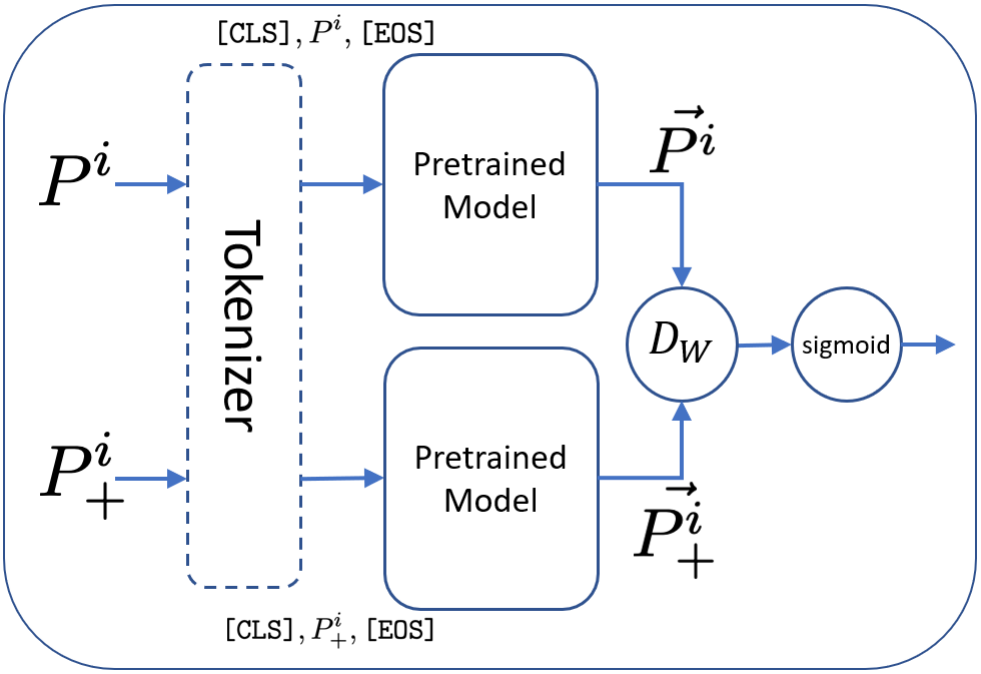


Figure 4.2: Deep Siamese Grader Architecture

A sample program P^i and a reference solution P^i_+ are tokenized using a pre-trained tokenizer (i.e., CodeBERT, UniXcoder). The tokenized programs are fed separately through two instances of the pre-trained model while sharing weights. The models' output is the `[CLS]` token embedding of each program denoted by $\vec{P}^i$ and $\vec{P}^i_+$. By applying the Euclidean distance function (D_W) on the vectors, we obtain a distance that is normalized using the sigmoid function. During training, the model does not make use of the sigmoid layer as the distance is passed directly into the contrastive loss function.

potential solutions in $\mathcal{P}^i_+$, we utilize a *majority voting* scheme. Algorithm 1 outlines the general procedure. The algorithm takes the trained model, the program to be evaluated for question i, denoted by $\widehat{P}^i$, and the set of correct programs for question i, denoted by $\mathcal{P}^i_+$, in order to predict the grade for $\widehat{P}^i$. While looping through each correct program $\widehat{P}^i_+ \in \mathcal{P}^i_+$, first, we pair the program $\widehat{P}^i$ with a correct program $\widehat{P}^i_+ \in \mathcal{P}^i_+$, then, tokenize the pair, and compute the label of $\widehat{P}^i$ using our trained model $\mathcal{M}$. Finally within the loop, we add the predicted label for $\widehat{P}^i$ to a set $\mathcal{L}$. Lastly, we compute the mode of

labels in $\mathcal{L}$ to obtain the final weak label for $\widehat{P}^i$ (this weak label will be used in the next section where we perform incremental grading). The $\widehat{P}^i$'s predicted grade is the average of grades predicted based on all $\widehat{P}^i_+ \in \mathcal{P}^i_+$. We also ensure that the current program is not paired with itself by being in the correct set ($\widehat{P}^i \neq \widehat{P}^i_+$).

Algorithm 1 Majority Vote Grading ($MajVote$)

Input: $\mathcal{M}$, trained model

Input: $\widehat{P}^i$, the program to be evaluated for question i

Input: $\mathcal{P}^i_+$, the set of correct programs for question i

1: $\mathcal{L} \leftarrow \{\}$

2: **for each** $\widehat{P}^i_+ \in \mathcal{P}^i_+$ **do**

3: $\quad seq \leftarrow Tokenizer(\widehat{P}^i, \widehat{P}^i_+)$

4: $\quad \widehat{y} \leftarrow \mathcal{M}(seq)$

5: $\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{\widehat{y}\}$

6: **end for**

7: $y_{pred} \leftarrow Mode(\mathcal{L})$ ▷ predicted label for $\widehat{P}^i$

8: $s_{pred} \leftarrow \underset{y \in \mathcal{L}}{Average}(prob(y))$

4.4.2 Incremental Transductive Grading

In section 4.4.1, we assumed that there is a set of correct solutions for each question i. However, in a more realistic setting, we have only one correct solution in $\mathcal{P}^i_+$. In order to address this issue, we design an incremental transduction grading scheme, taking advantage of other potential correct solutions in the set of programs to be evaluated. The procedure is demonstrated by Algorithm 2. While this approach assumes only one correct program initially in the correct set $\mathcal{P}^i_+$ (line 3), it expands this set with newly found correct solutions using the majority vote algorithm's predictions. The final correct set represents the predicted correct programs and the rest represents the predicted incorrect

programs. In particular, this algorithm requires a trained model, the set of programs to be evaluated for each question $i \in \mathcal{Q}$ (set of questions), and a single correct program P_+^i. For each question, we repeat the following process until the correct set for that specific question remains unchanged: First, for each program, we predict the program label based on the majority vote grading approach outlined in Algorithm 1 (line 5). If the program is predicted as a correct one (based on the predicted weak label), we add the program to our correct set $\mathcal{P}_+^i$ (line 7). We repeat the process until no new correct programs are found for the question. Similar to the previous algorithm, we do not pair up a program with itself when the program is also contained in the correct set for a question.

Algorithm 2 Incremental Transductive Grading

Input: $\mathcal{Q}$, the set of questions

Input: $\mathcal{M}$, trained model

Input: $\mathcal{P}^i$, the set of programs to be evaluated for question i

Input: P_+^i, a correct program for each question $i \in \mathcal{Q}$

1: **for each** $i \in \mathcal{Q}$ **do**
2: $\quad \mathcal{P}_+^i \leftarrow \{P_+^i\}$
3: $\quad$ **repeat**
4: $\quad\quad$ **for each** $p \in \mathcal{P}^i$ **do**
5: $\quad\quad\quad (y_{pred}^i, s_{pred}^i) \leftarrow MajVote(\mathcal{M}, p, \mathcal{P}_+^i)$
6: $\quad\quad\quad$ **if** $y_{pred} = 1$ **then** ▷ correct label
7: $\quad\quad\quad\quad \mathcal{P}_+^i \leftarrow \mathcal{P}_+^i \cup p$
8: $\quad\quad\quad$ **end if**
9: $\quad\quad$ **end for**
10: $\quad$ **until** $\mathcal{P}_+^i$ is unchanged
11: **end for**

4.5 Interpretability / Explainability

Large neural network language models have achieved state-of-the-art results on many NLP tasks [13, 23, 28, 73], however, they lack interpretability and explainability of their decisions involved in many different tasks [69]. This caused people to question the trustworthiness of these models' predictions especially when considering high risk environments [21]. In the field of Explainable Artificial Intelligence (XAI), researchers are searching for ways to explain black-box neural network models such like ours [65, 71, 77]. Thus, we decided to use attention analysis and LIME methods to interpret our deep language models and to provide feedback for programs.

4.5.1 Attention Analysis

The most recent state-of-the-art language models have one important commonality. Most, if not all, share the same Transformer [68] architecture revolving around the attention mechanism [7]. With attention the model is able to assign weights to each feature representing the importance of the feature for the prediction. For example, when predicting whether an image contains a cat or a dog, a convolutional neural network will pay more attention to some specific parts of the image such as the nose of the animal. Similarly, in the next-token prediction task in NLP, a language model will have higher attention weights on tokens that are most important in predicting the next token (e.g., the subject of the sentence). Researchers have discovered the possibility of analyzing the attention layers to explain the intentions of large language models such as BERT [15]. These large models' architecture contain multiple attention heads and layers which exhibit different patterns that can be analyzed. For example, we can deduce based on a given textual input to what tokens or part of the sentence does the model pay most attention to. These patterns reveal that certain attention heads correspond to syntactical linguistic notions. In natural language, certain heads attend to determinants, others to

nouns or verbs. While in programming language, the attention is more prominent to "for" and "while" loops in some heads or variables in others.

When extracting the self-attention matrix learned by the language model for an input sequence, we can observe how certain tokens attend to other tokens in the sequence. The attention matrix is a four-dimensional matrix where the dimensions are the number of heads, followed by the number of layers for each head, the number of tokens that attend, and the number of tokens that receive attention. We can then aggregate over different dimensions in order to obtain distinct visualizations for which we can derive corresponding interpretations.

4.5.2 LIME

Local Interpretable Model-agnostic Explanations (LIME) [58] is a method used to explain predictions of large black-box models by presenting the features that had the most influence on the prediction. Given an input to the model, LIME creates random perturbations to the input sequence and constructs a locally linear model using the ground-truth predictions obtained from the distinct perturbations. Following LIME, other perturbation methods were investigated on sequence-to-sequence models in natural language processing by providing causal relational explanations of token groups [6].

In our case, LIME will provide us with the specific programming keywords that mostly influenced the decisions of our deep language models. It will create perturbations of the evaluated programs while leaving the reference solution intact. It will only use the reference solution as part of the input to the model in order to make predictions.

4.6 Feature Engineering Approaches

In order to show the effectiveness of the proposed deep learning models, we develop some baselines based on extensive feature engineering approaches. Our baseline models are

inspired by the model proposed in [62]. The pipeline includes *feature extraction*, *feature invariant transformation*, and a *shallow prediction model*. Similar to our deep learning approaches, we train the models using a set of correct and incorrect solutions, but unlike our approaches, we utilize more than one correct solution for each question to evaluate a program's correctness.

Figure 4.3 presents the pipeline of our baseline model for a specific question. This model requires a large amount of correct responses to perform well since it utilizes all correct responses to compare and judge the correctness of a program. Hence, it aims to predict the correctness of a response to a new question given a large enough correct response set. This is quite an inconvenience since developers of new programming questions do not have a large array of correct responses to the question. They usually write a solution and grade the responses based on that. Producing multiple solutions is very expensive in terms of time and labour.

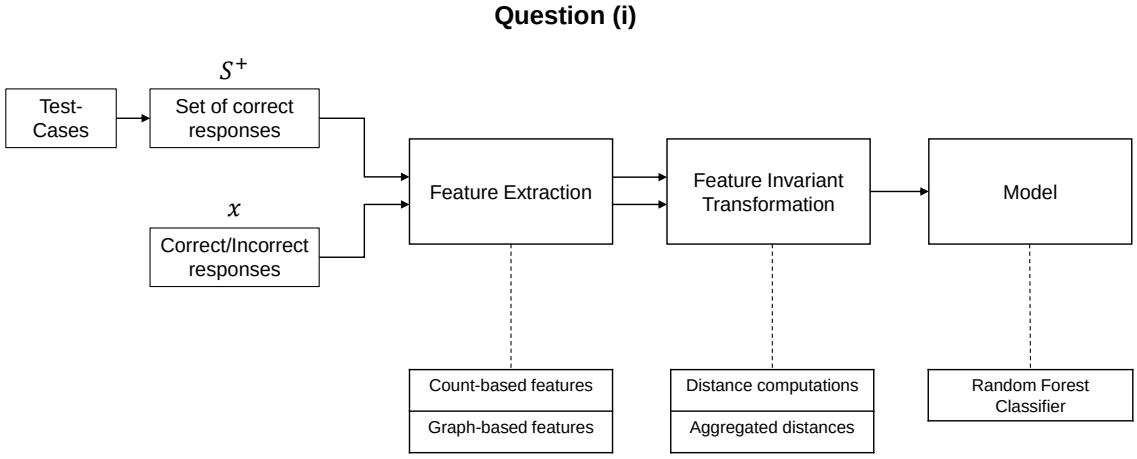


Figure 4.3: Feature-based Model Pipeline

Test-cases allow us to weakly label the responses in order to create a set of positive (correct) responses to a question. We can then extract lexical count-based features and semantic graph-based features from the programs. For each response, we compute distances between the given response (x) and each positive response (s^+) for a specific question, and aggregate the computed distances using the mean of the 10% minimum

distances. Finally, we input these vectors to the classifier and the model outputs the label coinciding with the given response's correctness.

4.6.1 Feature Extraction

This section presents a set of features that can be extracted from the programming solutions. It is important to extract both *lexical* and *semantic* features from the programs. Thus, we focus on features capturing both aspects of the programs using *count-based* and *graph-based* features. The features emphasize key tokens, the overall structure, and data dependency between both statements and expressions, which are the essential properties of a computer program.

Count-based features: We first define different lexical count-based features separated into categories. Examples of keywords that are considered can be seen in Table 4.2. The keywords adhere to the grammar of the specific programming language being used. We have found that similar features exist in both C++, Python, and other programming languages.

Graph-based features: In some cases, incorrect programs can have similar keyword counts to correct programs, however, they could be structurally different. In order to identify the structure of the program and compare the programs based on their structures, we need graph-based features. As such, we define semantic features extracted from graphs representing the programs' structure. We define a *control-flow* graph, *statement data dependency* graph, and *expression data dependency* graph as well as the *Abstract Syntax Tree (AST)* graph.

The control-flow graph (see Figure 4.4) presents the overall structure of the program showing which statements are inside others. Nodes represent statements and edges represent containment of statements in the code. For example, if a "for" loop code statement contains a variable declaration statement, then a directed edge would exist from the loop statement to the variable declaration statement in a control-flow graph.

Table 4.2: Count-based feature categories

Categories	**Examples**
Basic	alphabet characters, numeric characters, ...
Loops	'for', 'while', ...
Data types	'int', 'float', ...
Operator	addition, subtraction, ...
Punctuation	semicolons, commas, ...
Bracket	curly, round, square, ...
Objects	'new', 'class', ...
Access modifier	'public', 'private', ...
Exception	'try', 'catch', ...
Conditional	'if', 'else', ...
Function	function declarations, function calls, ...
Input and output	'cin', 'cout', ...
Advanced	maximum integer, maximum array dimension, ...
Other	'using', 'include', 'define', ...

The statement data dependency graph (see Figure 4.5) presents the data flow between statements containing variables. Nodes represent statements containing variables and edges represent the supply of data. For example, a variable initialization statement will have a directed edge to all the statements that utilize that variable in some way.

The expression data dependency graph (see Figure 4.6) presents the data flow between variable expressions. Nodes represent expressions and edges represent the supply of data. Although similar to the previous graph representation, this graph breaks up the statements into individual expressions and allow a more detailed representation of the data flow in a program.

The abstract syntax tree (see Figure 4.7) is a complex graph representation of source code widely used for parsing and program analysis [74, 79, 83, 85]. The nodes are written in a formal language similar to a syntactic tree in natural languages. The leaf nodes contain the actual code keywords and their parents contain different abstract constructs. Due to the amount and complexity of existing distinct constructs, an abstract syntax tree representation of a program is often considerably larger than the previous graph representations. The tree can be also be represented as a graph with directed edges from the root of tree to the leaf nodes. Figure 4.8 shows an example of the full abstract tree representation of a medium sized program. As can be seen in the figure, these representations are quite complex and large in size.

In a computer program, a statement is defined as a fragment of code that carries out a specific task. While an expression is composed of an operator and its corresponding operand which forms a computation.

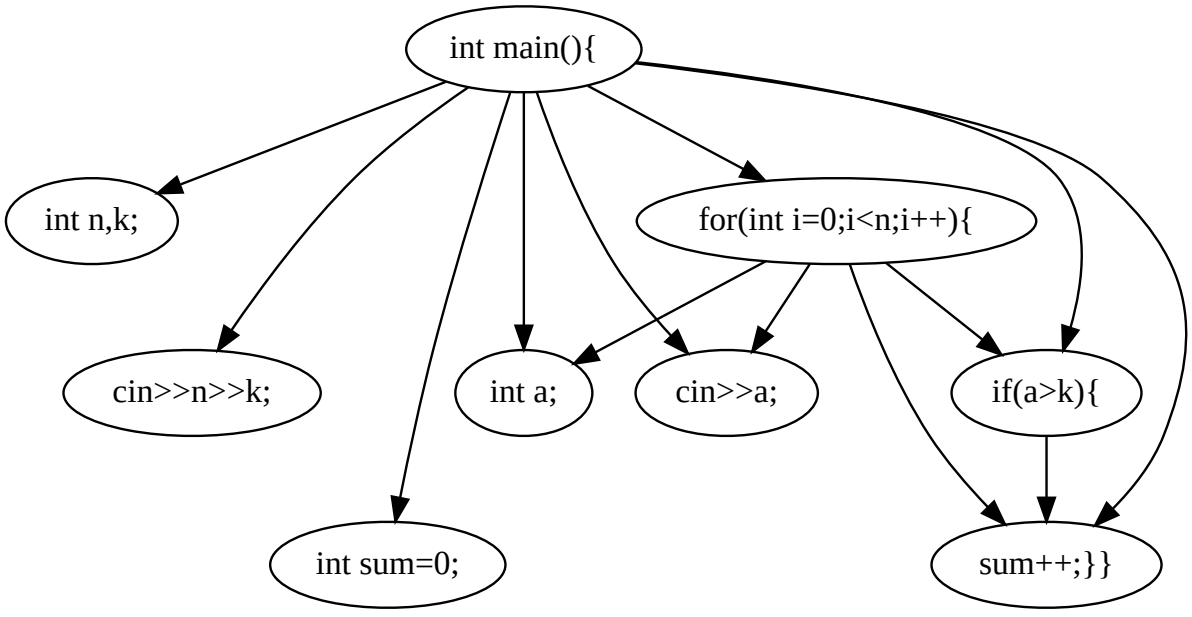


Figure 4.4: Control-flow graph

As we have these graphs, we extract different features from them, such as mini-

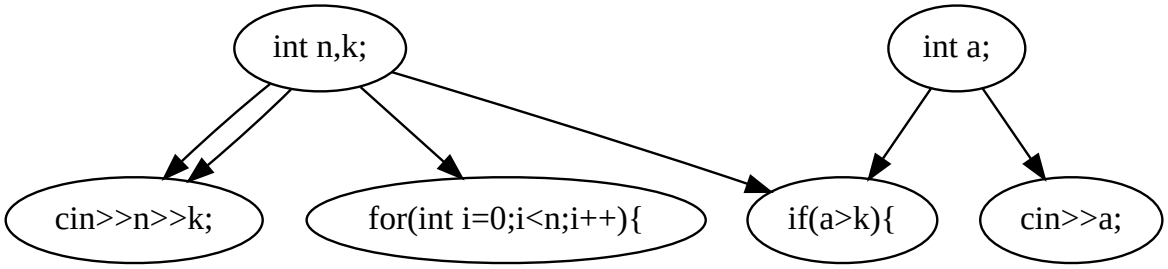


Figure 4.5: Statement data dependency graph

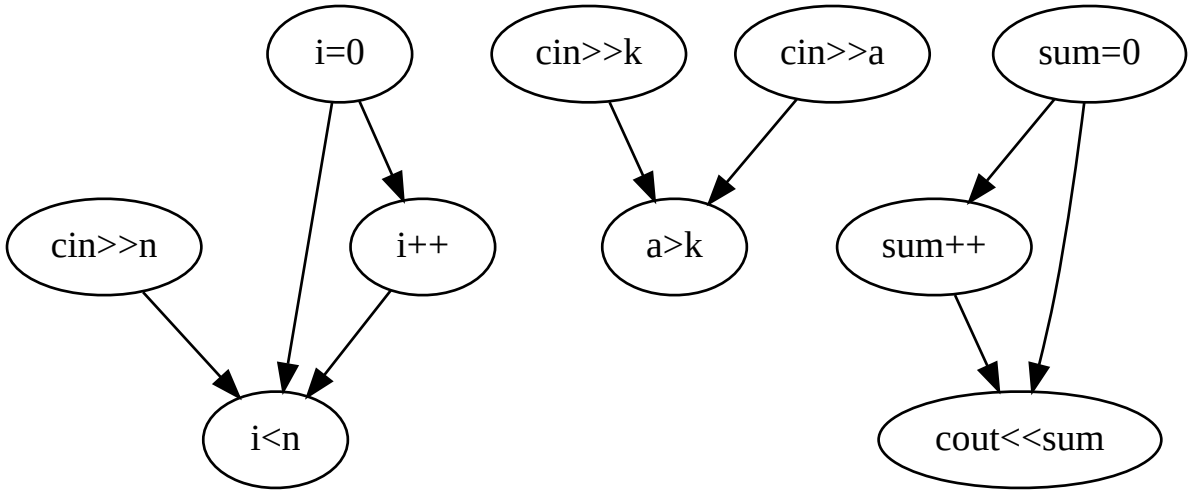


Figure 4.6: Expression data dependency graph

mums, maximums, means of degrees, centrality, closeness, and clustering coefficients in the graph. We also consider the number of edges, the density, and the transitivity of the graphs as features. Compared to our approach, in [62], they simply maintain counts of specific keywords following each other in different graph representations. We wanted to extend this part of the work by also providing the models with multiple properties that can be extracted from the graphs. For example, the degree centrality of nodes in a graph encode the fraction of nodes each node is connected to. The closeness centrality is the reciprocal of the average shortest path distances from all nodes connected to each node [24]. The clustering coefficients of each node is the fraction of triangles that pass through that node. The density of a graph is a ratio between the number of directed edges and

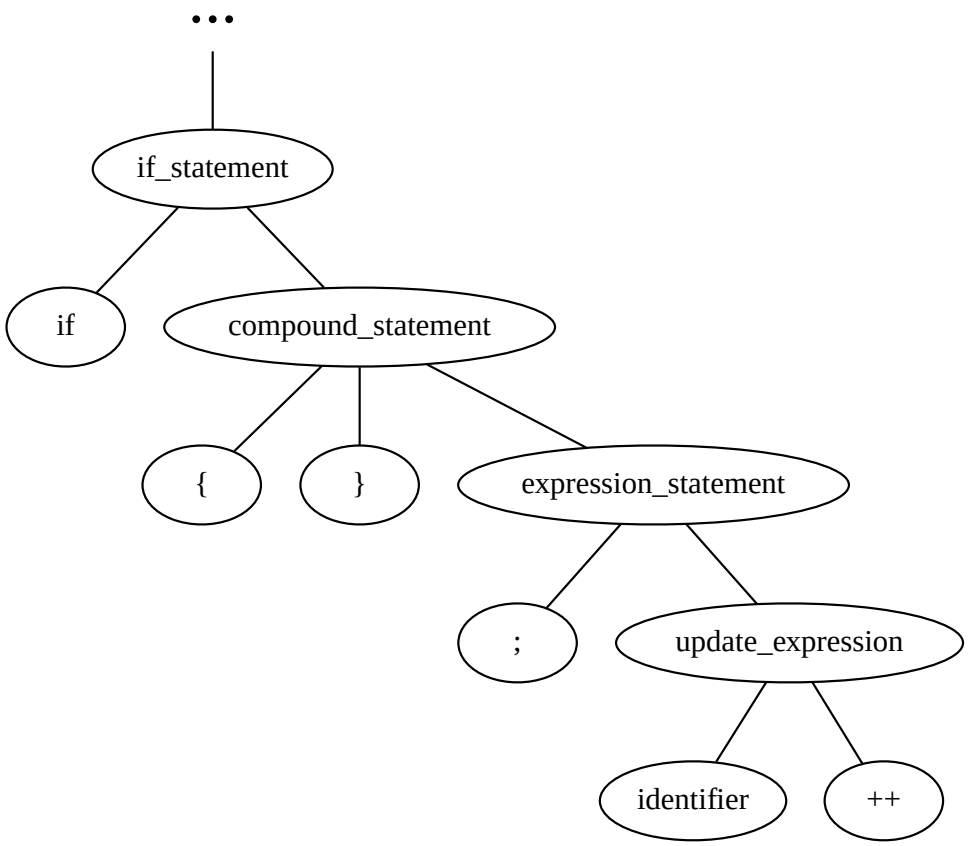


Figure 4.7: Abstract Syntax Tree

the maximum number of edges that can be in the graph. The transitivity encodes the fraction of all triangles over all possible triangles (triads) in the graph.

4.6.2 Feature Invariant Transformation

In order to create a question-independent model, we followed the feature invariant transformation approach introduced in [62]. The idea is that the input to the model of a correct response for a question has to be similar to the input of a correct solution for a different question. The feature invariant transformation implies that we compute the distances between a solution (represented by a vector of feature) and the set of correct solutions for a specific question. We then take an aggregate of the distances and use that

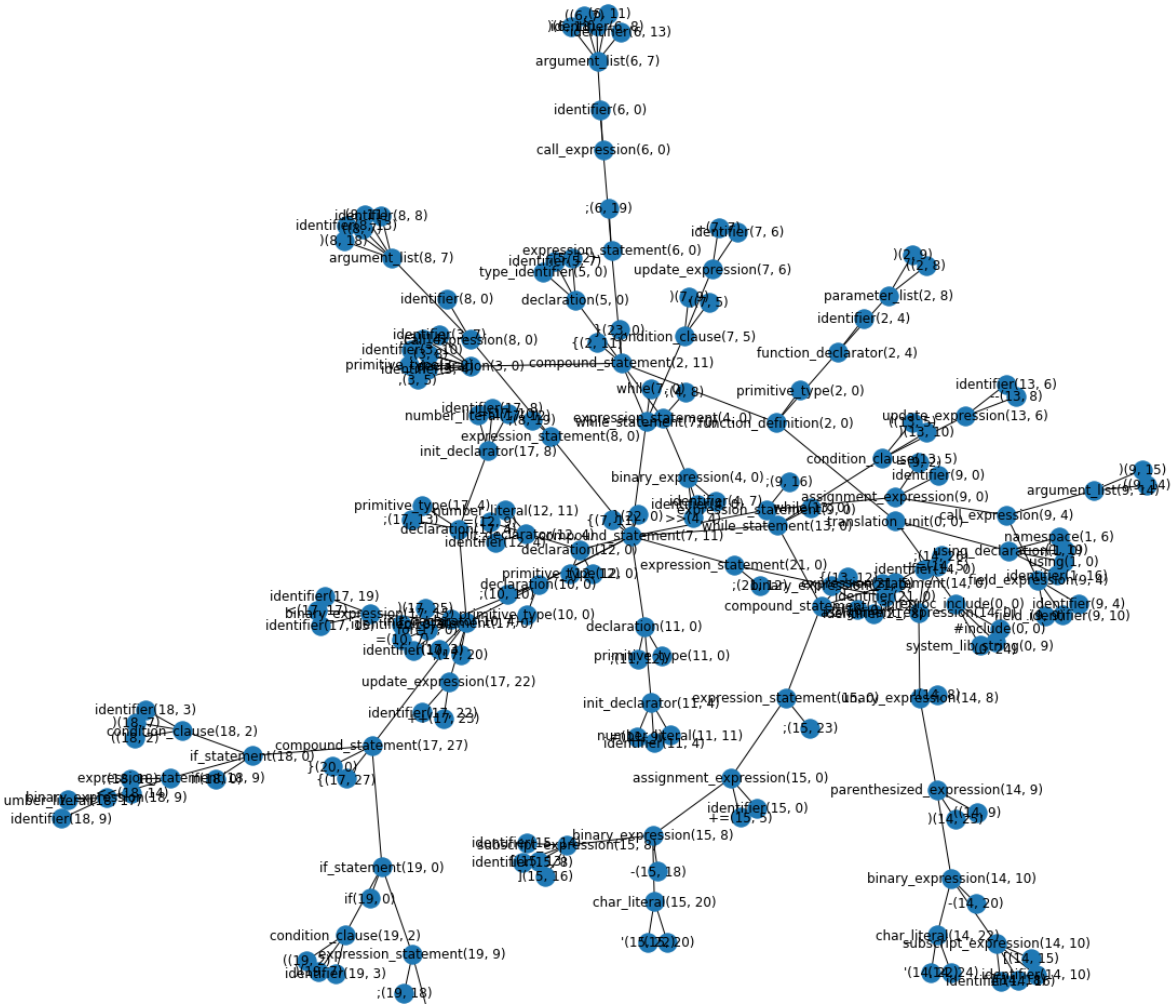


Figure 4.8: Complete Abstract Syntax Tree

as the predictive model input for that solution. The intuition behind this is that two correct programs should be similar and return a low distance while the distance between an incorrect and a correct program should be higher. In the following definition, we assume that we have multiple correct solutions for each problem. However, in the case that we only have a single correct solution, we simply compute a single distance vector and use that as the input to our model as aggregation becomes unnecessary.

$$D^+ = \Psi(\widehat{P}^i, \widehat{P}^i_+) \qquad \forall\ \widehat{P}^i_+ \in \mathcal{P}^i_+ \tag{4.6}$$

$$\bar{x} = Mean(Min10(D^+)) \tag{4.7}$$

Ψ: Distance function

$Min10$: Minimum 10%

In the approach from [62], they took the average of the top 25% of closest solution vectors. However, we found that the average of the top 10% of closest solution vectors will result in better performance (see Figure 5.8). Thus, we set this hyper-parameter to 10 when using multiple correct solutions per problem.

Multiple distance functions exist that compute distances between two 1-Dimensional vectors. For the empirical results, we used the Euclidean distance or L_2 norm to be consistent with the previous research paper [62].

$$\Psi(u, v) = ||u - v||_2 = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} \tag{4.8}$$

where u and v are arbitrary input vectors to the distance function. Additionally, the following distance functions between two vectors (u, v) were examined:

Canberra

$$\sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|} \tag{4.9}$$

Chebyshev

$$\max_i |u_i - v_i| \tag{4.10}$$

Cityblock

$$\sum_i |u_i - v_i| \tag{4.11}$$

Bray-Curtis

$$\frac{\sum |u_i - v_i|}{\sum |u_i + v_i|} \tag{4.12}$$

Correlation

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2} \tag{4.13}$$

Cosine

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2} \tag{4.14}$$

4.6.3 Shallow Prediction Models

We tested many shallow models in our experiments and report the results of the best performing models as the baselines. In particular, we utilize linear regression with both L_1 regularization (LASSO) and L_2 regularization (Ridge regression), Random Forest, and support vector machine (SVM) for training and evaluation purposes.

The Random Forest model's default parameters use 100 decisions trees as estimators, with the Gini impurity metric to measure the quality of the splits, and the number of features to consider when looking for the best split is set to the square root of the number of features the input has. By tweaking the default parameters, using 1000 estimators, with entropy metric, and using all the features when determining the best split, we achieved a superior performance. We tweaked some of the default parameters for LASSO, mainly, we set the regularization strength to 1.0, the max iterations to 1E9, and the optimization tolerance to 5E-5. For the ridge regression model, which uses the linear least squares function as its loss function, we used similar parameters to LASSO. The support vector machine adjusted for binary classification uses 1.0 for the regularization parameter, the radial basis function kernel, and a optimization tolerance of 5E-5.

Since the scale of the distance vectors can vary over questions we scaled the data over features before inputting it into the model. We also attempted normalizing the input by the average distance between the responses and the correct set.

Chapter 5

Experiments and Results

5.1 Data Collection and Preprocessing

The data was retrieved from an open-access coding website called Codeforces that hosts competitive programming contests. A total of 200K Python programs and 200K C++ programs were retrieved from 1000 programming problems. The data was then cleaned to remove duplicate programs and problems with missing submissions, ending up with a total of 190K Python submissions from 985 questions and 168K C++ submissions from 992 questions. The website also provided labels (correct/incorrect) for each submission based on question specific test-cases created by the authors of the questions. A balance of correct and incorrect submissions was maintained during the retrieval process.

The collected dataset was then pre-processed. First, each program was ensured to be compilable using the `tree-sitter`[1] parsing tool. The code was then minimized and standardized using a custom modified version of the `pyminifier`[2] tool. The programs were stripped from blank lines, indentation was standardized, multi-line code was minimized to a single line, comments and doc-strings were removed and finally, unnecessary space characters around operators and other keywords were removed. Programs that still

[1] `https://github.com/tree-sitter/tree-sitter`
[2] `https://github.com/liftoff/pyminifier`

did not compile because of white-space inconsistencies or other formatting errors which could not be automatically fixed were manually adjusted. In our proposed deep neural network models the maximum sequence length is bounded by the available memory and the input batch size. This poses a problem for long programs as their token vector gets truncated and lose part of their solution to the problem. Hence, the models lose the ability to accurately assess their correctness. To address this issue, we removed problems consisting majorly of long programs or requiring a substantial amount of code to solve the problem.

The final dataset[3] includes problems ranging from a beginner's programming level requiring few simple control structures to an expert level including functions, recursion, multiple interlaced loops, many subtle conditional cases and high dimensional array manipulations. The statistics of the data are presented in Table 5.1.

	Python	C^{++}
No. of programs	190,451	168,110
No. of correct programs	96,934	82,430
No of incorrect programs	93,517	85,680
No. of questions	985	992
Avg. No. of programs/question	193.4	169.5
Avg. No. of correct prog./question	98.4	83.1
Avg. No. of incorrect prog./question	95.0	86.4

Table 5.1: Dataset Statistics

When observing our test set, we find various interesting properties related to our collected data. We computed histograms (see Figure 5.1) comparing the correct and

[3]Code and data are available at `https://github.com/peter-nagy1/Deep-Grader`

incorrect programs based on their distance level using our feature-based algorithms. Their distance level is computed simply by aggregating their distance vectors.

We observe that there is a majority of correct programs at low distance levels and a majority of incorrect programs at high distance levels. This validates our previously mentioned goal in the feature invariant transformation section, where we define a transformation of our features with the goal of producing this specific property in the data.

Additionally, these histograms are left-skewed which means that the data is more concentrated at low distance levels. This suggests that most programs are somewhat similar to their related set of correct programs in our data.

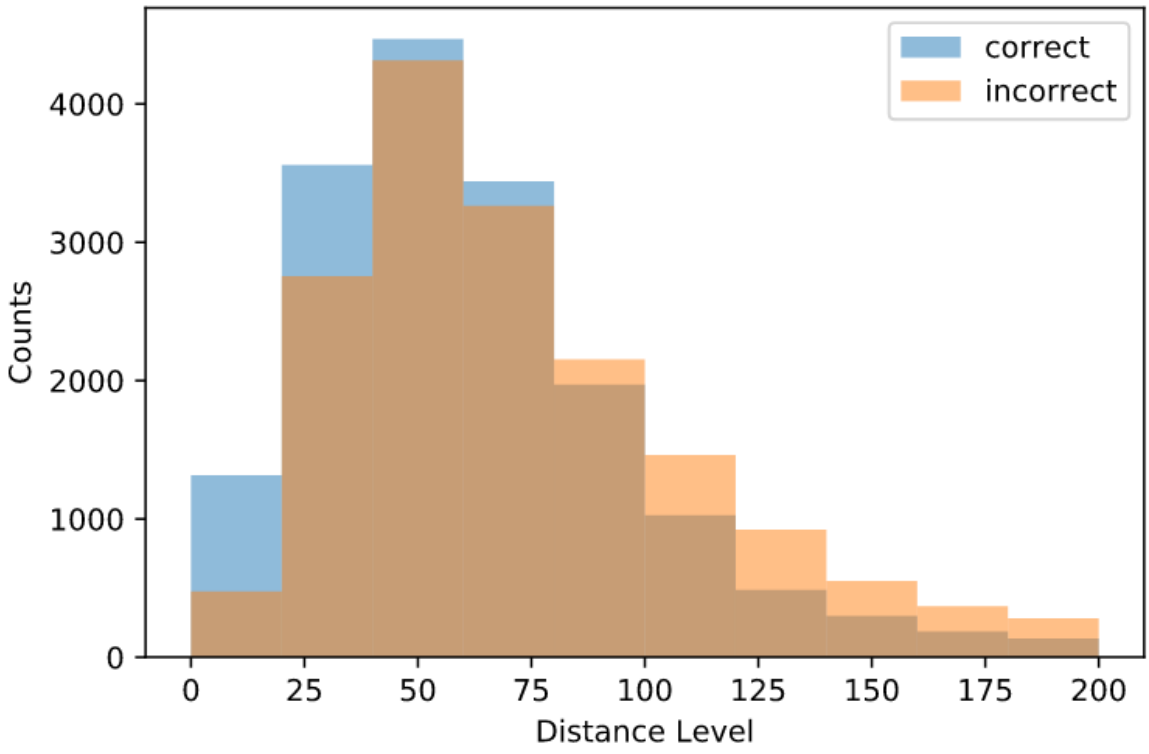


Figure 5.1: Histograms presenting the distance levels in features for correct and incorrect programs used for evaluation.

Finally, when plotting a pair of graph representation features from instances of our test data (see Figure 5.2), we observe many outlier points. The same observation can be made

with different feature pairs. These outliers indicate potential mistakes in the problems' test-suite. The removal of such outliers could potentially improve the performance and the robustness of the models.

With some further investigation, we have found that some programs which are correct are labelled incorrect by the source website due to the test-suite failing for unknown reasons. When re-executing the test-suite on such programs manually, we find that the programs pass the test-cases without a problem. The time and memory usage during the execution of programs are captured. When programs do not complete the test-suite due to time or memory limits, the data source clearly indicates this and we do not consider these programs as their correctness is unknown. Thus, these constraints cannot be the reason for errors in the data. We also disregard programs that fail the test-suite due to compilation errors.

5.2 Baselines and Experimental Setup

We compare the proposed models with four shallow models (i.e., Random Forest, SVM, LASSO, and Ridge). The feature engineering pipeline is based on the previous state-of-the-art method [62] built upon the features discussed in section 4.6. The set of programs is split along questions. In particular, 80% of questions will appear in the training set, 10% will appear in the validation set and the responses from the remaining 10% of questions will be used in our evaluation set. We optimize all hyper-parameters using the validation set. We run the experiments 3 times and report the performance of models, namely, Accuracy, F1-measure (F1) in predicting weak labels, and the Correlation, Mean Squared Error (MSE) between the model's predicted scores and the marks given by experts. We present the results for both Python and C++ datasets.

$$F1 = 2 \cdot \frac{(precision \cdot recall)}{(precision + recall)} \tag{5.1}$$

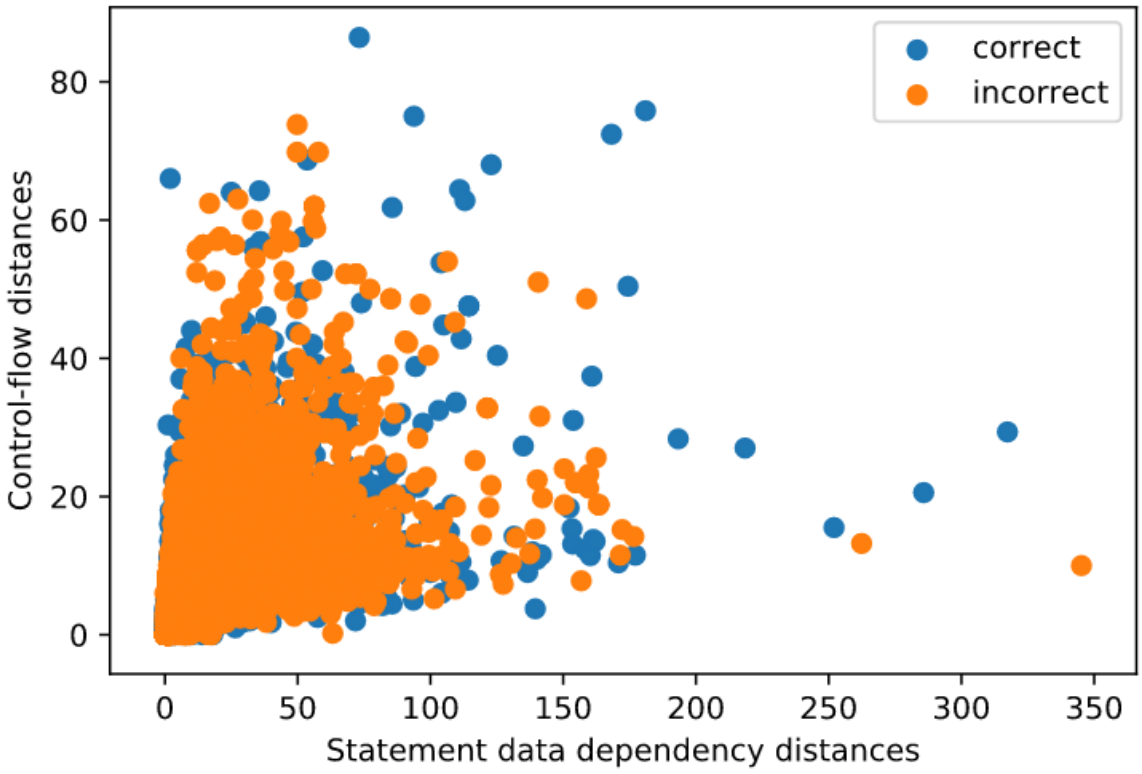


Figure 5.2: Pair of graph representation feature distances for correct and incorrect programs used for evaluation.

5.2.1 Feature Engineering

Along experimentation performed on our test dataset, we found that with the inclusion of graph-based features, the model was able to correctly label 5.5% of our test data which was initially mislabeled by the model trained solely on count-based features.

In the following examples (see Figure 5.3) of two programs retrieved from our dataset, the incorrect solution does not account for a specific input case. Although the count-based features alone do not detect this program as incorrect, graph-based features help the model notice the structural differences and data dependency differences between this program and the correct program. Merging the two conditions present in the if statements with an "&&" operator can make this program correct.

Also, note that 68% of the correct submissions gathered for this problem use at least

Correct

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

int main()
{
  int n ; cin>>n;
  string s ; cin>>s;
  if(s[0]=='S' && s[n-1]=='F')
    cout<<"YES";
  else
    cout<<"NO";

  return 0;
}
```

Incorrect

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

int main()
{
  int n;
  string b;
  string a;
  cin >> n;
  cin >> b;
  a.append(b);
  if (a[0] != a[n-1]) {
    if (a[0]=='S')
      cout << "YES" << endl;
  }
  else {
       cout << "NO" << endl;
    }
  }
```

Figure 5.3: Importance of graph-based features.

In terms of count-based features, the programs are similar, however, in terms of graph-based features, the programs show a distinct structure.

one loop to solve this problem. Despite the fact that this incorrect submission does not have any loops it does not produce a large feature distance for the loop category. It accurately assimilates with other correct submissions that do not use loops.

5.2.2 Feature Importance

We provide an impurity-based feature importance graph showing the importance of each feature category in Figure 5.4. Also known as the Gini importance, the importance values represent the total reduction of the criterion brought by that feature. We observe that most of our features contribute similarly to the prediction of the models with the exception of the object, access-modifier, and exception category features which show close to zero Gini importance. Our basic features which encode the underlying program's statistics obtain the highest importance.

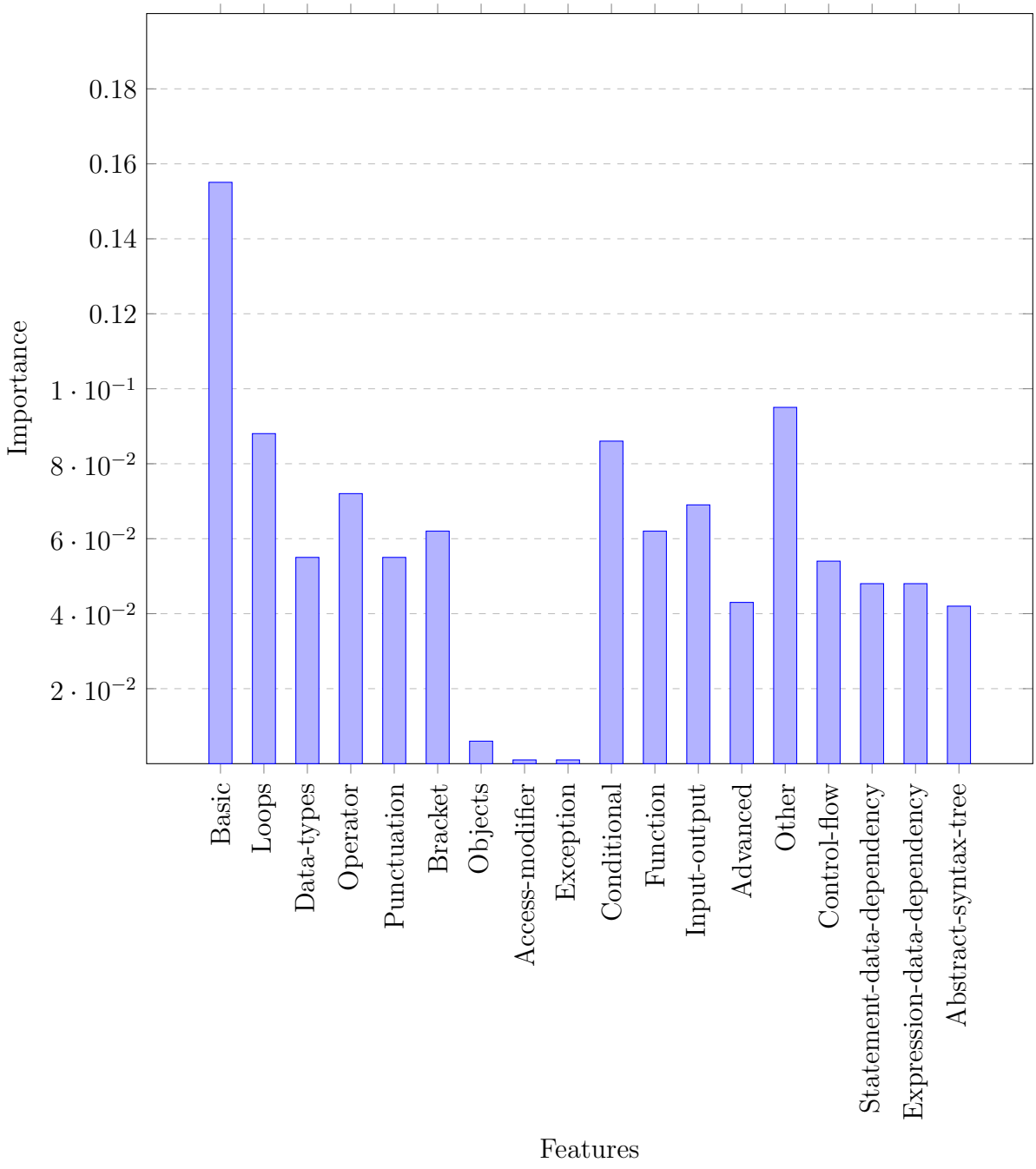


Figure 5.4: Impurity-based feature importance

We investigate the importance of different graph-based features using correct and incorrect example programs and find interesting results in Figures 5.5 and 5.6.

<table>
<tr><th>Correct</th><th>Incorrect</th></tr>
<tr><td colspan="2">Control-Flow Difference</td></tr>
<tr><td>

```
#include<iostream>
using namespace std;
int main()
{
    int n;
    cin>>n;
    int a[n];
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    for(int i=0;i<n;i++)
    {
        cout<<a[i];
    }
}
```

</td><td>

```
#include<iostream>
using namespace std;
int main()
{
    int n;
    cin>>n;
    int a[n];
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
        for(int i=0;i<n;i++)
        {
            cout<<a[i];
        }
    }
}
```

</td></tr>
</table>

Figure 5.5: Comparing examples of correct and incorrect programs using control-flow. Closing the bracket of the first for loop after the second for loop would render this program incorrect. The count-based features remain nearly identical even though the two programs are structurally different. The incorrect program has a nested for loop while the correct program does not.

5.2.3 Distance Functions

We investigated multiple distance functions and arrived to the conclusion that the Chebyshev distance function produced the best results (see Table 5.7). Furthermore, we explored using different percentage values (ρ) to aggregate the computed distance vectors

Correct	Incorrect
Data Dependency Difference	

`#include<iostream>`
`using namespace std;`
`int main()`
`{`
`  int n,m;`
`  cin>>n>>m;`
`  n = n * 2;`
`  m = m * 3;`
`  cout<<n+m;`
`}`
	`#include<iostream>`
	`using namespace std;`
	`int main()`
	`{`
	`  int n,m;`
	`  cin>>n>>m;`
	`  n = n * 2;`
	`  m = n * 3;`
	`  cout<<n+m;`
	`}`

Figure 5.6: Comparing examples of correct and incorrect programs using data dependency.

Replacing the assignment "m = m * 3" by "m = n * 3" will render the program incorrect. The count-based features remain nearly identical even though the two programs have different data dependencies. In the incorrect program, the value of the variable "m" depends on the value of the variable "n" while this dependency is not present in the correct program.

using the average of a fraction of the top closest solution vectors. From Figure 5.8, we arrived to the conclusion that the top 10% of closest solution vectors returns the best performance.

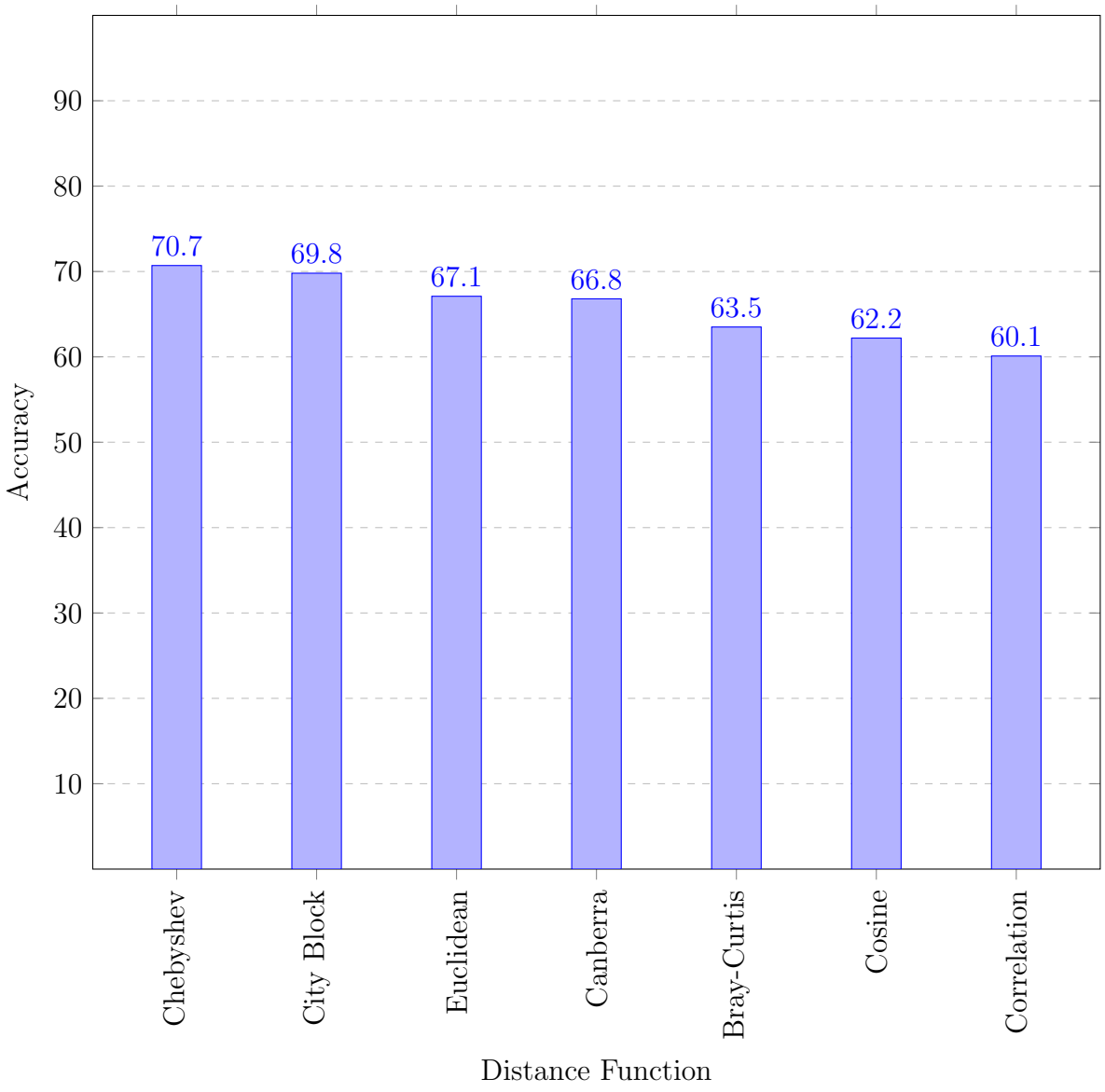


Figure 5.7: Accuracy for each distance function (evaluated with Random Forest and *Python* code)

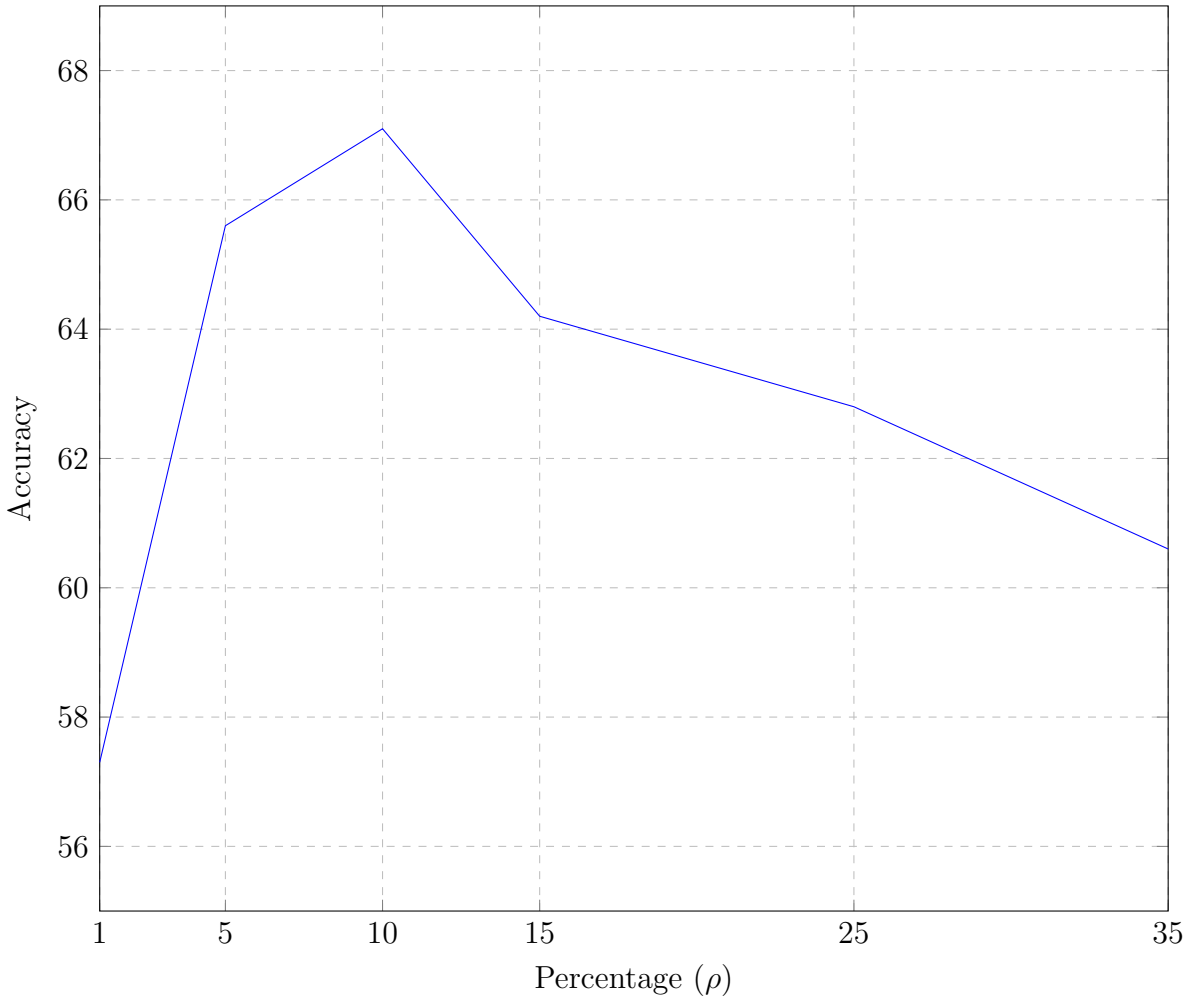


Figure 5.8: Percentage of minimum distances (evaluated with Random Forest and *Python* code)

5.3 Proposed Models

Tables 5.2, 5.3 show the performance of the proposed models as well as the baselines where we use only one correct solution to evaluate an unknown solution. As can be seen, the proposed models outperform the baseline models consistently by a large margin for both Python and C++ datasets. We also see that in most cases Deep Grader outperforms Deep Siamese Grader especially when evaluated on Python data. Moreover, the CodeBERT pre-trained model performs better for both Deep Grader and Deep Siamese Grader in terms of F1 measure and Correlation. As another observation, although the pre-trained models (CodeBERT, UniXcoder) were never trained on C++ code, the results show similar performance for both programming languages. This proves that the pre-trained models are highly flexible for this task and the pre-trained model trained based on one language can be fine-tuned and used in our framework for grading the solutions of a question in another language. The reason we see slightly higher results from the models on C++ data overall can be due to the distinct collection of problems and distribution of programs used in the data.

Although the Deep Grader models always perform in superiority over all metrics compared to the baseline models, the Deep Siamese Grader models do not reach high results on the Python dataset. The decreased performance from the Siamese models may be due to the contrastive learning method employed showing its downside in capturing the embedding distance with this specific programming language.

5.3.1 The Effect of Majority Vote Grading

Tables 5.4, 5.5 illustrate the performance of the proposed models as well as the baselines where we use multiple correct solutions (in our Python dataset we have on average 98.4 correct solutions for each question) to evaluate unknown solutions for a question. Similar to Tables 5.2, 5.3, the proposed models perform much better than the baselines

consistently in terms of all performance measures for both Python and C++ datasets. Furthermore, in most cases, we observe that for this setting Deep Grader is superior to

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	62.3	63.7	51.0	2.18
SVM	63.9	68.7	61.0	1.82
LASSO (L1)	61.5	70.1	48.7	2.03
Ridge (L2)	61.3	68.9	54.3	1.89
Deep Grader (CodeBERT)	**74.5**	**74.4**	**77.2**	1.24
Deep Grader (UniXcoder)	73.9	71.9	77.1	**1.03**
Deep Siamese Grader (CodeBERT)	67.9	71.5	71.9	1.12
Deep Siamese Grader (UniXcoder)	67.8	71.7	73.8	1.04

Table 5.2: Performance of models with single correct solution on *Python* code

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	64.4	61.8	55.7	1.61
SVM	65.2	66.0	55.8	1.49
LASSO (L1)	63.9	65.2	51.3	**1.47**
Ridge (L2)	65.2	66.0	51.0	1.49
Deep Grader (CodeBERT)	75.8	76.4	75.6	2.87
Deep Grader (UniXcoder)	**76.6**	76.8	75.2	2.90
Deep Siamese Grader (CodeBERT)	74.4	**77.1**	**75.7**	**1.47**
Deep Siamese Grader (UniXcoder)	73.6	76.1	75.4	1.86

Table 5.3: Performance of models with single correct solution on C^{++} code

Deep Siamese Grader in terms of both datasets. It is worth mentioning that the baseline approaches based on the proposed framework in [62] feed the average of distances of all feature vectors of correct solutions to the feature vector of an unknown solution into the models and predict the unknown solution's label/score accordingly. While in the proposed majority vote approach, we predict the label for each pair of unknown and correct solution individually and then consider the majority of the predicted labels/average of the scores as the final label/score.

When comparing the results of the models exploiting a single correct pairing presented in Tables 5.2, 5.3 with Majority Vote Grading in Tables 5.4, 5.5, we observe that in most cases Majority Vote Grading helps improve the grading capability of our models. The reason is that the Majority Vote Grading uses the full correct set containing all correct responses for a question i for the input $\mathcal{P}_{+}^{i}$ when computing the majority vote of the test programs.

In Tables 5.4, 5.5, there is a difference between the way we evaluate the baselines and our proposed models using the full correct set. The main difference between the baseline method's standard evaluation and its majority vote evaluation is the way in which the distances get forwarded to the model. The standard evaluation computes the distances between the program pairs and aggregates them creating one aggregated distance which is then fed to the model to obtain the label. The majority vote based evaluation differs in that each distance computed is individually fed to the model producing multiple labels which are then aggregated using the mode function to obtain one final label.

5.3.2 The Incremental Transductive Learning Analysis

Starting from a single solution, we can expand the correct set for each problem and utilize Majority Vote Grading to evaluate programs using multiple correct solutions. From Tables 5.6, 5.7, we can discern that Incremental Transductive Grading helps improve the results in most cases. For example, Deep Grader (UniXcoder) accuracy (Acc) is 73.9%

and 76.6% when using a single correct solution and multiple correct solutions to evaluate an unknown solution for the Python dataset accordingly. In this case, incremental

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	67.1	70.3	70.2	1.75
SVM	65.7	71.4	71.9	1.63
LASSO (L1)	62.0	70.7	63.8	1.98
Ridge (L2)	62.8	70.6	69.8	1.87
Deep Grader (CodeBERT)	**77.0**	**77.3**	77.3	1.08
Deep Grader (UniXcoder)	76.6	75.2	**79.8**	**0.86**
Deep Siamese Grader (CodeBERT)	70.9	66.6	75.2	1.01
Deep Siamese Grader (UniXcoder)	70.3	65.0	74.7	0.97

Table 5.4: Performance of models with multiple correct solutions on *Python* code

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	69.2	67.5	53.6	1.66
SVM	68.6	70.0	55.8	1.47
LASSO (L1)	66.5	68.1	51.7	1.46
Ridge (L2)	68.1	69.8	49.3	**1.44**
Deep Grader (CodeBERT)	77.1	78.4	**76.8**	2.77
Deep Grader (UniXcoder)	78.2	**79.0**	76.2	2.88
Deep Siamese Grader (CodeBERT)	**80.0**	78.1	76.4	2.14
Deep Siamese Grader (UniXcoder)	77.4	75.9	75.3	1.85

Table 5.5: Performance of models with multiple correct solutions on C^{++} code

learning can achieve 75.7% accuracy which is an increase of 1.8pp when 2.7pp (76.6 - 73.9) is the room for improvement by using all correct solutions in the evaluation of an unknown solution. Furthermore, we observe that our Incremental Transductive Grading results mostly tend to approach the Majority Vote Grading results.

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	64.6	69.5	63.7	1.81
SVM	63.3	70.1	66.8	1.72
LASSO (L1)	60.0	69.9	58.0	2.05
Ridge (L2)	59.3	68.8	65.0	1.95
Deep Grader (CodeBERT)	75.0	**75.1**	75.7	1.15
Deep Grader (UniXcoder)	**75.7**	73.9	**78.1**	**0.97**
Deep Siamese Grader (CodeBERT)	72.0	69.8	75.9	1.00
Deep Siamese Grader (UniXcoder)	71.9	69.5	74.4	0.98

Table 5.6: Performance of models on Incremental Transductive Grading on *Python* code

5.3.3 Comparing with Human Grading

To compare the proposed models' grades with partial marks given by a domain expert, two experts have graded 200 random submissions from 10 random problems (with 10 incorrect and 10 correct solutions). The selected problems are of medium level difficulty necessitating few loops and conditionals to solve. The submissions are given a grade from 1 to 5 following the rubric described in Table 5.8. The average of the two graders' marks is considered as the final grade. The output of the proposed models that have been trained on weakly labelled data (i.e., the output of the *sigmoid* function) are normalized between 1 to 5. The correlation and MSE between the partial marks (given by the domain expert)

Model	Accuracy	F1	Correlation	MSE
Random Forest	65.6	66.7	50.2	1.60
SVM	65.4	68.6	56.6	**1.45**
LASSO (L1)	62.3	66.3	49.7	1.50
Ridge (L2)	63.2	67.2	48.5	**1.45**
Deep Grader (CodeBERT)	76.4	**77.6**	**76.9**	2.76
Deep Grader (UniXcoder)	76.7	77.0	76.2	2.87
Deep Siamese Grader (CodeBERT)	**77.5**	76.9	76.4	2.15
Deep Siamese Grader (UniXcoder)	75.1	74.5	75.2	1.84

Table 5.7: Performance of models on Incremental Transductive Grading on C^{++} code

and the models' normalized scores are reported in the performance tables.

Tables 5.2, 5.3 show significant improvement in the correlation measures when using the proposed methods. In particular, Deep Grader (CodeBERT) reveals a 16.2pp improvement over the best baseline (i.e., SVM) for the Python dataset, and Deep Siamese Grader (CodeBERT) is 19.9pp better than SVM for the C++ dataset. Tables 5.4, 5.5 demonstrate a similar improvement when multiple correct solutions are available in inference time. The results are outstanding given the fact that the models have not seen human grade labelled data during training. When comparing our Incremental Transductive Grading method's results presented in Tables 5.6, 5.7 with our single correct solution method's results from Tables 5.2, 5.3, we see improvements in the correlation values when considering the best results achieved by our models. Thus, our proposed Incremental Transductive Grading allows us to also improve on the partial grades that our models produce.

Score	Interpretation
5	**Exceptional:** The code meets all of the requirements specified in the problem.
4	**Acceptable:** Necessary control structures and data dependencies are present with minor errors.
3	**Amateur:** Correct control structures are present with problems in data dependency.
2	**Unsatisfactory:** Demonstrating a slight understanding of the problem.
1	**Unrelated:** Solution is unrelated to the problem.

Table 5.8: Grading Rubric

Direct training on soft labelled data is unfeasible as these large language models rapidly overfit due to the small size of the data. Both supervised and semi-supervised learning on soft labelled data produces poor, unstable and unreliable results.

5.3.4 Runtime Analysis

Figure 5.9 shows the average runtime of grading a program with the proposed models. The experiments were carried out on a cluster of 4 NVIDIA V100 GPUs. As can be seen, the grading time per program is fast (a fraction of a second).

5.4 Evaluation Methods

There are two ways to train and evaluate the models in this problem. We can split the data along questions (question independent model) or we can split the data along

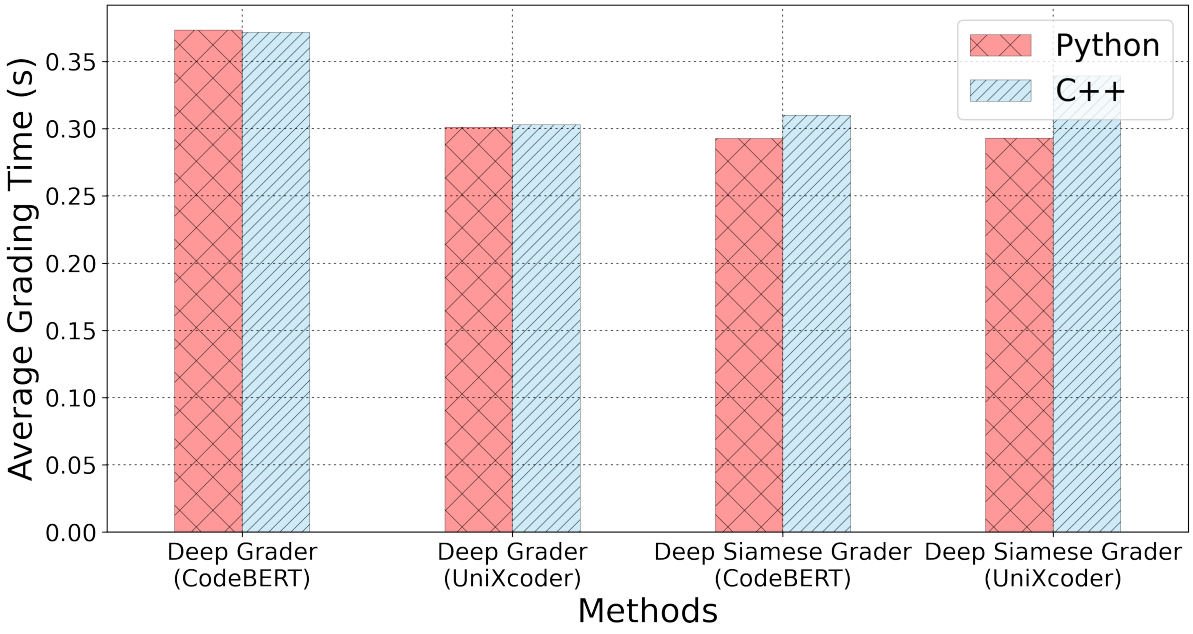


Figure 5.9: Models' Grading Time per Submission

responses (question dependent model). We focus our attention towards the former since it better reflects the real-life needs of such a model. By training a model on a sufficiently large scale of questions, we would be able to accurately predict labels for responses to any newly created programming questions.

5.4.1 Question Independent Grading

With our question independent model, responses are split along questions, thus 80% of the questions will appear in the training set, 10% will appear in the validation set and the responses from the remaining 10% of questions will be used for our test set. In this study, we mainly focus on this type of evaluation since it is the most closely related to a real-world deployment scenario of this model. However, if the environment permits, training a separate model for each programming question could possibly improve the models' performance.

5.4.2 Question Dependent Grading

The data can also be split along responses such that each set contains a portion of responses from all existing questions. The same percentage split would apply here, where 80% of responses for each question will appear in the training set and the rest will be equally separated between the validation and test set. Therefore, in this case, the model is trained on a number of questions that will have new unseen responses to predict. This method is useful if we reuse previously created questions with labeled responses to grade new responses. Naturally, this approach also provides us with greater results as the models, having already seen the questions, have an easier time predicting the weak labels for unseen programs (see Tables 5.9, 5.10).

Model	**Accuracy**	**F1**	**Correlation**	**MSE**
Random Forest	65.3	67.2	58.0	1.91
SVM	66.2	70.7	63.1	1.73
LASSO (L1)	66.5	73.9	48.5	2.03
Ridge (L2)	65.6	71.9	53.7	1.88
Deep Grader (CodeBERT)	82.4	84.2	76.5	1.67
Deep Grader (UniXcoder)	**82.8**	**84.3**	**78.7**	1.03
Deep Siamese Grader (CodeBERT)	73.4	73.4	76.5	0.97
Deep Siamese Grader (UniXcoder)	70.0	71.3	75.7	**0.93**

Table 5.9: Performance of models using single correct solution with a question dependent setting on *Python* code

Model	Accuracy	F1	Correlation	MSE
Random Forest	63.2	65.7	48.8	1.79
SVM	63.1	68.2	54.7	1.53
LASSO (L1)	61.9	67.9	48.8	**1.48**
Ridge (L2)	61.9	67.0	48.5	1.49
Deep Grader (CodeBERT)	82.1	83.3	**76.3**	3.25
Deep Grader (UniXcoder)	**84.5**	**85.6**	75.0	3.16
Deep Siamese Grader (CodeBERT)	78.0	76.4	70.9	3.65
Deep Siamese Grader (UniXcoder)	73.1	77.0	71.4	3.67

Table 5.10: Performance of models using single correct solution with a question dependent setting on C^{++} code

5.5 Interpretability

5.5.1 Attention Analysis

We performed an attention analysis on our deep-learning models. Transformer-based models process all tokens simultaneously and form direct connections between the tokens with an attention mechanism. This allows them to have great parallelization and high accuracy. We extracted the hidden layers' parameters as well as the attention layers from our trained models. In our case, the model is performing self-attention by searching for other tokens in the sequence that could positively influence the encoding of each token. We obtained an attention matrix of four-dimensionality where the dimensions represent the heads, the layers per head, the tokens that attend, and the tokens that receive attention. We can visualize this attention matrix to understand how the model forms composite representations to encode programs.

Let's take an example question i and a model input consisting of two programs: a sample program P^i (see Figure 5.10) and a correct solution P^i_+ (see Figure 5.11). The sample program is predicted as correct by the grading model with a 86% score. So we will attempt to interpret the decision of the model for this sample program given this correct solution as reference.

```
n,m = map(int,input().split())
l = [0]*n
for i in range(n):
    a,b = map(int,input().split())
    l[i] = (m*a)/b
print(min(l))
```

Figure 5.10: A sample program P^i

```
m,n = map(int,input().split())
x = []
for i in range(m):
    a,b = map(int,input().split())
    x.append(a/b*n)
print(min(x))
```

Figure 5.11: A correct program P^i_+

We can select a specific token in the sample program P^i, in this case the keyword "for", and plot out the attention scores for each layer and each head (see Figures 5.12, 5.13). Thus, we will analyze which tokens contributed the most to produce the encoding

of the "for" token. From these visualizations, we see that the model has designated specific heads and layers to pay attention to specific tokens. For example, in the first two heads, the models pays more attention to the tokens surrounding the "for" token to compute the encoding. While, in the third and fifth head, the model identifies the "for" token in the correct program as well and uses it to bake part of its representation into the encoding. We also observe that in most heads and layers, the [CLS] token or the sequence's start token has a high importance. This is the case when we analyze any token since the [CLS] token contains the embedding for the whole sequence. Thus, the model mostly uses the whole sequence's embedding to compute each token's encoding.

We can also reduce the heads and layers from the attention matrix and create a self-attention two-dimensional matrix to examine the attention of the whole sequence (see Figure 5.14). For illustration purposes, we remove the start token since it has a dominating effect on the matrix preventing us to clearly visualize the other attention locations. From this figure we observe that the highest attention scores are along the diagonal, in other words, tokens attend to themselves and their close surroundings the most.

We can also visualize this as a graph where the tokens are connected by lines of different strength dependent on the attention score between two tokens (see Figure 5.15a). However, there are too many lines in the graph, and thus, we cannot easily derive interpretations from this graph. Hence, we filter out the least important lines, in other words, the top 30% lowest attention score relational lines from the graph (see Figure 5.15b). From this filtered graph, we can interpret that some tokens have strong attention relations with tokens far away in the sequence. Although, these relations only appear for reoccurring tokens. We also observe that some tokens' encoding are highly influenced by neighbouring tokens. For example, the token "min" at the end of the sequence is mostly influenced by eight neighbouring tokens and a previous occurrence of the token. The apparent takeaway is that our program grading models mostly focus their attention

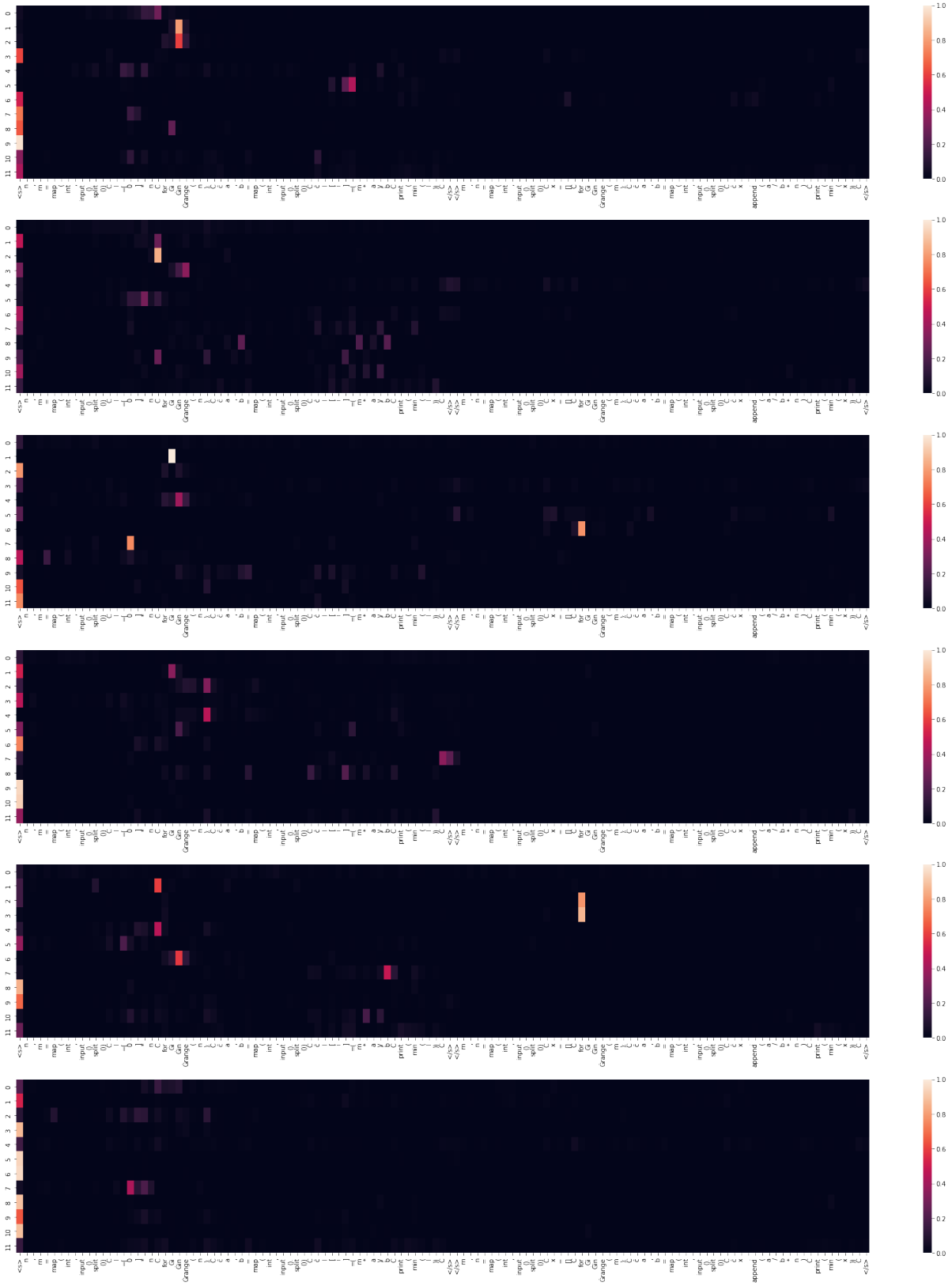


Figure 5.12: Attention for the token “for” over heads and layers (Heads 1-6)

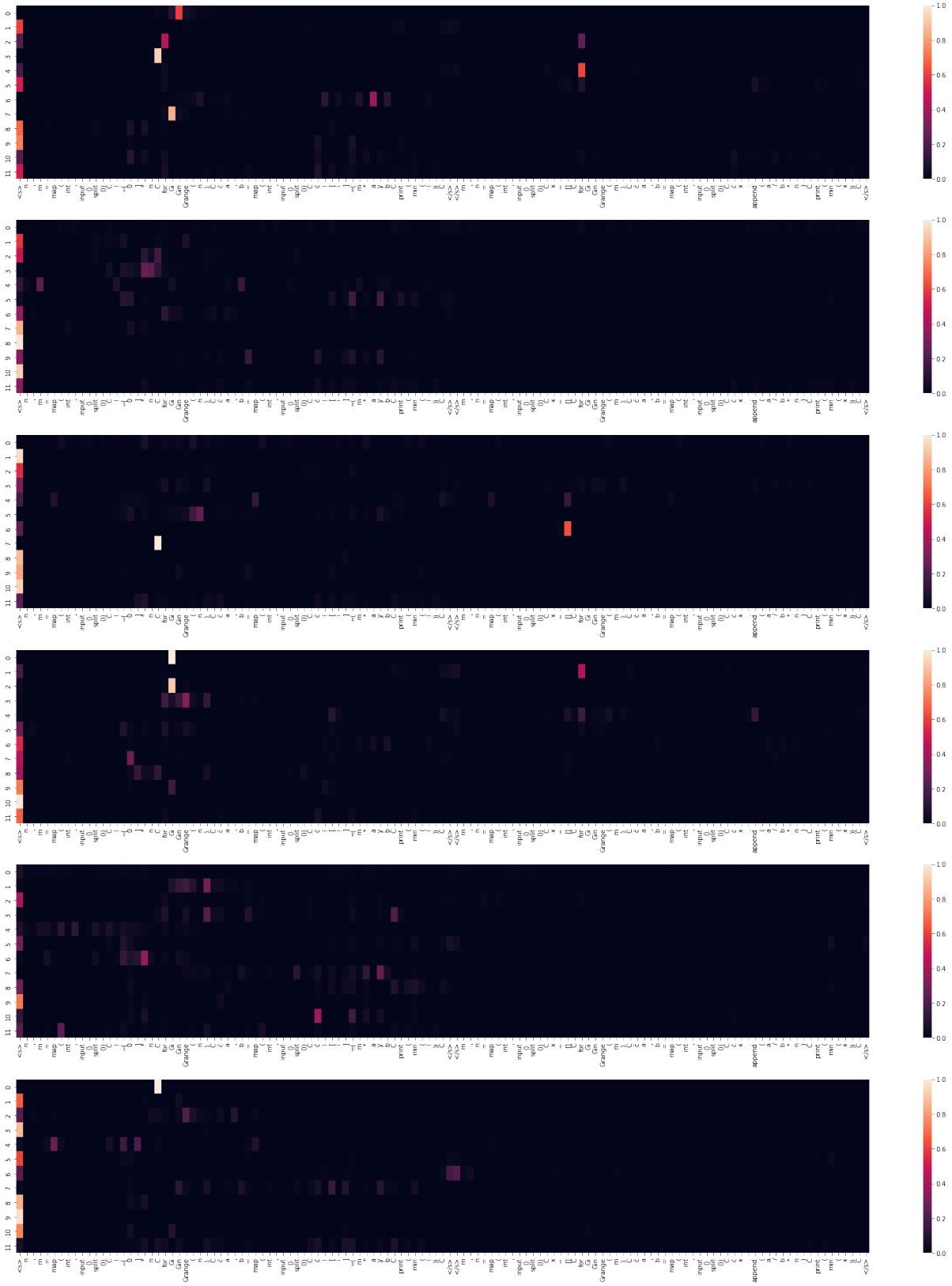


Figure 5.13: Attention for the token “for” over heads and layers (Heads 7-12)

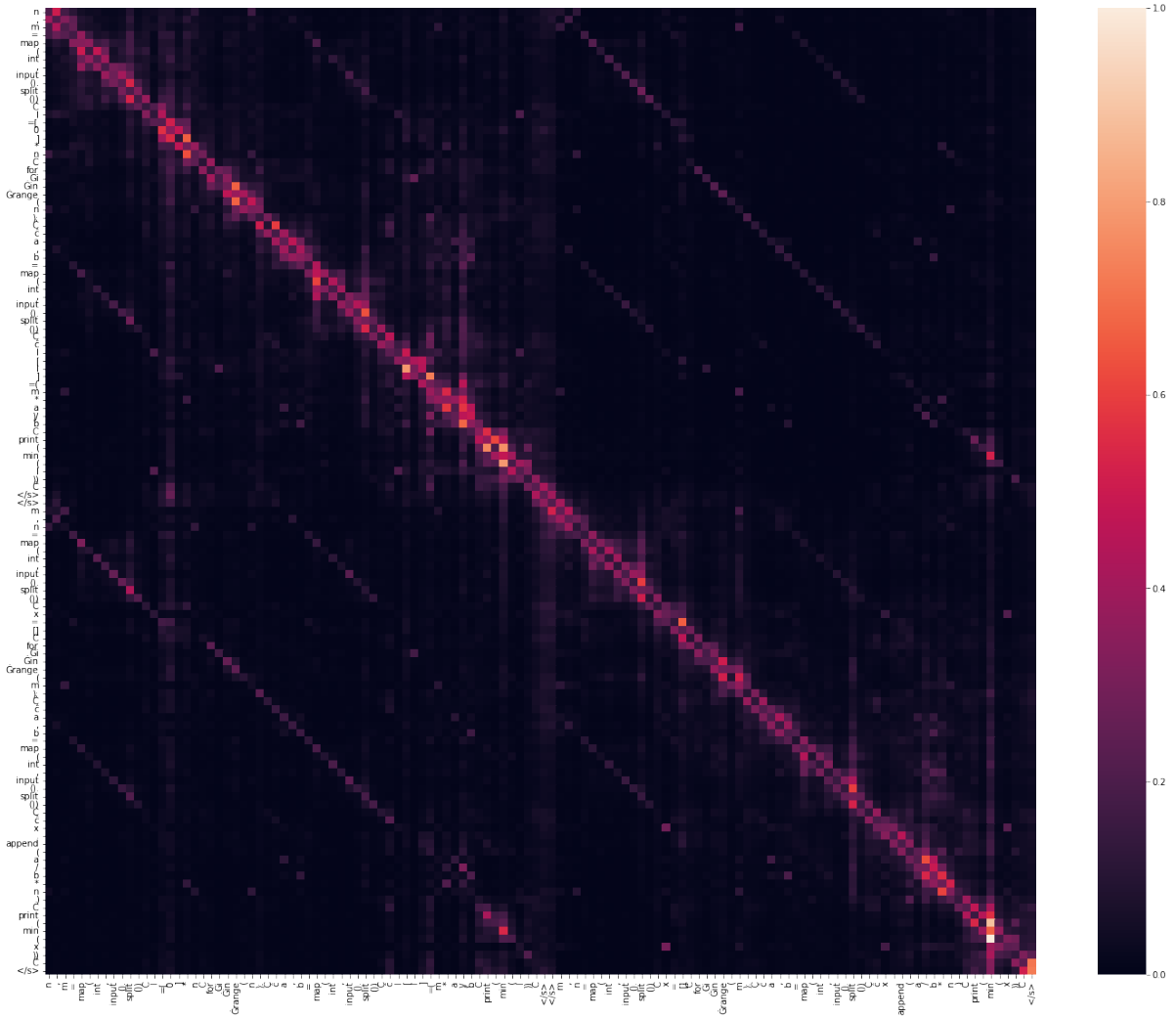
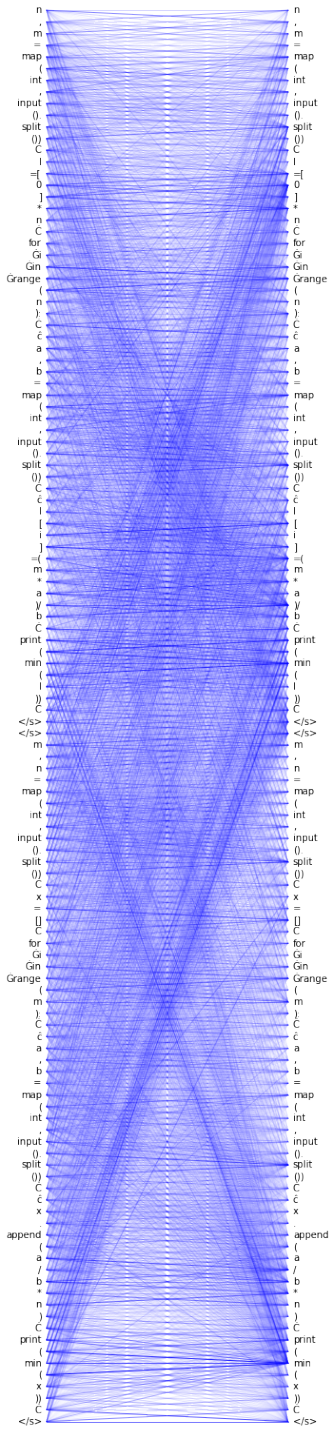
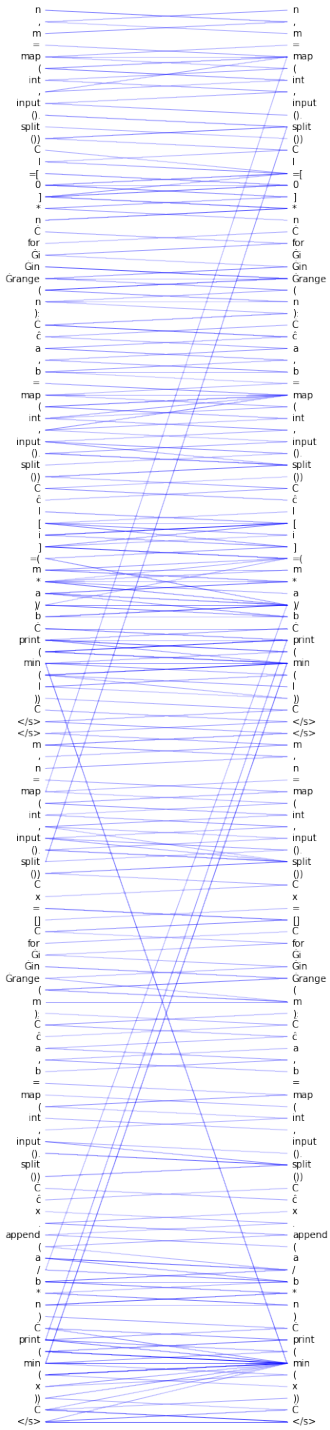


Figure 5.14: Self-attention plot



(a) Self-attention graph



(b) Self-attention graph filtered

on neighbouring tokens and co-occurring tokens when encoding the embeddings of the sequence.

Since we have two programs in our input sequence, we can partition the self-attention matrix in order to visualize how the two programs attend to each other, and therefore, explain the model's prediction. In order to create the partition, we take only the bottom left part of our self-attention matrix (see Figure 5.16). That is, we have the correct program's tokens on the y-axis and the sample program's tokens on the x-axis. We generate a filtered version (see Figure 5.17) and line plots (see Figures 5.18a, 5.18b) with this partitioned matrix. In the line plot representations, the left-side shows the tokens of the correct program P_+^i and the right-side shows the tokens of the sample program P^i. From the filtered self-attention plot, we observe an incomplete diagonal line with missing pieces, however, a large number of tokens appear in both programs at around the same location. From the filtered graph, we see that most tokens co-occur in both programs, and thus, we have high attention scores between co-occurrences. However, variables in the programs do not have strong attention relations with each other. This is a crucial finding from the model, since generally in two programs, variables can be defined with the same keyword but have different purposes and hold different values. So the model should not rely on variable keywords to measure the similarity of programs.

Finally, we can reduce the dimension of the attention matrix even more by obtaining the attention weights of each specific token in the sequence. This way we can visualize which tokens were the most important in the decision of the model. We decide to display this matrix directly on the code using a highlighting technique where lightly highlighted tokens represent low attention weights and strongly highlighted tokens represent high attention weights (see Figure 5.19). With this visualization, we observe that some tokens present in both programs have strong attention scores and we interpret these tokens as the most important tokens that need to be present for a program to be considered correct. Conversely, we also observe that some tokens with high attention scores only appear in

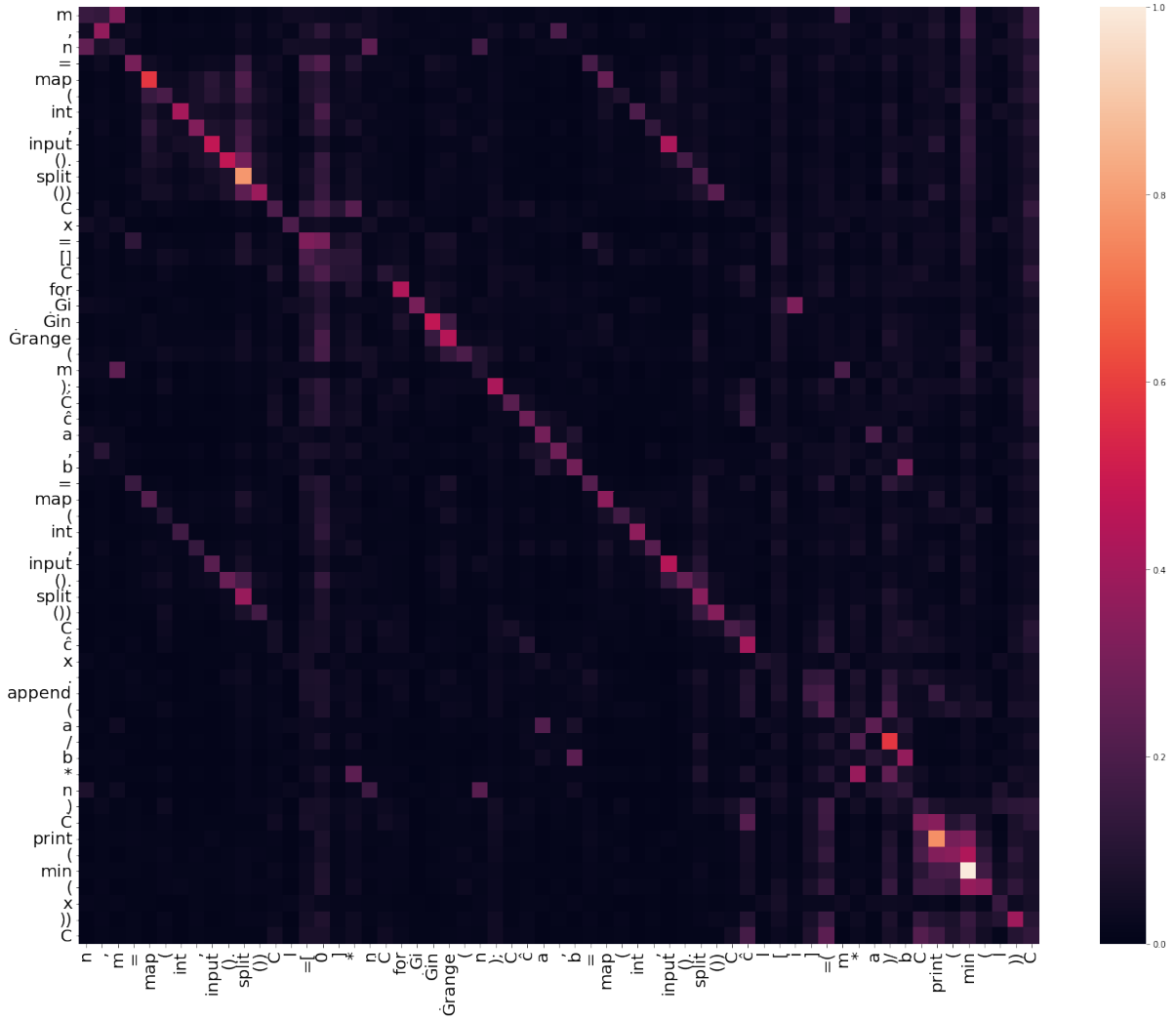


Figure 5.16: Self-attention plot (partition)

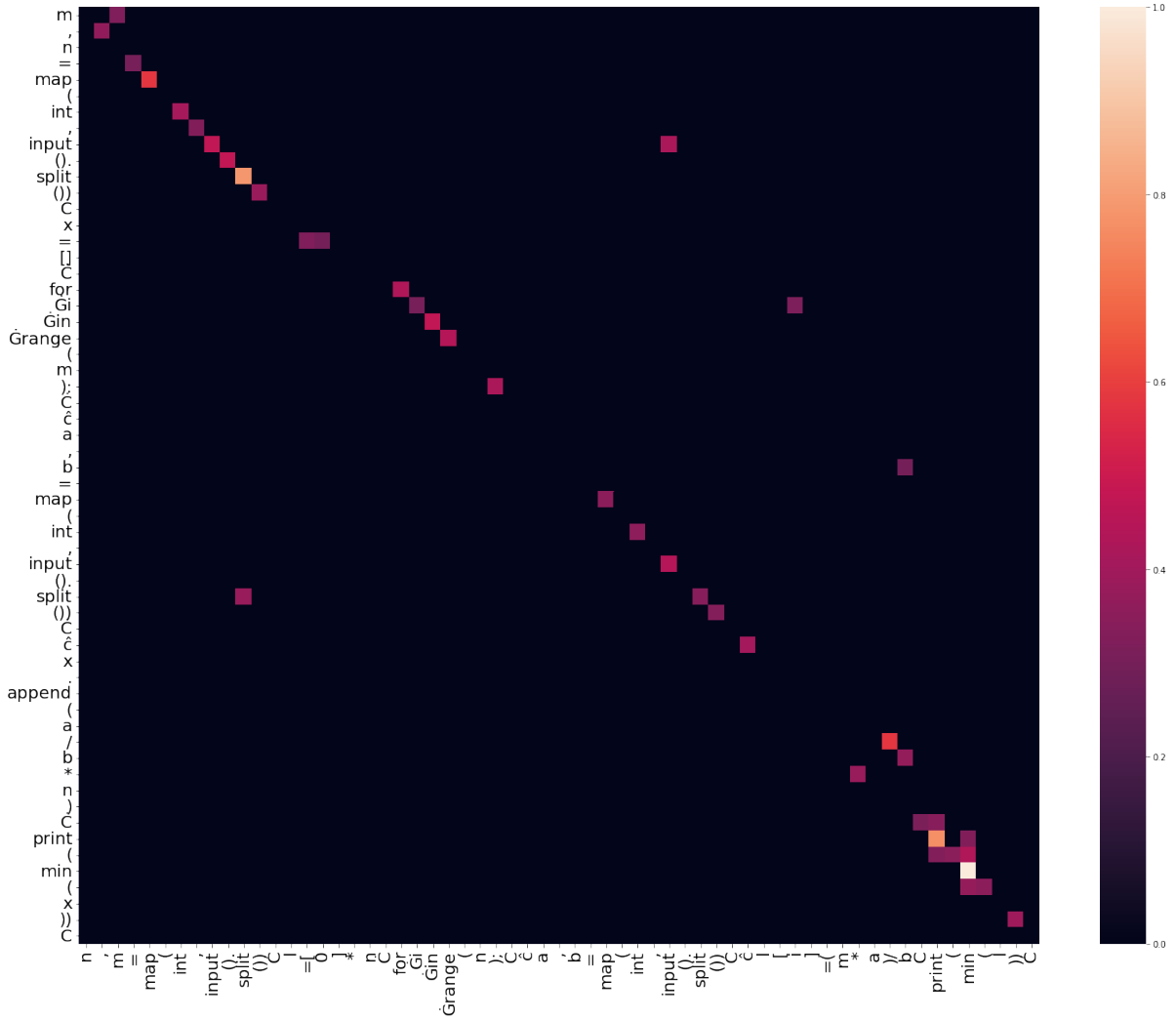
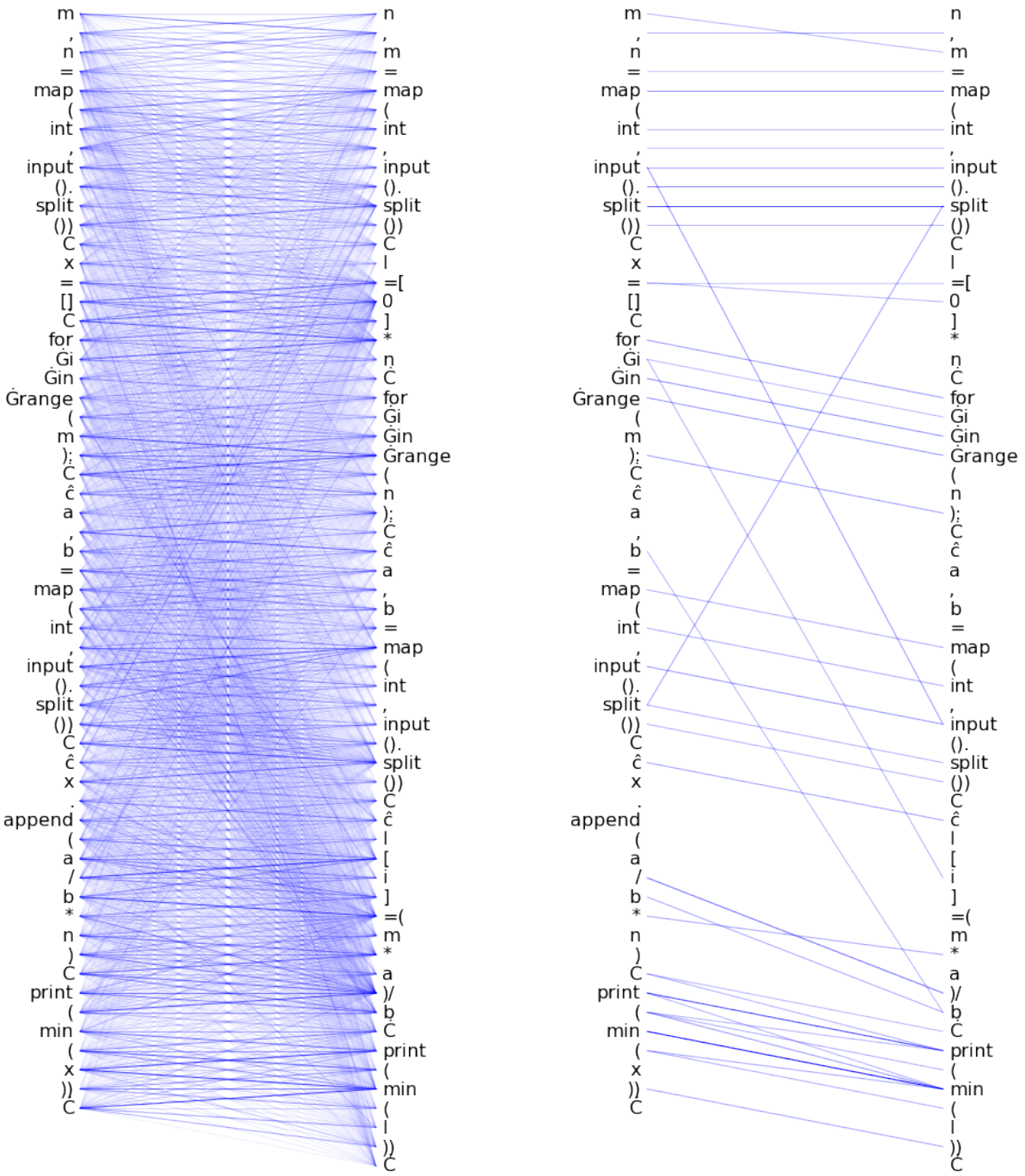


Figure 5.17: Self-attention plot filtered (partition)



(a) Self-attention graph (partition)

(b) Self-attention graph filtered (partition)

one program signifying that since these tokens are missing in the other program, the sample program is more likely to be considered incorrect. All in all, by investigating the attention mechanism of our deep language models, we have found numerous insights from the models by analyzing multiple types of visualizations of programs as attention matrices.

```
n,m=map(int,input().split())
l=[0]*n
for i in range(n):
    a,b=map(int,input().split())
    l[i]=(m*a)/b
print(min(l))


m,n=map(int,input().split())
x=[]
for i in range(m):
    a,b=map(int,input().split())
    x.append(a/b*n)
print(min(x))
```

Figure 5.19: Program highlighted with attention weights

5.5.2 Case Study of using LIME

We use LIME to evaluate our program 5.10 and retrieve the tokens that contributed the most to the prediction of the model both negatively and positively (see Figure 5.20). LIME generated 100 perturbations of the program and yielded the 10 most influential features or tokens from the sequence. From the explanations given from LIME, we derive that the use of the "min" function was the most influential feature of the program followed by the "print" function, and the "split" function. It seems that for this problem omitting any of these functions will render the program incorrect. The "min" function can be

justified by the fact that the output should be the minimum value from a list and it is the default approach, however, there are multiple ways of retrieving the minimum value from a list without using the "min" function. The "print" function is obviously important for the program to produce an output and there is not many alternatives. Finally, the "split" function is important to split the inputs over commas and it is the most logical approach. The rest of the features have low weights so they don't influence the prediction as much. We also provide an example of a program that is considered incorrect by the model using the same correct reference solution (see Figure 5.21). Although, in this case, LIME does not provide as strong feature weights as previously to explain the prediction even though the prediction strongly indicates that this is an incorrect implementation. The most influential feature found which explains the sample program's incorrectness is the "if" token since the correct reference program does not contain an "if" statement. All in all, LIME successfully identifies the most important parts of the program that justify the prediction. This feedback provided from LIME can easily be used by evaluators to understand the critical points of the program that influence its grade.

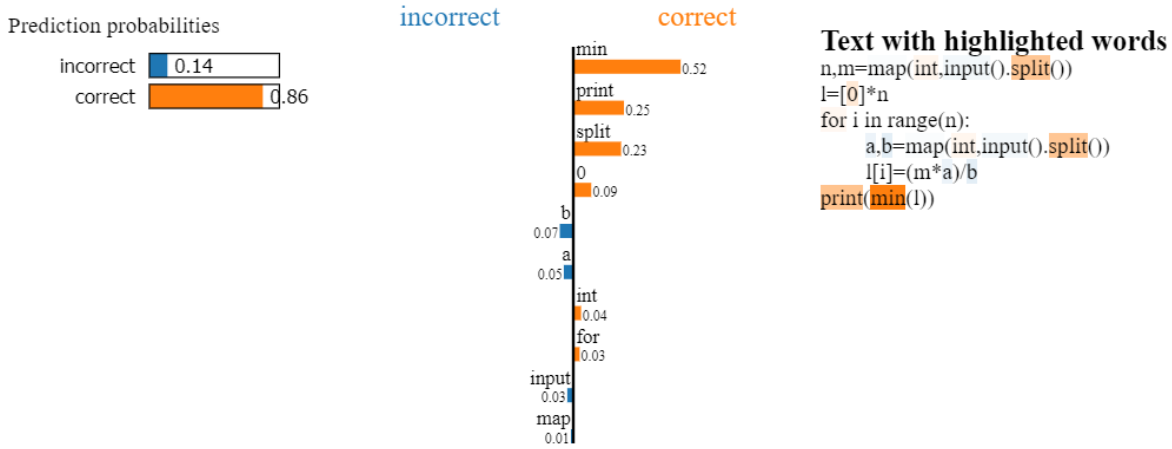


Figure 5.20: LIME explanation of a correct implementation

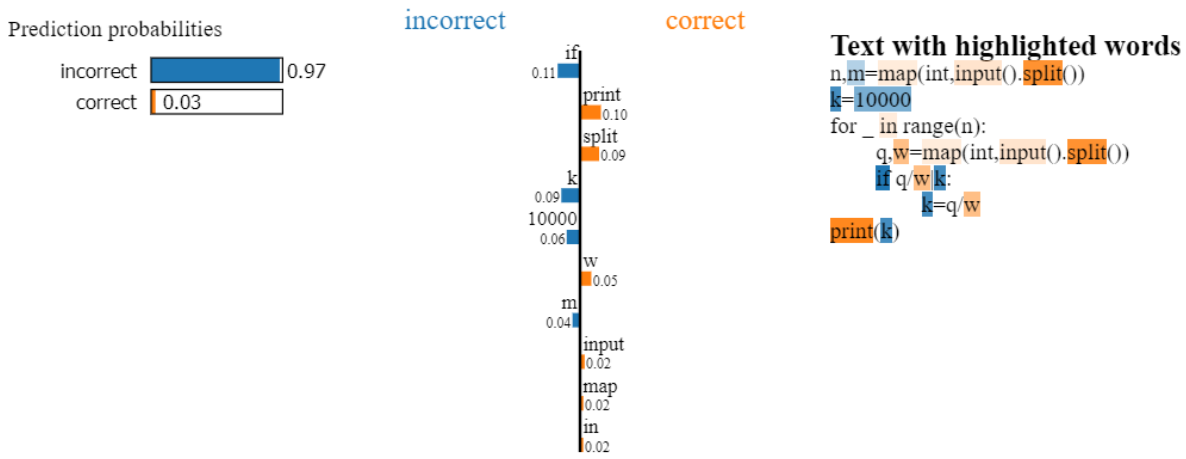


Figure 5.21: LIME explanation of an incorrect implementation

5.6 Summary

In this chapter, we evaluated our proposed deep language models leveraging the pre-trained CodeBERT and UniXcoder models on the task of computer program grading providing both binary correctness and partial grading. We then compared a feature engineering approach against our models to justify the importance and performance of large deep neural networks pre-trained specifically on programming language with millions of programs. These results prove the usefulness and efficiency of an automatic program grader powered by our proposed models. We also proposed different algorithms and settings tackling different type of deployment environments the models can perform in. Additionally, we present multiple ways of analyzing and interpreting the decisions of our deep language models.

Chapter 6

Conclusion and Future work

In this thesis, we presented a question-independent framework to automatically grade computer programs without the need of test-cases. We designed two architectures and fine-tuned the state-of-the-art pre-trained deep programming language models (i.e., CodeBERT [23], UniXCoder [28]) for the code grading task. Considering that the proposed approaches do not need any feature engineering, which depends on the programming language that we consider, they outperform the shallow models by a high margin. We also present a feature-engineering approach to this problem by computing different metrics from the keywords and graph representations of programs. Thus, we contribute to the field of computer program grading with new approaches to tackle the problem and provide a multitude of resources for future research.

When considering the ethical and social implications of an automatic program grader in a real-world scenario, we need to examine the fairness of such a technology. As it has certain limitations, there will always be edge cases where an automatic grader will fail. In those cases, the programs will need to be manually graded to obtain reliable grades. Further implications of the ethical and social aspects of this work will need to be further examined in a future work.

6.1 Thesis Contribution Highlights

The main contributions of this thesis can be summarized as follows:

- First, we demonstrate a straightforward architecture built by fine-tuning the state-of-the-art deep learning models: CodeBERT, UniXCoder.

- Additionally, we present a more complex architecture, namely the Siamese network, which circumvent some of the shortcomings of large language models (i.e., input vector size limitation), and introduce a contrastive learning approach.

- In particular, we developed a set of baselines based on both *count-based* and *graph-based* features and showed that they underperform compared to the proposed models.

- Furthermore, we developed a transductive learning algorithm taking advantage of correct solutions in the test set to make better predictions. In fact, this algorithm builds on another algorithm (i.e., Majority vote) which enables the model to evaluate programs using multiple reference solutions.

- We successfully teach deep learning models how to grade computer programs without the need of test-cases. To the best of our knowledge, this is the first work towards using deep learning models for the computer program grading task.

- We conduct different methods of analyzing the interpretability of deep language models and obtain various insights and potential feedback that can be provided to a human evaluator to specify the reasoning that have led to the given grade.

- We also investigate an application of our automatic program grader by providing certain improvements to the current work in program synthesis.

6.2 Limitations

The proposed models in this research demonstrated great results on automatic program grading. However, there are some limitations to this work that would need to be considered.

- While this work's main objective is the grading of computer programs, the proposed models were not trained on giving a partial grade to a program. Although, we see promising correlations between the model's grades and human grades, we would require a larger set of graded programs in order to be able to specifically train the model on predicting program grades. Since test-cases present multiple drawbacks, a human labelled training dataset would also serve to completely remove test-cases from the framework.

- The selection of the correct reference solution greatly influences the grading results obtained from our model. One possible research direction would be to improve on the random selection by selecting the reference solution that best represents a correct solution for the problem.

- Lastly, the size and complexity of the problems and corresponding solutions affects the performance of the models. By training the models restrictively on separate problem categories ranging from simple to more complex problems, we could minimize the disturbance in training.

6.3 Future Work

With an extended soft label data we could experiment using supervised learning directly on graded programs data. Developing semi-supervised models taking advantage of a large pool of weakly labeled programs and a smaller pool of programs evaluated by human graders in the training phase would be another interesting research direction.

The collection of graded data would be a simple process with the incorporation of data collection in programming courses while respecting the institution's regulations.

Another approach to tackle the problem would be to develop a Graph Neural Network model utilizing abstract syntax trees to leverage the code's structural properties inside the model. This approach requires a considerably large memory as abstract syntax tree representations of programs are immense. Additionally, all programs will have different size representations, thus a padding and a truncation method would have to be used considerably limiting the program's size.

We hope that the proposed models and the resources (including the models and datasets) which are prepared by this work pave the way for further research.

Chapter 7

Appendix I

7.1 Training Details

We provide the training details for the proposed models in Table 7.1.

Table 7.1: Training details

Models	**Batch size (per gpu)**	**Learning rate**	**Number of epochs**	**Time per epoch**
Deep Grader (CodeBERT)	16	5e-5	3	0:29:27
Deep Grader (UniXcoder)	16	5e-5	3	0:30:31
Deep Siamese Grader (CodeBERT)	8	5e-5	3	1:13:40
Deep Siamese Grader (UniXcoder)	8	5e-5	3	1:18:55

7.2 Applications of Deep Grader in Program Synthesis

7.2.1 Overview

The emergence of deep learning has substantially advanced the field of programming language (PL) learning with generative models. The main tasks tackled in this field are the task of program induction and program synthesis. The former consists of generating program outputs from contextual program representations. The latter is the task of generating code based on a given context specification. In [82], they proved that language models can learn to use operators such as addition, subtraction in a general way and they could also memorize effectively. Researchers have then continued work towards program induction by introducing the Neural Program Interpreter [50, 55, 61] and the Universal Transformer [18]. On the other hand, program synthesis involves the generation of code based on a natural language contextual specification. Popular approaches in this field have tried using abstract syntax trees (ASTs) to incorporate the code's structure into the model and generate structurally correct code [3, 4, 46, 81]. Others have found that omitting AST representations and using n-grams or character-level language models have higher success in program synthesis [8, 32, 43], for example, with the Latent Predictor Networks [43] or the DeepCoder [8] model which was trained on predicting methods in code based on function definitions or docstrings. With an interesting approach, Balog et al. [8] focus on solving difficult programming problems using provided input and expected output examples. They approach the problem from a more indirect view by searching for specific pre-written code from the community and puzzling the snippets of code together to answer the programming problems.

Recently, a large body of research has been made towards the longstanding challenge of program synthesis prompted by the great advancements of Transformer-based language

models and the substantial amount of code publicly available on Github. Due to the large success of Transformer-based language models in the NLP field like T5 [54] and GPT [11, 12, 52, 53, 73], researchers have proposed programming language counter parts like PyMT5 [16], GPT-J [73], and Codex [13] for program synthesis, or more specifically, for translating between function signature/docstring and body. The GPT-J model has a size of approximately 6 billion parameters and was trained on the Mesh Transformer JAX [72] codebase. Unlike most of the other large GPT models, the GPT-J model was publicly released along with a web demo version [1].

From Chen et al. [13], the approach has become more direct in the way the model learns to understand and generate code without the need of any corpus or reference, however, the difficulty of problems tackled has not improved. In this task, the generated code samples are evaluated using test-suites. Unlike with natural language generative models which are evaluated with heuristics such as the BLEU [48] score. Using such heuristics proves to be detrimental in capturing semantic features specific to code [57]. Furthermore, match-based metrics are unable to accommodate for the large space of correct programs functionally equivalent to a given solution. Thus, Chen et al. [13] evaluate their models with the $pass@k$ metric which is defined as the fraction of problems that are solved by generating k sample programs. To solve programming problems, they generate multiple samples from the models, and check if any of them pass the unit tests. The number of samples generated appears to be proportional with the chance of one of them passing the test suite. The method of generating a single sample solves 28% of problems, while the method of generating 100 samples and selecting the one that passes the test suite solves 77% of problems. If the test-case evaluation of 100 samples per problem is too demanding or not feasible in deployment, they propose using the mean log-probability to determine the best sample. With this approach 44% of problems are solved. To train their models they use 159GB of unique Python files retrieved from 54

[1] https://6b.eleuther.ai/

million public software repositories hosted on GitHub. To benchmark their models they create a dataset, called HumanEval, which is composed of 164 hand-written programming problems unseen to the models. Each problem consists of a docstring, a function signature, a body, and a test-suite to test the function.

Current generative models show difficulty in generating a correct solution when faced with complex problems. What makes a problem harder than another is the natural language specification that is provided. It's length, vocabulary complexity, and number of components regulates the difficulty of a problem. For example, the further we increase the number of chained components in the context (see Figure 7.1), the worse the generative model performs on generating a correct solution to the problem even though the individual components are trivial for the model to solve (see Figure 7.2).

```
# Add 3 to b,
# then subtract 4 from a and b,
# then return the product of the 4 numbers.
def compute(a,b,c,d):
    t = b + 3
    u = a - 4
    v = c * d
    return v
```

Figure 7.1: Example of a generated solution for a three component problem.

An important drawback of [13] that we attempt to address is the use of test-cases to evaluate generated programs. They generate multiple program samples for each problem and filter out the best one by running the programs on test-cases. Thus, the current code generative models are highly dependent on unit-testing. Generative language models are trained on publicly available code, hence, they have unknown intent and are often

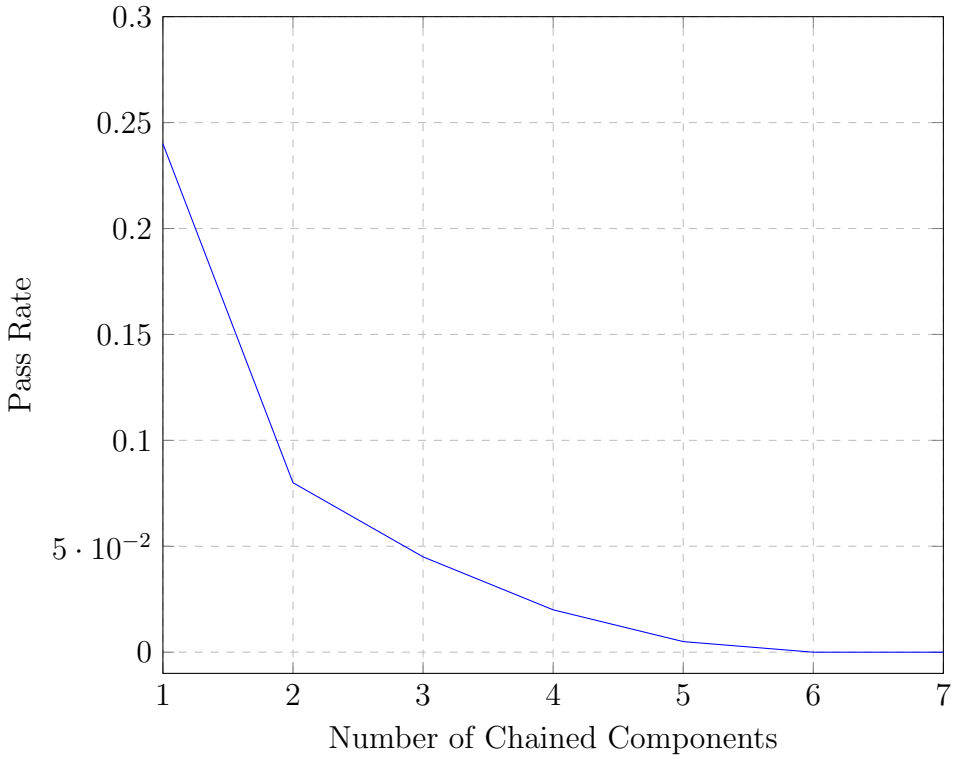


Figure 7.2: Effect of multiple components on the performance of generative models.

incorrect; some posing a serious security risk if executed. Running potentially malicious generated code on test-cases requires proper security measures taken in place and certain deployment environments do not have access to such infrastructures.

Thus, we suggest a more efficient sampling strategy without the need of test-cases which correlates more to the program's correctness than some of the previously proposed methods. At the same time, it poses less security risks and requires less resources compared to the best performing sampling strategy.

The following examples (see Figure 7.2, 7.3) show how our proposed program grading model handles different situations when it comes to syntactically different generated programs. The first generated example (see Figure 7.2) gets attributed a score of 81% by the correctness model and we can observe that it is only missing the definition of the variable "x_mean" when compared to the reference solution. In the second example (see Figure 7.3), the generated solution gets a 100% score even though the reference solution

uses a different variable naming.

Generated Solution

```
from typing import List


def mean_absolute_deviation(numbers: List[float]) -> float:
    """ For a given list of input numbers, calculate Mean Absolute Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the average absolute difference between each
    element and a centerpoint (mean in this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0
    """
    return sum(abs(x - x_mean) for x in numbers) / float(len(numbers))
```

Reference Solution

```
from typing import List


def mean_absolute_deviation(numbers: List[float]) -> float:
    """ For a given list of input numbers, calculate Mean Absolute Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the average absolute difference between each
    element and a centerpoint (mean in this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0
    """
    mean = sum(numbers) / len(numbers)
    return sum(abs(x - mean) for x in numbers) / len(numbers)
```

Table 7.2: Example of a generated program with a correctness score of 81% and its reference correct solution.

7.2.2 Improved Sampling Strategy

In [13], they proposed generating k solutions to a problem and sampling the best one. They proved that a larger value of k produces better results (up to $k = 100$). They used test-cases for sampling the best generated codes (i.e., oracle) and proposed the mean log-probability from the sampling tree in environments where running a test-suite numerous times is not possible or practical in certain deployment scenarios. The mean

Generated Solution

```
def get_positive(l: list):
    """Return only positive numbers in the list.
    >>> get_positive([-1, 2, -4, 5, 6])
    [2, 5, 6]
    >>> get_positive([5, 3, -5, 2, -3, 3, 9, 0, 123, 1, -10])
    [5, 3, 2, 3, 9, 123, 1]
    """
    return [x for x in l if x > 0]
```

Reference Solution

```
def get_positive(l: list):
    """Return only positive numbers in the list.
    >>> get_positive([-1, 2, -4, 5, 6])
    [2, 5, 6]
    >>> get_positive([5, 3, -5, 2, -3, 3, 9, 0, 123, 1, -10])
    [5, 3, 2, 3, 9, 123, 1]
    """
    return [e for e in l if e > 0]
```

Table 7.3: Example of a generated program with a correctness score of 100% and its reference correct solution.

log-probability is taken from the probabilities of tokens along a path in the sampling tree of the generative model. An example of a natural language sampling tree can be seen in Figure 7.3.

As these probabilities have little to do with the correctness of programs, we propose a test-case free sampling method using our correctness score that aims to outperform the mean log-probability method. Since the final evaluation is based on passing the test-suite, the oracle sampling, which is based on test-cases, acts as an upper bound baseline to our proposed sampling technique.

We evaluated the sampling methods within the repeated sampling environment proposed by Chen et al. [13] and we benchmarked our correctness score based method to the oracle and to the mean log-probability sampling on the HumanEval dataset with

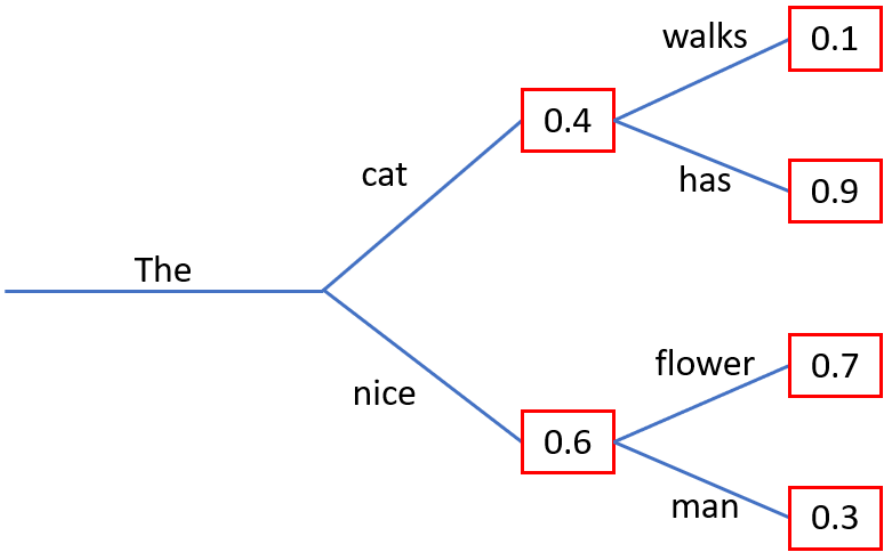


Figure 7.3: Example of Sampling Tree

the GPT-NEO 1.3B model (see Figure 7.4). We observe that as the k value increases our proposed method appears to better correlate to the oracle compared to the mean log-probability method.

Due to resource constraints, the chosen generative model is not capable of solving many problems. Furthermore, we could only evaluate the model up to $k = 10$. Thus, the pass rates are low and it is difficult to draw statistical conclusions from the sampling techniques. When we visualize the distribution of correctness scores for the problems in the HumanEval dataset, we notice a clear demonstration of the capabilities of the generative model (see Figure 7.5). We observe that, for the majority of problems, the GPT-NEO 1.3B model does not produce high correctness score solutions which explains the low pass rates in Figure 7.4.

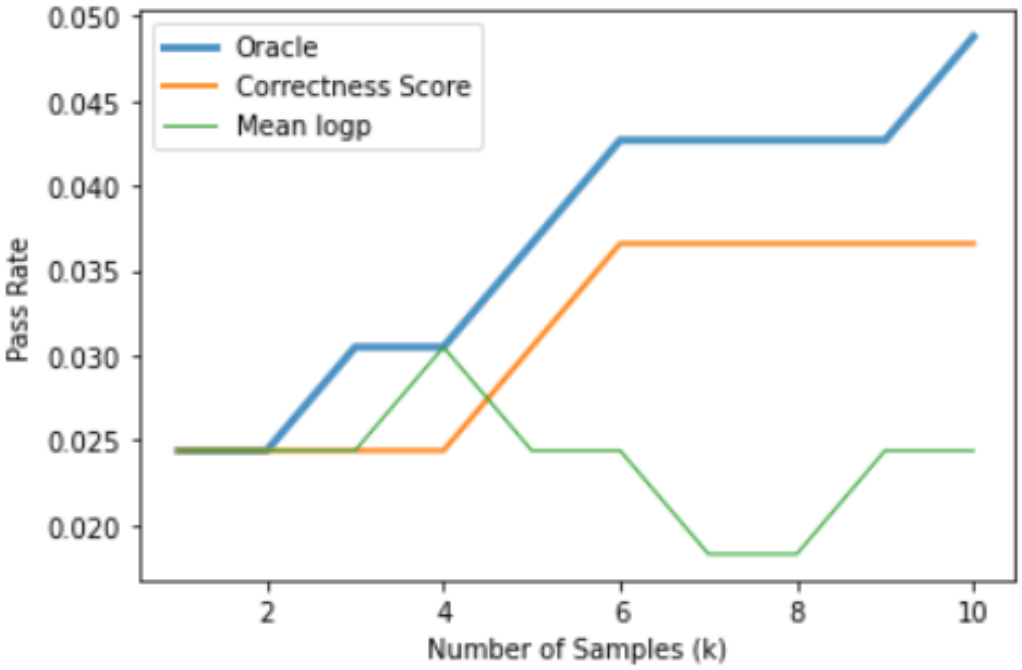


Figure 7.4: Sampling methods for generated programs

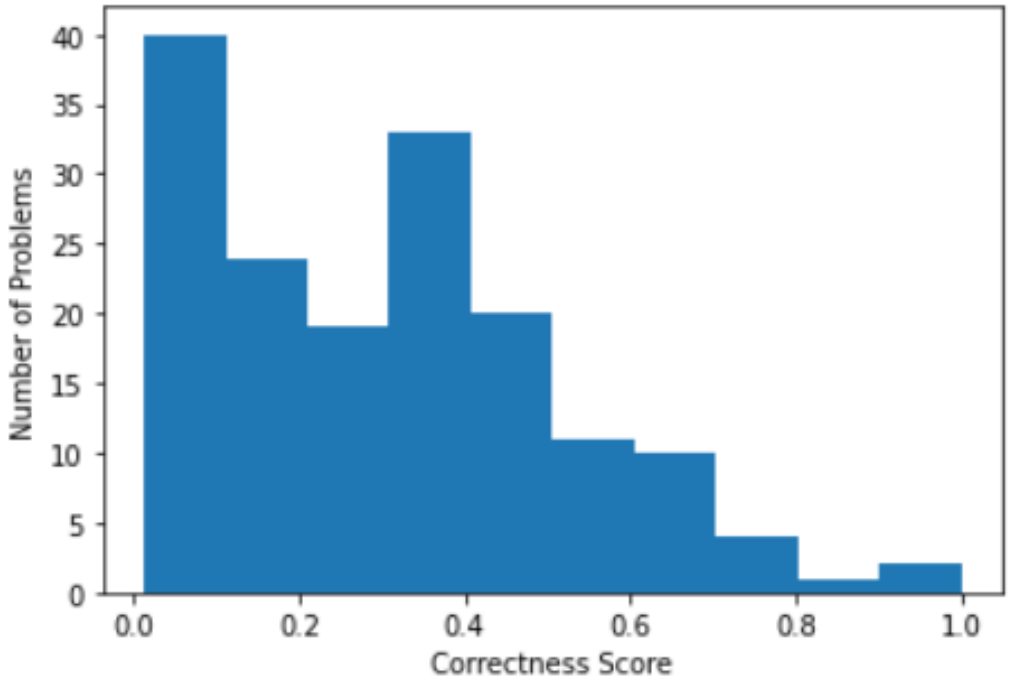


Figure 7.5: Distribution of Correctness Scores (k=10)

Bibliography

[1] Varun Aggarwal, Shahank Srikant, and Vinay Shashidhar. Principles for using machine learning in the assessment of open response items: Programming assessment as a case study. In *NIPS Workshop on Data Driven Education*, 2013.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL `https://aclanthology.org/2021.naacl-main.211`.

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France, 07–09 Jul 2015. PMLR. URL `https://proceedings.mlr.press/v37/allamanis15.html`.

[4] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1gKYo09tX`.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning

distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290353. URL `https://doi.org/10.1145/3290353`.

[6] David Alvarez-Melis and Tommi S Jaakkola. A causal framework for explaining the predictions of black-box sequence-to-sequence models. *arXiv preprint arXiv:1707.01943*, 2017.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ArXiv*, 1409, 09 2014.

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016. URL `http://arxiv.org/abs/1611.01989`.

[9] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL `https://arxiv.org/abs/2004.05150`.

[10] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

[11] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL `https://doi.org/10.5281/zenodo.5297715`.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess,

Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

[14] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. URL `http://arxiv.org/abs/1412.3555`.

[15] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*,

pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-4828. URL https://aclanthology.org/W19-4828.

[16] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *CoRR*, abs/2010.03150, 2020. URL https://arxiv.org/abs/2010.03150.

[17] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, sep 1995. ISSN 0885-6125. doi: 10.1023/A:1022627411411. URL https://doi.org/10.1023/A:1022627411411.

[18] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. *CoRR*, abs/1807.03819, 2018. URL http://arxiv.org/abs/1807.03819.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

[20] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4–es, September 2005. ISSN 1531-4278. doi: 10.1145/1163405.1163409.

[21] Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International Convention on Information and Commu-*

nication Technology, Electronics and Microelectronics (MIPRO), pages 0210–0215, 2018. doi: 10.23919/MIPRO.2018.8400040.

[22] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. In M.C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996. URL https://proceedings.neurips.cc/paper/1996/file/d38901788c533e8286cb6400b40b386d-Paper.pdf.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[24] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978. ISSN 0378-8733. doi: https://doi.org/10.1016/0378-8733(78)90021-7. URL https://www.sciencedirect.com/science/article/pii/0378873378900217.

[25] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL https://aclanthology.org/2021.emnlp-main.552.

[26] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and pro-

gram repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 465–480, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192387.

[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=jLoC4ez43PZ`.

[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics. URL `https://aclanthology.org/2022.acl-long.499`.

[29] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, 2006. doi: 10.1109/CVPR.2006.100.

[30] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL `https://openreview.net/forum?id=sD93GOzH3i5`.

[31] Hilt, Donald E. and Seegrist, Donald W., and United States. Forest Service. and

Northeastern Forest Experiment Station (Radnor, Pa.). *Ridge, a computer program for calculating ridge regression estimates*, volume no.236. Upper Darby, Pa, Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977, 1977. URL `https://www.biodiversitylibrary.org/item/137258`. https://www.biodiversitylibrary.org/bibliography/68934.

[32] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[33] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995. doi: 10.1109/ICDAR.1995.598994.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. URL `http://arxiv.org/abs/1909.09436`.

[36] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In Cassio de Campos and Marloes H. Maathuis, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 54–63. PMLR, 27–30 Jul 2021. URL `https://proceedings.mlr.press/v161/jiang21a.html`.

[37] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. *J. Educ. Resour. Comput.*, 5(3):2–es, September 2005. ISSN 1531-4278. doi: 10.1145/1163405.1163407.

[38] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/kanade20a.html`.

[39] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1181. URL `https://aclanthology.org/D14-1181`.

[40] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR. URL `https://proceedings.mlr.press/v32/le14.html`.

[41] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL `https://aclanthology.org/2020.acl-main.703`.

[42] Yiyi Liao, Yue Wang, and Yong Liu. Graph regularized auto-encoders for image representation. *IEEE Transactions on Image Processing*, 26(6):2839–2852, 2017. doi: 10.1109/TIP.2016.2605010.

[43] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W. Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *CoRR*, abs/1603.06744, 2016. URL `http://arxiv.org/abs/1603.06744`.

[44] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. Automatic grading of programming assignments: An approach based on formal semantics. In *ICSE-SEET*, pages 126–137, 2019.

[45] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL `http://arxiv.org/abs/1907.11692`.

[46] Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 649–657, Bejing, China, 22–24 Jun 2014. PMLR. URL `https://proceedings.mlr.press/v32/maddison14.html`.

[47] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadel-

phia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040.

[49] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://aclanthology.org/D14-1162.

[50] Thomas Pierrot, Guillaume Ligner, Scott E. Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. *CoRR*, abs/1905.12941, 2019. URL http://arxiv.org/abs/1905.12941.

[51] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. Sk_p: A neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2016, page 39–40, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344371. doi: 10.1145/2984043.2989222. URL https://doi.org/10.1145/2984043.2989222.

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[53] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learn-

ing with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL `http://jmlr.org/papers/v21/20-074.html`.

[55] Scott Reed and Nando de Freitas. Neural programmer-interpreters, 2016.

[56] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL `https://aclanthology.org/D19-1410`.

[57] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. URL `https://arxiv.org/abs/2009.10297`.

[58] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939778. URL `https://doi.org/10.1145/2939672.2939778`.

[59] Kelly Rivers and Kenneth R Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, volume 50, 2013.

[60] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[61] Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/7776e88b0c189539098176589250bcba-Paper.pdf`.

[62] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 263–272, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939696.

[63] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462195.

[64] Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 1887–1896, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329569. doi: 10.1145/2623330.2623377.

[65] Jiao Sun, Q. Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D. Weisz. Investigating explainability of generative ai for code through scenario-based design. In *27th International Conference on Intelligent User Interfaces*, IUI '22, page 212–228, New York, NY, USA, 2022. Association for

Computing Machinery. ISBN 9781450391443. doi: 10.1145/3490099.3511119. URL https://doi.org/10.1145/3490099.3511119.

[66] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[67] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[69] Giulia Vilone and Luca Longo. Explainable artificial intelligence: a systematic review. *CoRR*, abs/2006.00093, 2020. URL https://arxiv.org/abs/2006.00093.

[70] Urs von Matt. Kassandra: The automatic grading system. *ACM Special Interest Group on Computer Uses in Education Outlook*, 22, 03 2001.

[71] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2377–2388, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510050. URL https://doi.org/10.1145/3510003.3510050.

[72] Ben Wang. Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX. `https://github.com/kingoflolz/mesh-transformer-jax`, May 2021.

[73] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`, May 2021.

[74] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020.

[75] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. 2021.

[76] Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. *ArXiv*, abs/2205.02029, 2022.

[77] Yu Wang, Fengjuan Gao, and Linzhang Wang. Demystifying code summarization models. *CoRR*, abs/2102.04625, 2021. URL `https://arxiv.org/abs/2102.04625`.

[78] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685`.

[79] Hui-Hui Wei and Ming 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*, IJCAI'17, page 3034–3040. AAAI Press, 2017. ISBN 9780999241103.

[80] Michael Wick, Daniel Stevenson, and Paul Wagner. Using testing and junit across the curriculum. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, page 236–240, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139977. doi: 10.1145/1047344.1047427.

[81] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017. URL `http://arxiv.org/abs/1704.01696`.

[82] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL `http://arxiv.org/abs/1410.4615`.

[83] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019. doi: 10.1109/ICSE.2019.00086.

[84] Yang Zhang, Yunqing Xia, Yi Liu, and Wenmin Wang. Clustering sentences with density peaks for multi-document summarization. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1262–1267, Denver, Colorado, May–June 2015. Association for Computational Linguistics. doi: 10.3115/v1/N15-1136. URL `https://aclanthology.org/N15-1136`.

[85] Mengya Zheng, Xingyu Pan, and David Lillis. Codex: Source code plagiarism detection based on abstract syntax tree. In *AICS*, pages 362–373, 2018.