AUTOMATING CODE REVIEW FEEDBACK FOR STUDENT ASSIGNMENTS USING MACHINE LEARNING

A PROJECT REPORT

Submitted By

RAMAKRISHNAN A M 185001124
RISHI VARDHAN K 185001126
SACHIN KRISHAN T 185001128

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

Rajiv Gandhi Salai (OMR), Kalavakkam - 603110

May 2022

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this project report titled **AUTOMATING CODE REVIEW FEEDBACK FOR STUDENT ASSIGNMENTS USING MACHINE LEARNING** is the *bona fide* work of **RAMAKRISHNAN A M (185001124)**, **RISHI VARDHAN K (185001126)**, and **SACHIN KRISHAN T (185001128)** who carried out the project work under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Dr. T. T. MIRNALINEE
HEAD OF THE DEPARTMENT
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Dr. R. S. MILTON
SUPERVISOR
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on...........

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

We would like to express our gratitude to our guide, Professor **Dr. R. S. MILTON**, for his insightful ideas and suggestions, as well as his guidance, patience, and assistance in shaping and refining our work.

We would also like to thank **Dr. T. T. MIRNALINEE**, Professor and Head of the Department of Computer Science and Engineering, for her invaluable support and guidance. We would like to extend our gratidue to our project Coordinator **Dr. B. Bharathi**, Associate Professor, for her valuable suggestions during this project.

We also pay our respects to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. We also want to thank **Dr. V. E. ANNAMALAI**, Principal, for all of the help provided throughout this course of study.

We would like to express our heartfelt gratitude to all of our department's teaching and non-teaching staff members who have contributed directly or indirectly to our project work. Finally, we'd want to express our gratitude to our parents and friends for their patience, cooperation, and moral support over the years.

RAMAKRISHNAN A M **RISHI VARDHAN K** **SACHIN KRISHAN T**

ABSTRACT

Students learning to program would benefit greatly from specific feedback for each program submission in order to enhance their programming skills. However, evaluating student program submissions and reviewing each one of them is a time consuming and tedious task for instructors and teaching assistants. For these reasons, we propose an end-to-end pipeline that automatically examines program design as well as its functionality to provide appropriate specific feedback after evaluating the student code submission on a scale of one to ten. Three regression models viz Support Vector Regressor(SVR), Random Forest Regressor and Multi-Layer Perceptron (MLP) Regressor were trained with a dataset developed from a corpus of 480 Python programs submitted by students. In general, it was observed that the Random Forest Regressor performed better. When the program submissions are graded out of ten, the Random Forest Regressor model has a mean absolute error (MAE) of 1.7 and root mean square error (RMSE) of 2.3.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

BACKGROUND AND MOTIVATION

1.1 INTRODUCTION

Code Review frequently involves a Static Code Analysis (SCA), often known as Source Code Analysis. In general, static code analysis refers to the use of SCA tools to identify possible vulnerabilities in 'static' (non-running) source code. This allows to get a clear picture of the code's structure and can aid in ensuring that the code meets industry standards.

Static code analysis is used by software development and quality assurance teams to find potential vulnerabilities. The SCA tool will scan all of the code in a project for vulnerabilities while verifying it. Static code analysis is frequently successful in finding coding problems such programming mistakes, coding standards breaches, and security issues.

Static code analysis has several advantages including improving code quality by evaluating all of the code in an application. In comparison to manual code review, automated tools consume very little time. When static testing is paired with typical testing methods, more debugging depth is possible. With automated technologies, human mistake is less likely. It will increase online or application security by increasing the likelihood of detecting code flaws.

We propose to use Machine Learning to perform Static Code Analysis (SCA) on student coding assignments to evaluate the code and give valuable feedback to improve the quality of code. We propose to accomplish this by training a new pipeline on our manually annotated student assignment data set. In our study, the pipeline will produce a design value score ranging from 1 to 10, which will be utilised to give personalised feedback explaining the logic behind the predicted design value score.

As a result, because feedback is a crucial part of effective learning, the review or feedback provided helps students improve the quality of their code. According to research, in the context of student assignments, feedback is more strongly and consistently connected with achievement than any other teaching practice. This link exists regardless of grade, financial status, race, or educational setting. Students' self-esteem, self-awareness, and drive to learn can all benefit from feedback. This method of providing feedback to students has been proposed as a strategy to improve learning and evaluation performance.

1.2 MOTIVATION

Feedback is an essential component of scaffolding for learning as feedback assists students in achieving their learning goals and to improve their self-regulation capability. The findings show that participants were more engaged during the assessment when the feedback was valuable and the explanations were clear and helpful.

Especially in recent situtaions of online learning due the pandemic, feedback becomes even more critical since instructors and students are separated geographically. In this case, feedback enables the instructor to tailor learning content to the needs of the pupils. Giving feedback, on the other hand, is a difficult chore for instructors, especially in large groups. As a result, numerous automatic feedback systems have been proposed in order to lessen the instructor's labor.

Manually grading assignments is a time-consuming and error-prone process. Since grading is typically done for a large number of student submissions, there is a increased probability of errors. Artificial intelligence (AI) techniques can help to solve these problems by automating the grading process. Teachers can be assisted with corrections and can provide instant feedback to students, allowing them to enhance their programming assignment solutions before submitting their final submission.

Online education allows students to determine their own learning speed and has the extra benefit of allowing them to create a timetable that matches everyone's schedule. As a result, when students learn online, finding a good work-study balance is easy. Students in online classes who do not have access to a live teacher can still benefit from comments thanks to an aiding machine learning-based Static Code Analyser that provides individualised feedback.

CHAPTER 2

LITERATURE SURVEY

Predicting code characteristics or extracting relevant aspects from large volumes of code data has progressed significantly in recent years. Name prediction of program entities [19], code generation [11], code completion [20] and code summarization [12] depend on anticipating and determining program's attributes and features without program compilation or program execution. Representing the code suitable for a learning system to understand has been performed in two ways, either using Code Embeddings representation or using the AST (Abstract Syntax Tree) representation.

The goal of research in embeddings-based algorithms is to learn good representations of programs, comparison of source programs, and recommendation of ways for students to learn. David Azcona et al (2019) [3] tried to profile and cluster students based on their program submissions. Their work involved the comparison of many vectorization techniques of source programs to determine the correctness of a program submitted by a student. For research in AST representation, Mou et al [10] proposed a method for developing program vector using AST representation for use with Deep Learning models to classify computer programs.

In the field of providing feedback on code solutions, Piech et al [6] used Embeddings of programs to provide feedback to learners in Massive Online Open Courses (MOOC). They first learnt how to capture the functional and stylistic parts of program submissions of learners, and successively provided automatic feedback. This was accomplished by creating functionality matrices at each node in the submission's syntax tree. Paaben et al. [4] demonstrated that a continuous hint approach can anticipate what expert students would perform in a programming assignment of multiple steps, and that the edit cues provided by human instructors can be equaled by embedding-based clues. To offer feedback mechanisms and automatic example assignments, Gross et al [17] used structured solution spaces. Mou et al [9] introduced a tree-based Convolutional Neural Network (TBCNN) that captures structural information using a convolution kernel constructed over program ASTs. This method was also used to classify programs based on their functionality and to detect code snippets that matched specified patterns. Furthermore, Proksch et al [18] showed for C# programs by constructing a collection of syntax trees which were utilised for suggestions using solutions from GitHub.

To enhance the design of student programs, J Walker Orr et al [14] designed a system that provided individualised feedback for pupils. Their focus was only on the design quality of programs. They annotated the student submissions with a design score between 0 and 1 and considered the programs with design score over 0.75 as good programs. Around 40 features were extracted from the code's AST representation. Feedback was generated by comparing the feature vector of a code in question with the average feature vector of good programs.

In our work, we propose a feedback generation system that leverages the use of AST representation along with a code's functionality and design to provide constructive feedback to student code submissions.

CHAPTER 3

DATA PREPROCESSING

For experimentation, the collection of Python programs compiled by the Dublin City University [3] on student code solutions for Python assignments over 3 years was selected. Their data collection technique involved students submitting their solutions to an online grading platform where an auto grader reports as correct (Class 1) or as incorrect (Class 0), based on success and failure of test cases, respectively. Although this information is invaluable to instructors, we aim to improve the representation of programs and consequently improve the necessary recommendation provided. Attributing to the vast size of the dataset and basing our task to an educational context where student assignments are constantly being updated, we considered a subset of 4 different programming questions to provide constructive feedback. The shortlisted programs and descriptions for each are listed in Table 3.1. For each task, 120 different code solutions were randomly selected. Overall, for the 4 different questions, 480 code solutions were compiled for experimentation.

TABLE 3.1: Questions and Description

Questions	**Description**
Selection sort	Returns a sorted list from an unordered list
First negative element in a list	Returns the first negative element in a list
Unique characters count in a string	Returns a histogram of character counts in a string
Largest element of a list	Returns the largest element in a list

3.1 DATASET ANNOTATION

To model the dataset for the task of regression, we need continuous values for the target variable. Considering the derived dataset has categorical values (0 & 1) for the target variable, we devised a grading rubric to assign numeric scores to respective code records. The numeric scores are values in the range from 0 to 10 and they represent the goodness of the code solutions to the particular problem. Accordingly, higher numeric score represents a correct and better code solution to the problem and lower numeric score represents an incorrect or non-optimal code solution. Figure 3.1 represents the flowchart architecture for dataset annotation.

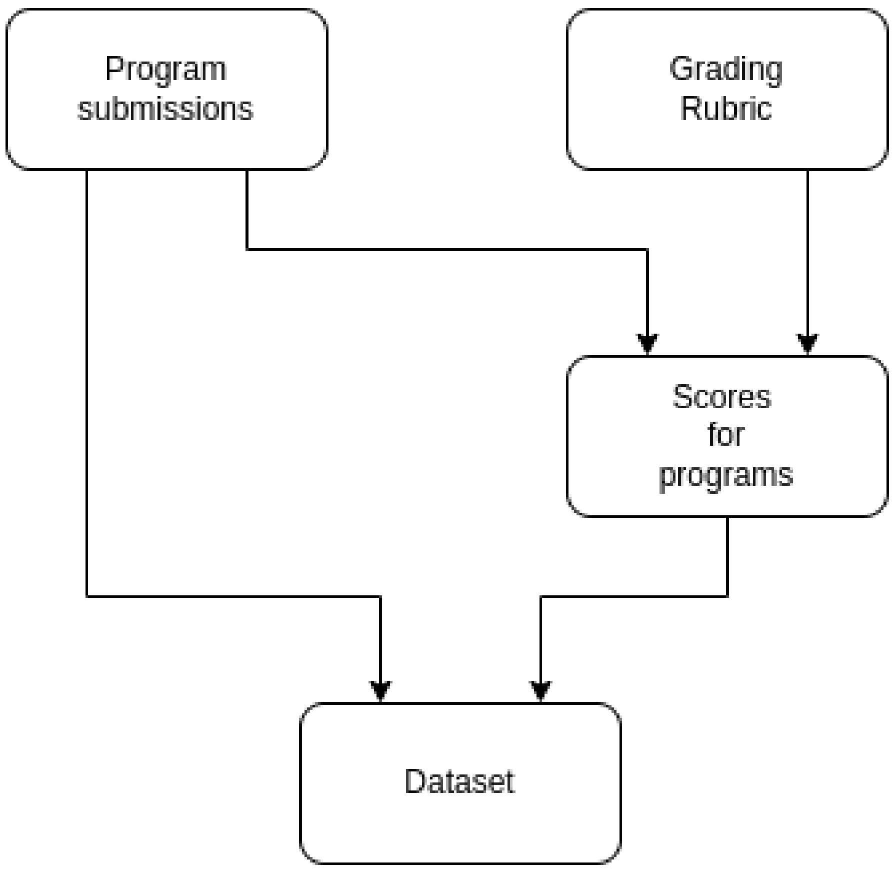


FIGURE 3.1: Data Annotation Flow

The score component for a value of 10 points was divided into two parts namely **Design** (5 points) and **Functionality** (5 points). The subsequent pattern managed to cover the entirety of a code solution to provide a value balancing the code's design and functionality.

For design, considering each problem has its own set of parameters to satisfy, the grading rubric is adjusted based on this requirement. The parameters used for evaluating design is mentioned in Table 3.2. Tables 3.3 to 3.6 details the appropriate score to respective problems based on their necessary individual criteria for each different task.

TABLE 3.2: Grading Parameters

Parameters
Use of Variables
Modularity
Logic Application
Efficiency

- **Selection Sort**

TABLE 3.3: Grading Rubric (Design) for Selection Sort

Parameter	**Weightage**
Code modularity	1
Use of variables	1
The logic applied	1
Efficient use of functions	2

For "Selection Sort" task, three functions had to be defined: one for swapping, another to find the smallest element of the list from a given index, and yet another function for sorting which will use the other two functions. Hence, the definition of functions and use of function calls becomes the important parameter for design consideration.

- **First Negative Element in List**

TABLE 3.4: Grading Rubric (Design) for First Negative Element in List

Parameter	Weightage
Use of variables	1
The logic applied	2
Efficiency of solution	2

For "First Negative Element in List" task, there was no need of functions as it only requires a simple looping statement and a conditional statement to find the first negative element in the list. The important design parameter now was the logic applied to find the first negative element and the efficiency of the solution (for instance, the use of break statement after finding the first negative element).

- **Largest Element in the List**

TABLE 3.5: Grading Rubric (Design) for Largest Element in List task

Parameter	Weightage
Use of variables	2
The logic applied	1
Efficiency of solution	2

For “Largest Element in the List” task, there was no need of functions again as it only requires a simple looping statement and a conditional statement to find and print the index of the largest element in the list. Here, the definition and use of variables was the important parameter along with the efficiency.

- **Unique Character Count in String**

TABLE 3.6: Grading Rubric (Design) for Unique Character Count in string

Parameter	**Weightage**
Use of variables	1
The logic applied	1
Code Modularity	1
Efficiency of solution	2

For “Unique Character Count in String” task, functions can be defined to check and ignore special characters, a function could be defined to count the unique tokens, etc. Thus, the code could be made modular. Since this particular task could be solved in a variety of ways, the efficiency of the solution matters more.

For **functionality**, the following rubric mentioned in Table 3.7 was followed to give a score in a scale of 5. The correctness and completeness of the code was considered to grade the code submissions.

TABLE 3.7: Grading Rubric (Functionality)

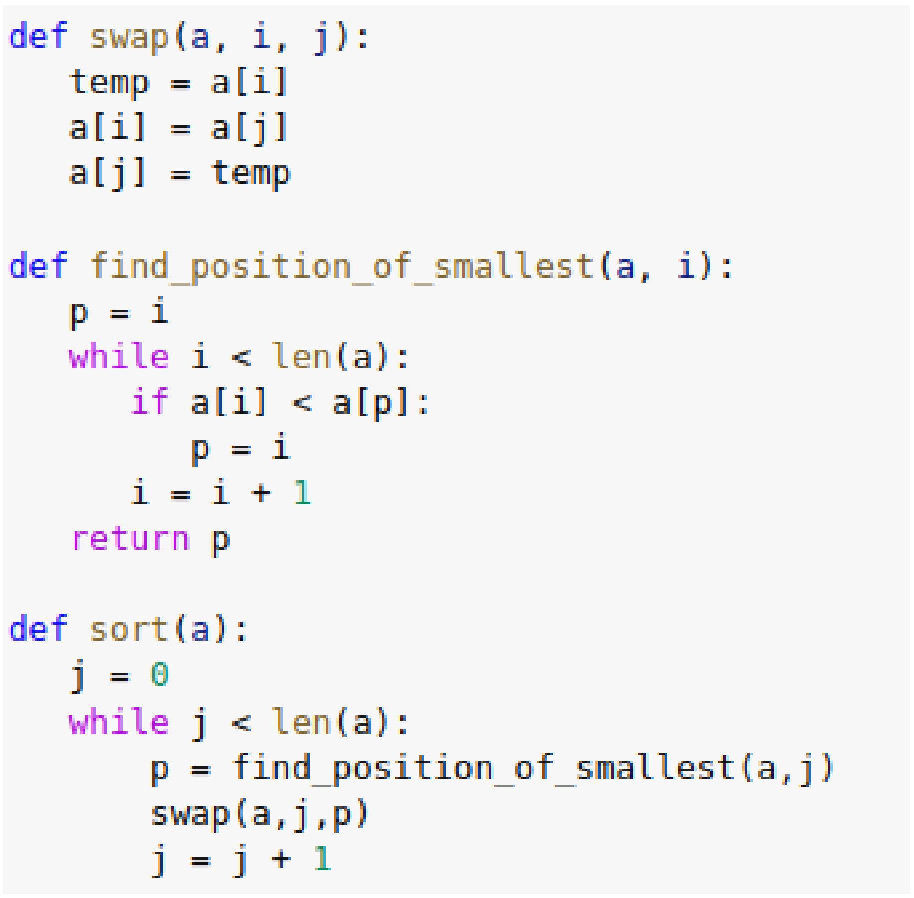
Code's correctness and completeness	**Score**
Perfectly correct and complete	5
Correct and complete but can be improved	4
Complete but partially incorrect	3
Incomplete and partially correct	2
Incomplete and totally incorrect	1

Finally, a dataset compiled of code solutions and appropriate numeric score values (in the range from 0 to 10) was generated to be used for the subsequent experimentation.

3.1.1 Example Code Submissions

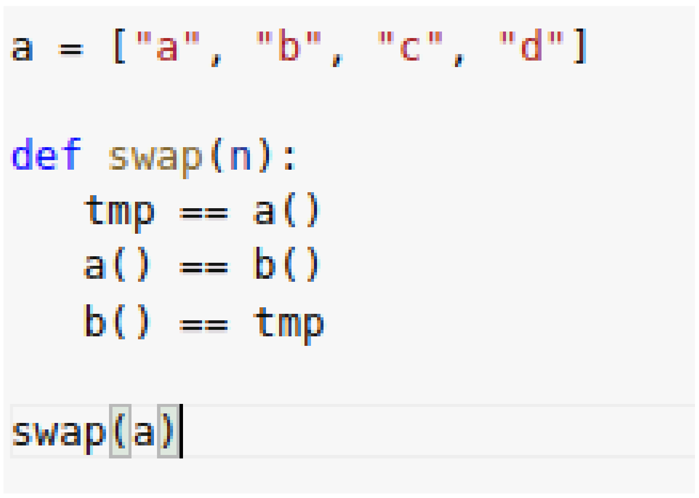
Figures 3.2 to 3.9 document the examples of best and worst programs for each problem with scores assigned using the developed grading rubric.

3.1.1.1 Selection Sort



```
def swap(a, i, j):
   temp = a[i]
   a[i] = a[j]
   a[j] = temp

def find_position_of_smallest(a, i):
   p = i
   while i < len(a):
      if a[i] < a[p]:
         p = i
      i = i + 1
   return p

def sort(a):
   j = 0
   while j < len(a):
       p = find_position_of_smallest(a,j)
       swap(a,j,p)
       j = j + 1
```

FIGURE 3.2: Best Program - Selection Sort



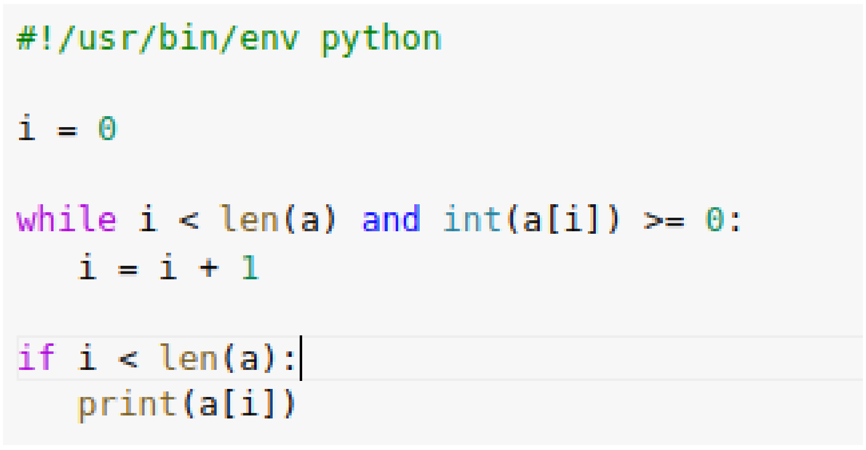
```
a = ["a", "b", "c", "d"]

def swap(n):
    tmp == a()
    a() == b()
    b() == tmp

swap(a)
```

FIGURE 3.3: Worst Program - Selection Sort

Figures 3.2 is one of the best program submissions for the selection sort task. The program was properly structured with three functions, one to do swapping, one to find smallest element's position from a given index from the list, and finally the sort function which will utilise the first two functions inside a loop to sort the list. Hence, it was awarded a perfect 10/10.

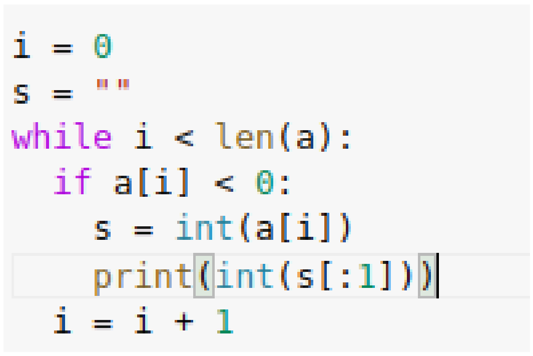
Figure 3.3 is one of the worst program submissions. This code solution has a swap function defined but with wrong logic. A unnecessary global variable 'a' was declared. The logic of the swap function was not right. Hence this solution was graded 1/10.

3.1.1.2 First negative element in a list



```
#!/usr/bin/env python

i = 0

while i < len(a) and int(a[i]) >= 0:
   i = i + 1

if i < len(a):
   print(a[i])
```

FIGURE 3.4: Best Program - First negative element in a list



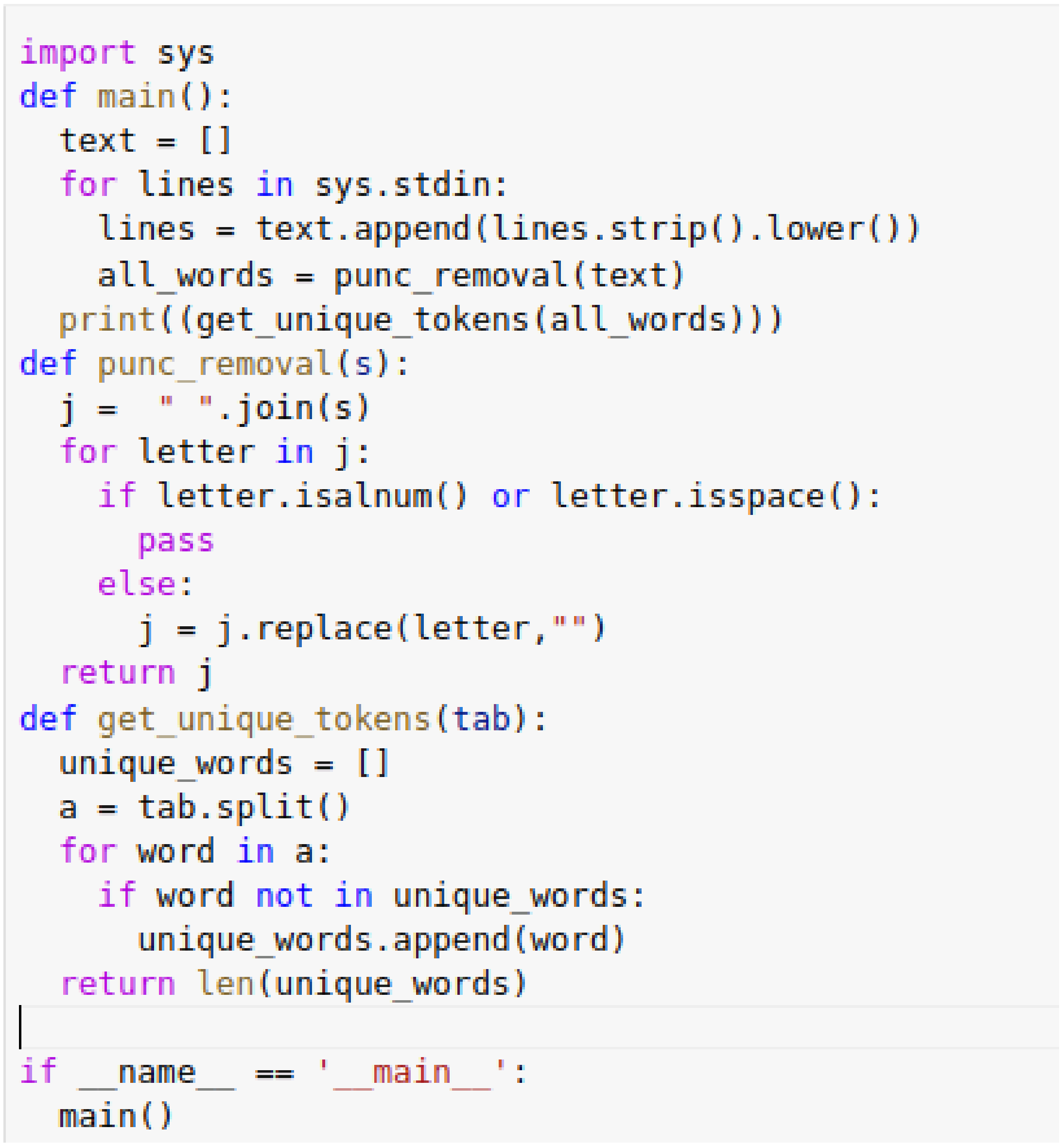
```
i = 0
s = ""
while i < len(a):
  if a[i] < 0:
    s = int(a[i])
    print(int(s[:1]))
  i = i + 1
```

FIGURE 3.5: Worst Program - First negative element in a list

Figure 3.4 shows a good solution for finding the first negative element in the list. The loop traverses the list until it finds the first negative element. When that index belongs to the list, the program prints the first negative element. The program is graded a perfect 10/10.

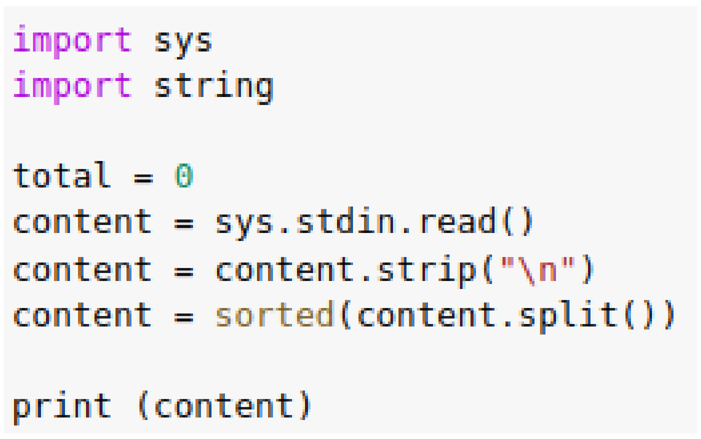
Figure 3.5 shows one of the bad code solutions submitted. The program is graded only 3/10 for using wrong logic and also for improper declaration and use of variables.

3.1.1.3 Unique characters count in a string



```python
import sys
def main():
  text = []
  for lines in sys.stdin:
    lines = text.append(lines.strip().lower())
    all_words = punc_removal(text)
  print((get_unique_tokens(all_words)))
def punc_removal(s):
  j =  " ".join(s)
  for letter in j:
    if letter.isalnum() or letter.isspace():
      pass
    else:
      j = j.replace(letter,"")
  return j
def get_unique_tokens(tab):
  unique_words = []
  a = tab.split()
  for word in a:
    if word not in unique_words:
      unique_words.append(word)
  return len(unique_words)

if __name__ == '__main__':
  main()
```

FIGURE 3.6: Best Program - Unique characters count in a string



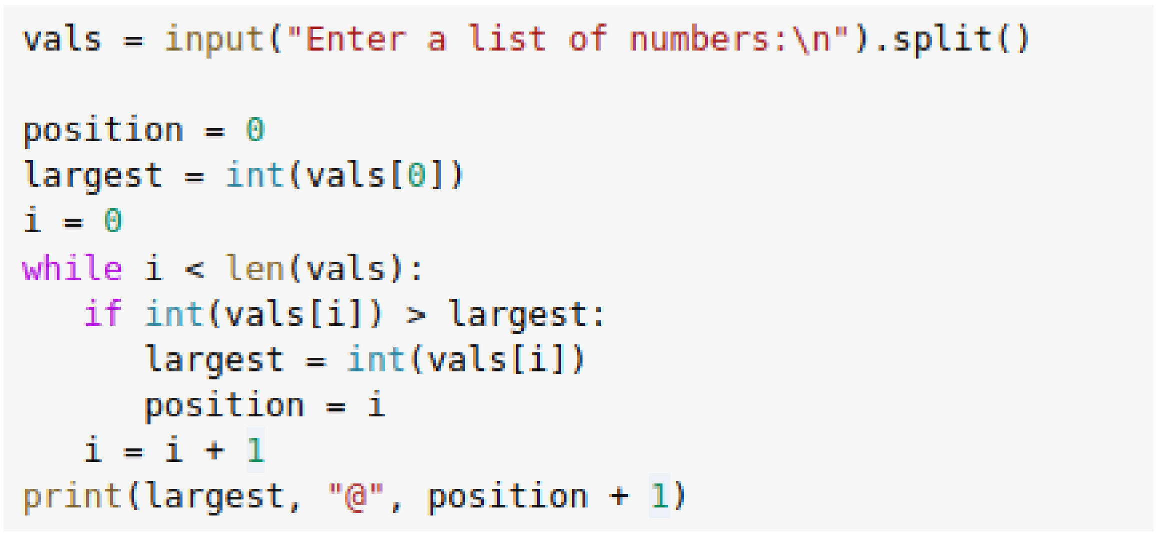
```
import sys
import string

total = 0
content = sys.stdin.read()
content = content.strip("\n")
content = sorted(content.split())

print (content)
```

FIGURE 3.7: Worst Program - Unique characters count in a string

Figure 3.6 is one of the best program submissions for the unique character count task. The code is properly organised, modular with three functions and the code has perfect logic. No unnecessary global variable is defined. Thus, this particular submission was awarded with the perfect score of 10.

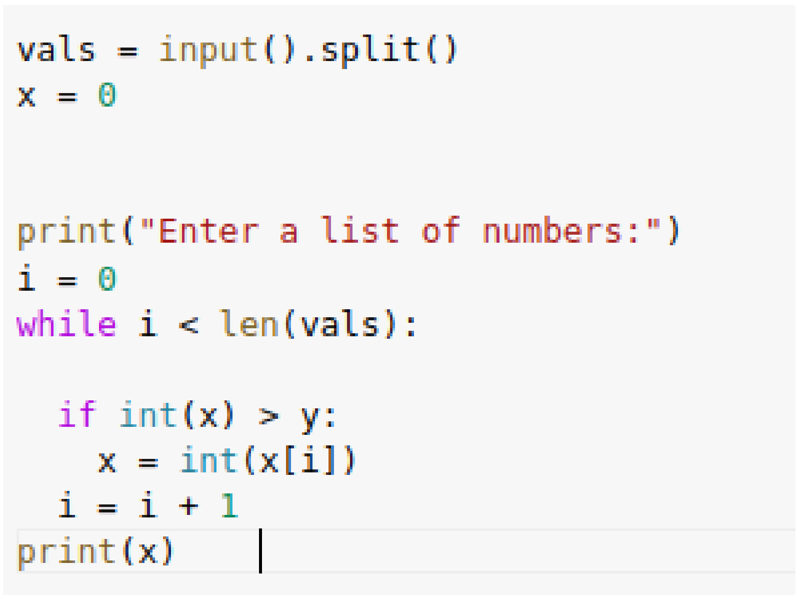
Figure 3.7 on the other hand shows one of the worst submissions for this particular task. No functions have been defined. An unnecessary global variable was declared. The code would not yield the expected result. Thus, this code was only given a score of 1.

3.1.1.4 Largest element of a list



```python
vals = input("Enter a list of numbers:\n").split()

position = 0
largest = int(vals[0])
i = 0
while i < len(vals):
   if int(vals[i]) > largest:
      largest = int(vals[i])
      position = i
   i = i + 1
print(largest, "@", position + 1)
```

FIGURE 3.8: Best Program - Largest element of a list



```python
vals = input().split()
x = 0


print("Enter a list of numbers:")
i = 0
while i < len(vals):

  if int(x) > y:
    x = int(x[i])
  i = i + 1
print(x)
```

FIGURE 3.9: Worst Program - Largest element of a list

Figure 3.8 is one of the best student code submissions for the largest element task. This particular solution has the right logic. Variables have been used properly. The program prints the correct position of the largest element in the list. Looping and conditional statements have been used correctly. Thus, this solution was awarded as score of 10.

Figure 3.9 is one of the worst student submissions for this task. Variables have not been properly used. The logic used is also wrong. Hence, this code solution was only given a score of 2.

3.2 FEATURE EXTRACTION

Following the annotation of the dataset, feature extraction and feature selection are performed for optimal representation of code solutions as vectors for regression. The following series of steps were performed for extracting meaningful features from the dataset.

- **Py2 to Py3 conversion**: The code solutions in the dataset are documented in Python 2 version. To support extraction of features, Python 2 to Python 3 conversion was performed using the '2to3' module [1].
- **Abstract Syntax Tree (AST) representation**: After converting every program to Python 3 format, the AST [2] representation for each program was computed to derive appropriate features. The inbuilt Python 'ast' module was used to get the ast representation of the student submissions. The 'ast' module provides a 'parse()' method which returns the AST representation of a program. A variety of features can be extracted from this ast representation with the help of 'walk()' function provided by the 'ast' module. The extracted features from the AST representation are documented in Table 3.9. Other features derived from code are represented in Table 3.8.

TABLE 3.8: Features extracted from code

return	Count of ’return’ statements in the code
break	Count of ’break’ statements in the code
continue	Count of ’continue’ statements in the code
pass	Count of ’pass’ statements in the code
assign	Count of assignment operators in the code
arith	Count of arithmetic operators in the code
comp	Count of comparison operators in the code
log	Count of logical operators in the code
cond	Count of conditional operators in the code
loop	Count of ’for’ and ’while’ loops in the code
#lines	Count of lines in the code

TABLE 3.9: Features extracted from AST

#fun	Count of function definitions in the code
#fcall	Count of function calls in the code
globals	Count of globals variables in the code
#lst	Count of lists and tuples in the code
#var	Count of the variables in the code (Count of variables defined)
#var_uses	Count of variable uses in the code (Count of variables used in different statements)

Ultimately we extracted 17 features(11 from source code and 6 from AST representation of source code).

3.3 FEATURE SELECTION

After extracting meaningful features from both the code and AST representation (Table 3.8, Table 3.9), feature selection is performed for each problem considering feature sets for each problem are independent of each other and are problem dependent.

Firstly, the features that share the same value for all the code solutions are identified and removed to avoid redundancy. Once the redundant features are reduced, the number of features eventually reduces from

- 17 to 14 for 'Selection sort'
- 17 to 12 for 'The First negative element in the list'
- 17 to 12 for 'The Largest Element in List'
- 17 to 15 for 'The unique character count in a string task.'

Then, the best features are selected for each task according to how well they are correlated with the target variable. The *f_regression* method [8] returns the degree of correlation (F-statistic) between each feature and the target.

The correlation F-statistic score of different features for the tasks are listed as tables below. After the F-statistic score is calculated, the features which have significant influence on the target variable(score) are selected and the regressor model for each task is trained with the selected features.

3.3.1 Implementation

Tables 3.10 to 3.14 report the F-Statistic score of the features set with their respective scores. The strike-through text in features list of the tables represent the features that are dropped due to their low $freg$ score.

- **Selection Sort**

TABLE 3.10: F-statistic table : Selection Sort

Feature	$freg$ **score**
#fun	387.16
loop	181.29
#fcall	179.15
#var_uses	122.33
comp	104.35
#lines	91.83
return	91.55
arith	68.54
assign	65.93
#var	56.68
cond	50.10
#lst	20.39
globals	15.75
~~log~~	1.81

From Table 3.10, it can be observed that for 'Selection sort' task, the number of functions, loops, function calls, variable uses and comparison operators have a very strong correlation with the output score. The first 13 features have a considerably big influence on the target variable when compared to that of number of logical operators.

- **First Negative Element in List**

TABLE 3.11: F-statistic table : First Negative Element in List

Feature	*freg* **score**
comp	31.42
#var_uses	6.33
cond	6.02
log	2.11
assign	2.02
loop	1.85
break	1.49
#lines	1.47
#var	1.32
#lst	1.01
~~globals~~	0.50
~~arith~~	0.05

From Table 3.11, it is observed that for the 'First negative element in the list' task, the number of comparison statements have a large correlation on the output variable. The number of variable uses and the number of conditional statements also have a good influence with the target variable. Here with a minimum cutoff of a F-reg score of 1, top 10 features become the selected features.

- **Largest Element in List**

TABLE 3.12: F-statistic table : Largest Element in List

Feature	*freg* **score**
assign	17.51
arith	15.91
#var_uses	12.47
globals	9.47
#lst	3.82
comp	3.35
cond	3.35
#var	2.49
~~loop~~	0.90
~~break~~	0.84
~~#lines~~	0.73
~~log~~	0.02

From Table 3.12, it is observed for the 'Largest element in the list' task, the number of assignment statements, arithmetic operators and variable uses have a great influence on the final target value. With a minimum cutoff of a F-reg score 1.0 the top 8 features are selected.

- **Unique Character Count in List**

TABLE 3.13: F-statistic table : Unique Character Count in List

Feature	*freg* **score**
log	15.76
#fcall	4.94
#var	3.58
cond	3.52
pass	3.42
#var_uses	3.22
#lst	2.91
loop	1.91
~~return~~	0.95
~~comp~~	0.78
~~#fun~~	0.46
~~#lines~~	0.34
~~globals~~	0.20
~~arith~~	0.17
~~assign~~	0.05

From Table 3.13, it is observed for the unique character count in a string, the number of logical operators has the highest influence on the final target output. This is due to the number of 'not' and 'not in' operators. With a minimum cutoff set at a F-reg score of 1.0, only the first eight features are selected.

CHAPTER 4

DESIGN AND ARCHITECTURE

We propose a machine learning based student program evaluation system which represents student programs as a set of features extracted from the source programs and the AST representation of the source programs. Deep learning is purposely avoided since it would make the program's representation difficult to comprehend. The ML models are regression models which will output a score between 0 to 10 for each program submission. Feedback is provided to each program submission based on comparing the feature vector of the program with the average feature vector of 'good' programs.

4.1 TRAINING

The dataset contains the code and corresponding scores for each of the student programs. Features from code were extracted by using two ways. Some features were extracted directly from the code. For the other features, the code was first transformed into an Abstract Syntax Tree(AST); then the features were extracted from the AST. The extracted features were used to train the model. Three models were trained separately: Linear Support vector regressor(SVR) , Random forest and Multi layer perceptron(MLP) regressor. The models trained were all regression models which were trained to predict the scores that should be given to the evaluated codes.

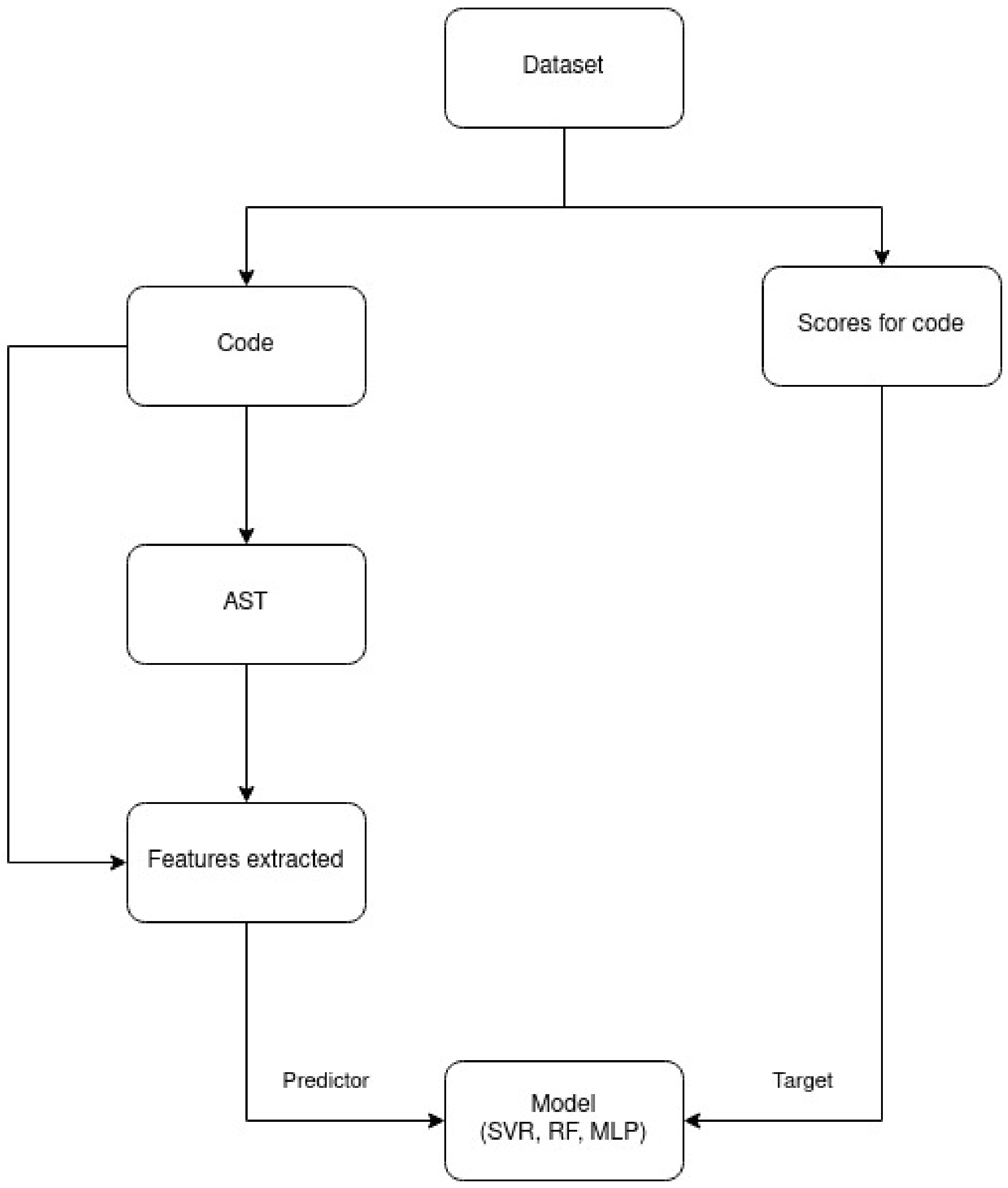


FIGURE 4.1: Training

4.2 DEPLOYMENT

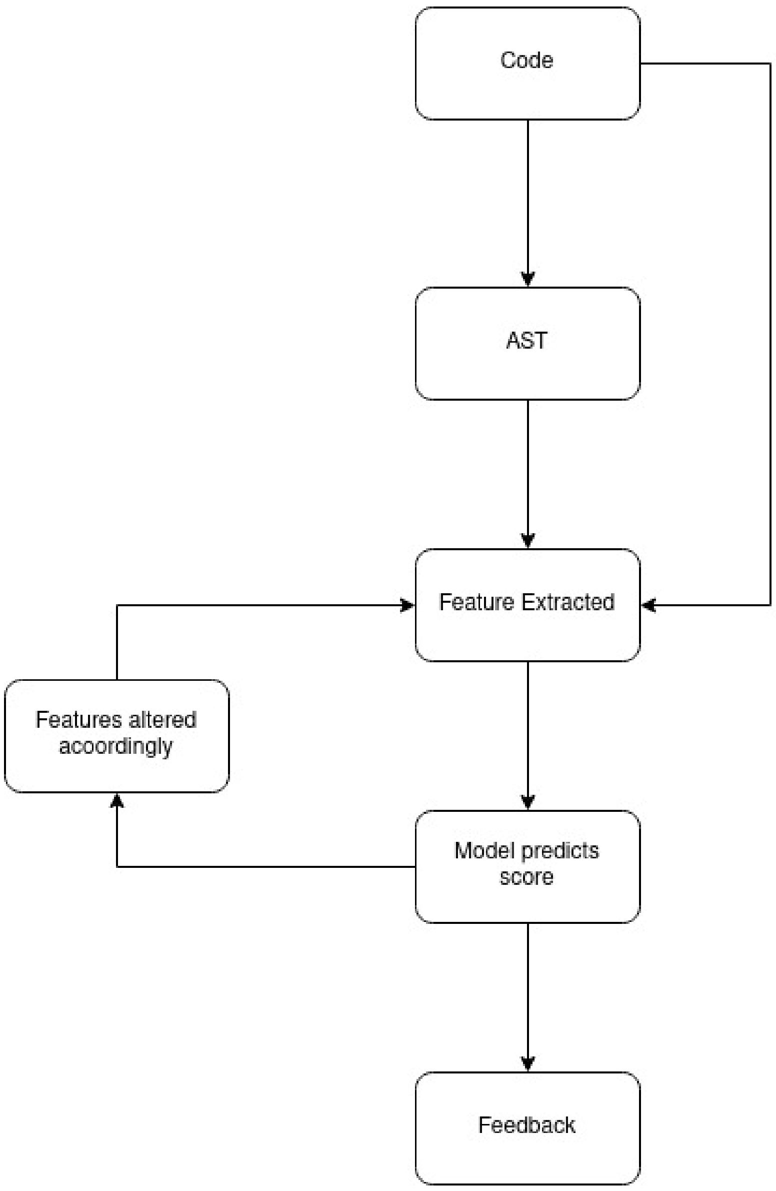


FIGURE 4.2: Deployment

The student code is the input. Features are extracted from the code as explained in the section above. Some features are extracted from the code itself and the rest from an AST representation of the code. This feature set is input into the trained models from the previous section. The model predicts a score for the input code based on the feature set.

As explained earlier, our model is trained only with the feature values of the program and its score. Our trained model will evaluate how modifications in program's features can lead to a better score and thus there is no necessity for a explicitly annotated dataset with feedbacks for each code.

From the training dataset, average feature vector x is computed for the ”excellent” programs(programs with a perfect score of 10). Feedback is generated with correspondence to the difference between the feature values of the submitted program and the average feature vector of the excellent programs.

4.3 ALGORITHMS

After representing code solutions as feature vectors, the dataset was exported to perform training for machine learning models. The following models

- Support Vector Machine (SVM)
- Random Forest
- Multi Layer Perceptron (MLP)

were employed for training. The detailed description of the mentioned algorithms are listed below.

4.4 SUPPORT VECTOR MACHINE

Support Vector Machines (SVMs) are supervised learning models with associated learning algorithms that examine data for classification and regression in machine learning. Support Vector Regression [7] is used to predict continuous values. The Support Vector Machine and Support Vector Regression are both based on the same premise (SVM). The goal of a support vector machine method is to find a hyperplane in an n-dimensional space that categorises data points clearly. Support Vectors are the data points on each side of the hyperplane that are closest to the hyperplane. These have an effect on the hyperplane's location and orientation, and so aid in the construction of the SVM.

4.4.1 Hyperparameters in SVR

Following are the various key hyper parameters used in Support Vector Regression.

- **Hyperplane** (Red line in Figure 4.3)
  A decision boundary that forecasts continuous output is known as a hyperplane. Support Vectors are the data points on each side of the hyperplane that are closest to the hyperplane. These are used to draw the essential line that depicts the algorithm's projected outcome.

- **Kernel**
  A kernel is a collection of mathematical functions that take input data and transform it into the required form. These are most commonly employed to locate a hyperplane in higher-dimensional space.

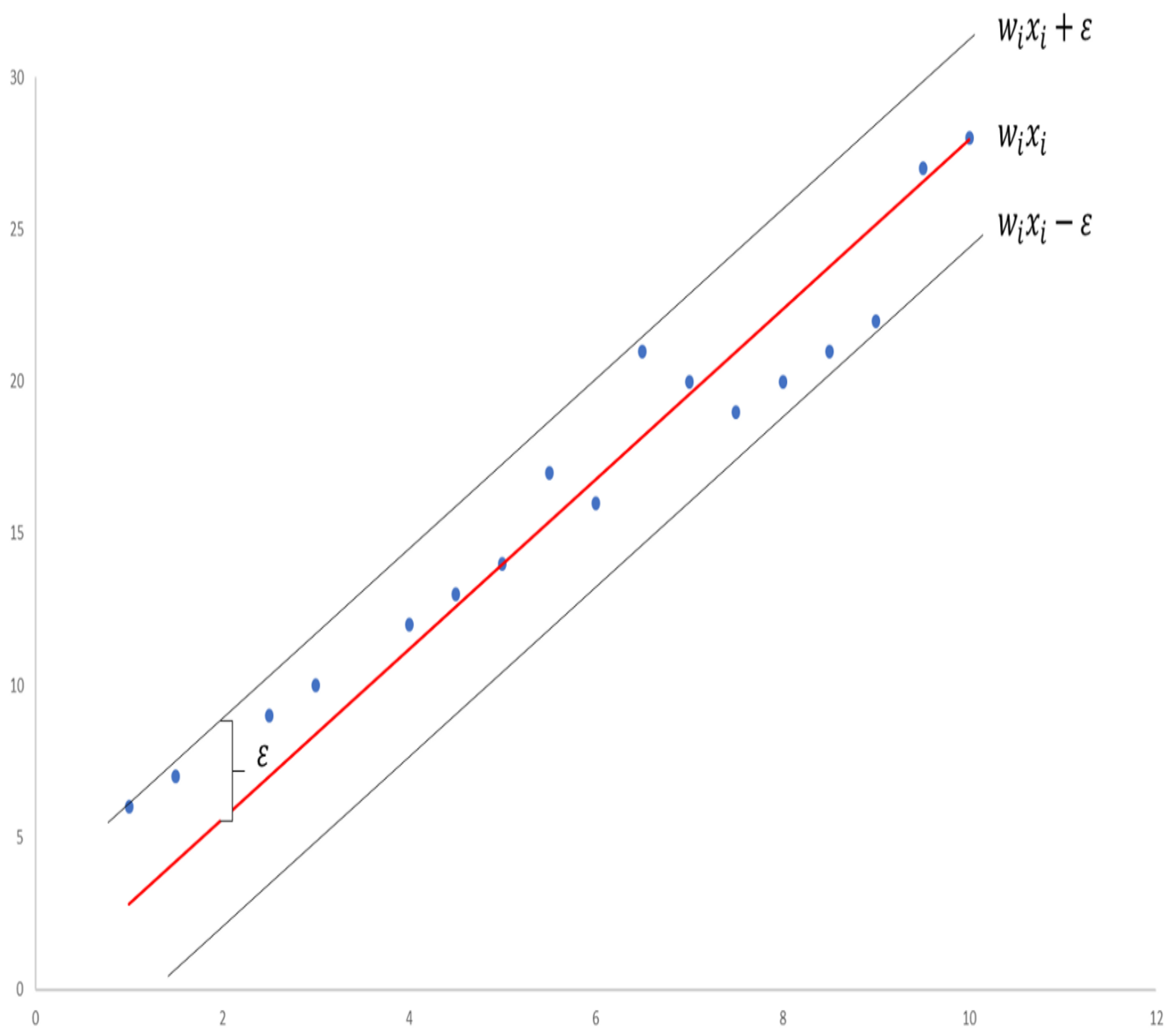


FIGURE 4.3: Simple SVR

- **Boundary Lines** (Grey lines in Figure 4.3)

  These are the two lines that are drawn at an ε (epsilon) distance around the hyperplane. It's used to separate the data points by a margin.

4.4.2 Support Vector Regression

The best fit line or hyperplane with the most points is found using Support Vector Regression (SVR). The SVR, unlike other regression models, aims to fit the best line within a threshold value, rather than minimising the error between the real and projected value (the distance between the hyperplane and boundary line). As a result, we'll only consider points that fall inside the decision boundary and have the lowest error rate, or those that fall within the margin of tolerance. This results in a more accurate model.

4.5 RANDOM FOREST

Random forest is a frequently used supervised machine learning algorithm for classification and regression tasks. The functioning of Random Forest is contingent on three main factors,

- Decision Trees
- Ensemble Learning
- Bootstrapping

The following subsections detail on the mentioned factors description.

4.5.1 Ensemble Learning

Ensemble learning is the process of combining many models that have been trained on the same data and averaging their findings to provide a more accurate predictive/classification result. Ensemble learning requires that the errors of each model (a decision tree model) be independent and distinct from one another.

4.5.2 Decision Tree

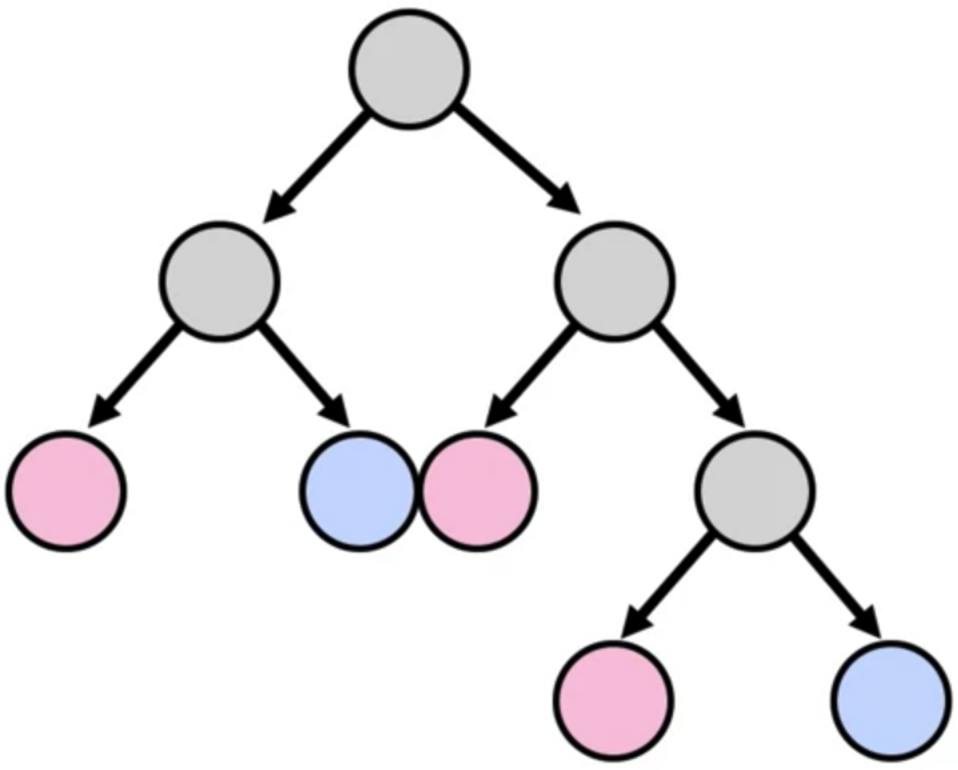


FIGURE 4.4: Decision tree layout

Both regression and classification issues can be solved using decision trees. The objective is to learn basic decision rules from data attributes to develop a model that predicts the value of a target variable. They begin at the root of the tree and go via splits depending on varied outcomes until they reach a leaf node where the result is delivered.Figure 4.4 details a visual representation of the decision tree layout.

4.5.3 Bootstrapping

The process of randomly sampling subsets of a dataset across a specified number of repetitions and variables is known as bootstrapping. To get a more accurate result, these results are averaged together.

4.5.4 Random Forest Regression

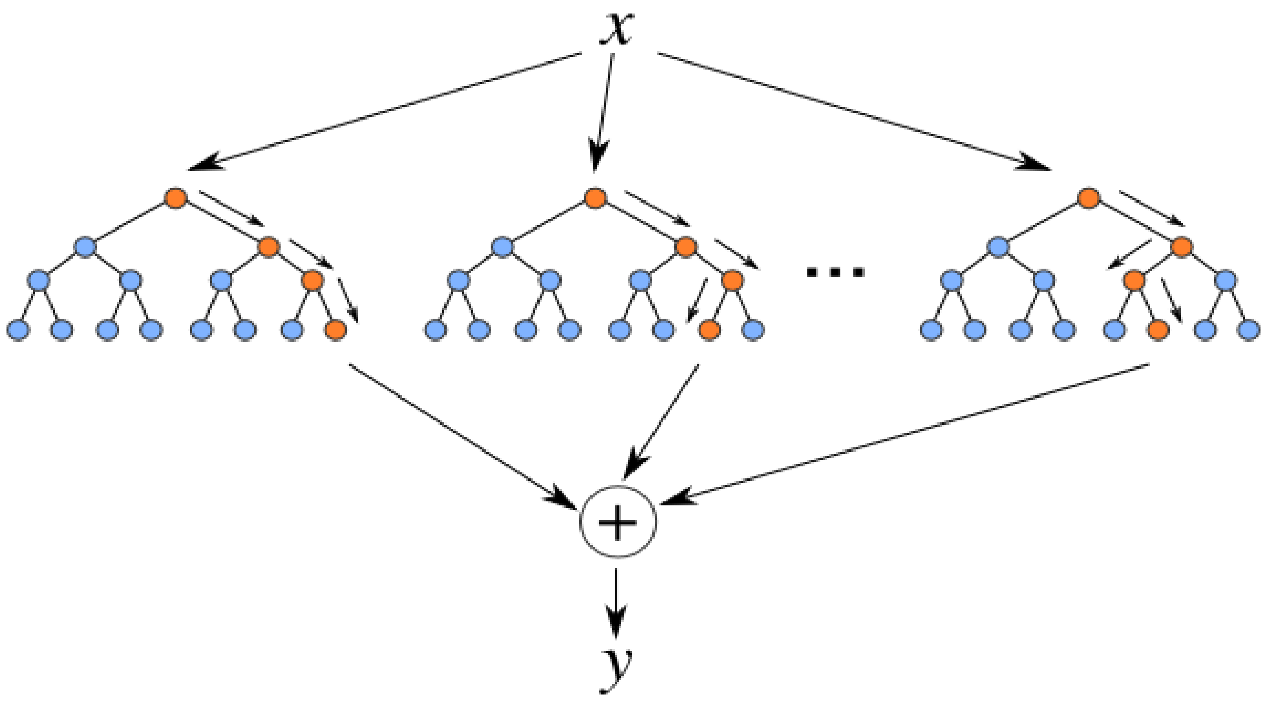


FIGURE 4.5: Simple RF

The bootstrapping Random Forest algorithm [5] combines ensemble learning techniques with the decision tree framework to generate numerous randomly drawn decision trees from data, then averaging the results to produce a new result that typically leads to better predictions/classifications. Figure 4.5 documents a visual representation of random forest wherein X is the feature vector and y is the target vector.

4.6 MULTI LAYER PERCEPTRON

The Multi Layer Perceptron bases its fundamental design to the interlinking of several neuron-like structures representing a Neural Network (NN) architecture. Given $i = 0, 1, \ldots, n$ where n is the number of inputs, the quantities w_i are the weights of the neuron. The inputs x_i correspond to features or variables and the output y to their prediction/estimation. Figure 4.6 shows the simplified representation of the above steps.

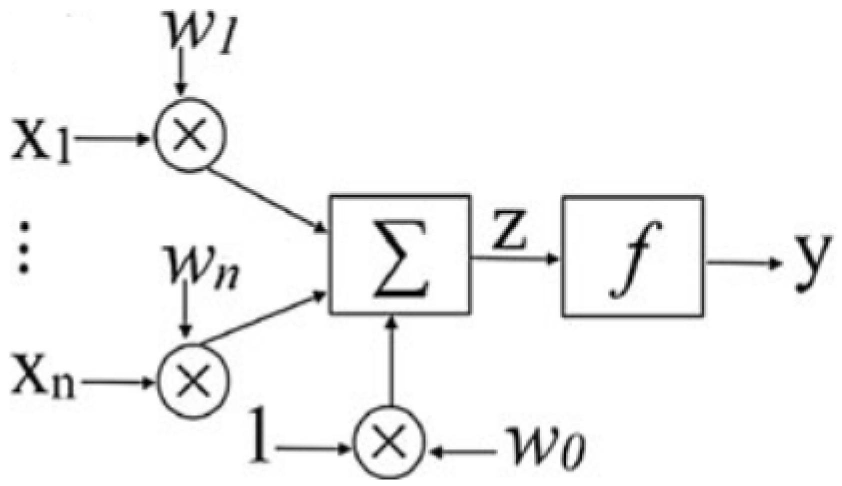


FIGURE 4.6: Perceptron model

The weighting step involves the multiplication of each input feature value by its weight $w_i x_i$ and then they are summed together $(x_0 w_0 + x_1 w_1 + ... + x_n w_n)$. In the transfer step, an activation function f (also called a transfer function) is applied to the sum producing an output y presented as

$$z = \sum_{i=0}^{n} w_i x_i \tag{4.1}$$

$$y = f(z) \tag{4.2}$$

wherein $x_0 = 0$, w_0 is the bias and y is the output. The activation function can be of the form of Unit step, Linear or Logistic operation.

For n dimensions, the function is a hyperplane with equation:

$$\sum_{i=0}^{n} w_i x_i = 0 \tag{4.3}$$

The goal of learning is to minimise a cost function, which is commonly a square error between the known and estimated vectors, in order to optimise the weights. The best weight vector can be determined using optimization techniques such as the gradient descent algorithm. The algorithm eventually converges on a solution that reaches an operational configuration network. The perceptron and single layer perceptron, on the other hand, do not address the nonlinearly separable problem.

To solve this problem, Multi Layer Perceptron (MLP) architecture is created by aggregating layers of perceptrons wherein the output of one layer acts as the input of another layer. Multi Layer Perceptron [13] is a feedforward neural network that consists of three layers, the input layer, the hidden layer and the output layer.

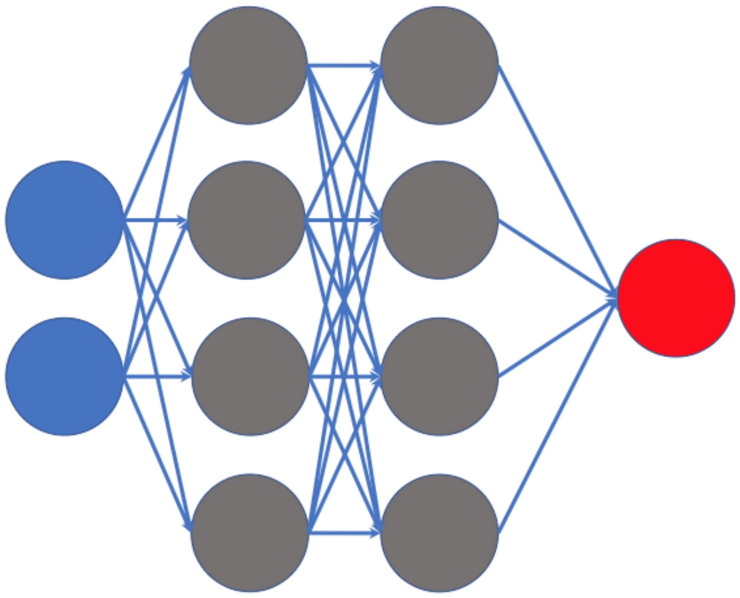


FIGURE 4.7: MLP model

Figure 4.7 presents an MLP with two inputs (blue), two hidden layers (grey) and one output (red).

The input signal to be processed is received by the input layer. The output layer is responsible for tasks such as prediction and categorization. The true computational engine of the MLP is an arbitrary number of hidden layers inserted between the input and output layers. In an MLP, data flows from input to output layer in the forward direction, similar to a feed forward network. The weights of the neurons in the MLP can be adjusted by propagating errors from layer to layer, starting with the output layer and going backwards, using the back propagation learning process. MLPs can tackle issues that aren't linearly separable and are designed to approximate any continuous function.

CHAPTER 5

IMPLEMENTATION

Implementation of the models mentioned in Chapter 4 was performed using the Scikit-Learn library [15]. For each code submission task, the dataset was split in the ratio 70:30 for training and testing, with respective hyper-parameters initialized for the individual models,

- The MLP regressor model was initialised with a single hidden layer with 100 neurons. The maximum iterations parameter was set to 5000.
- The Random Forest regressor was initialised with 100 estimators (decision trees).
- The Support Vector Regressor was initialised with linear kernel.

5.1 METRICS

Mean Absolute Error (MAE): MAE is the mean of magnitude of difference between true value t_i and prediction y_i of n observations.

$$MAE = \frac{\sum_{i=1}^{n} |t_i - y_i|}{n} \tag{5.1}$$

Root Mean Square Error (RMSE): RMSE is defined as the square root of the mean of residuals (difference between between true value 't_i' and prediction 'y_i') of 'n' observations

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{n} |t_i - y_i|^2}{n}} \tag{5.2}$$

5.2 RESULTS

The observed values of MAE and RMSE obtained for the different sets of tasks are reported in subsections 7.2.1 to 7.2.4. The values enclosed in brackets report scores observed with feature selection and the values not enclosed in brackets report scores observed without feature selection.

5.2.1 Selection Sort

TABLE 5.1: Results - Selection Sort

Model	***MAE***	***RMSE***
SVM	1.35 **(1.27)**	1.94 **(1.84)**
MLP	2.18 **(3.29)**	2.60 **(3.61)**
RF	1.18 **(1.17)**	1.87 **(1.83)**

From Table 7.1, we observe that Random Forest outperforms SVM and MLP in performance.

5.2.2 First Negative Item in List

From Table 7.2, we observe that Random Forest outperforms SVM and MLP in performance.

TABLE 5.2: Results - First Negative Item in List

Model	***MAE***	***RMSE***
SVM	2.13 (**2.11**)	2.97 (**2.99**)
MLP	3.02 (**3.13**)	3.80 (**3.90**)
RF	1.60 (**1.64**)	2.15 (**2.20**)

5.2.3 Largest Item in List

TABLE 5.3: Results - Largest Item in List

Model	***MAE***	***RMSE***
SVM	2.19 (**1.89**)	2.85 (**2.35**)
MLP	2.73 (**3.28**)	3.61 (**4.23**)
RF	1.51 (**1.47**)	2.10 (**2.07**)

From Table 7.3, we observe that Random Forest outperforms SVM and MLP in performance.

5.2.4 Unique Character count in a string

TABLE 5.4: Results - Unique Characters count in a string

Model	***MAE***	***RMSE***
SVM	2.63 (**2.68**)	3.19 (**3.23**)
MLP	3.11 (**2.71**)	3.93 (**2.86**)
RF	2.16 (**2.19**)	2.62 (**2.64**)

From Table 7.4, we observe that Random Forest outperforms SVM and MLP in performance.

5.3 INFERENCE

The number of features in the dataset is a relatively small (only 17 features). And with feature selection, the number reduces even further. As mentioned earlier, various decision tree together comprise a random forest regression model. Since decision trees output a score based on feature value conditions(for instance, when functions = 3 for selection sort task, the decision tree will output a better score) and the outputs of the decision trees are ensembled, the random forest regression algorithm fits well with this data and outperforms the other two models in this case.

Since it is not possible to visualise the 17 features in 2 dimensions, the sum of all the features was plotted against the program score. Figures 7.1 to 7.3 show the difference between the actual score (points plotted in orange) and the predicted score by the three regressors. The difference is represented as a line between the points. The three plots are for the selection sort task.

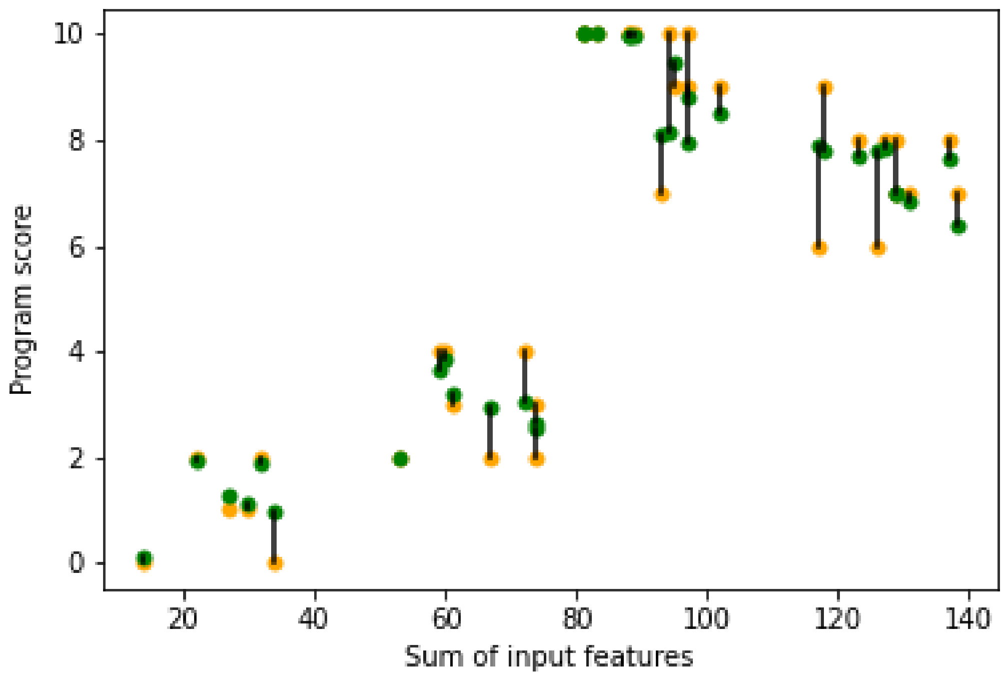


FIGURE 5.1: Actual score vs score predicted by Random Forest Regressor

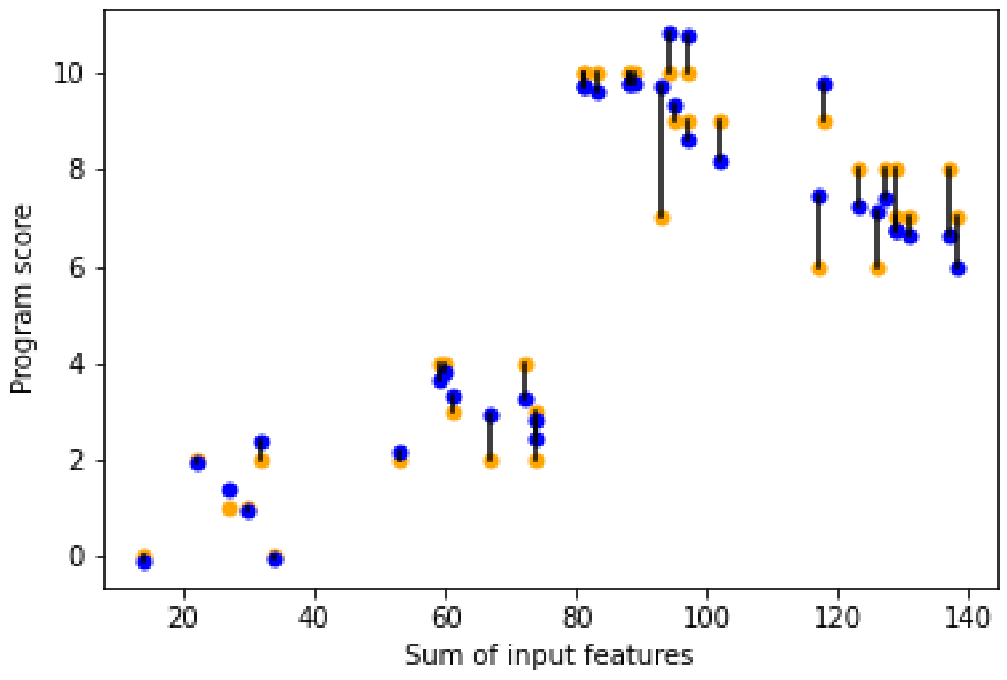


FIGURE 5.2: Actual score vs score predicted by Support Vector Regressor

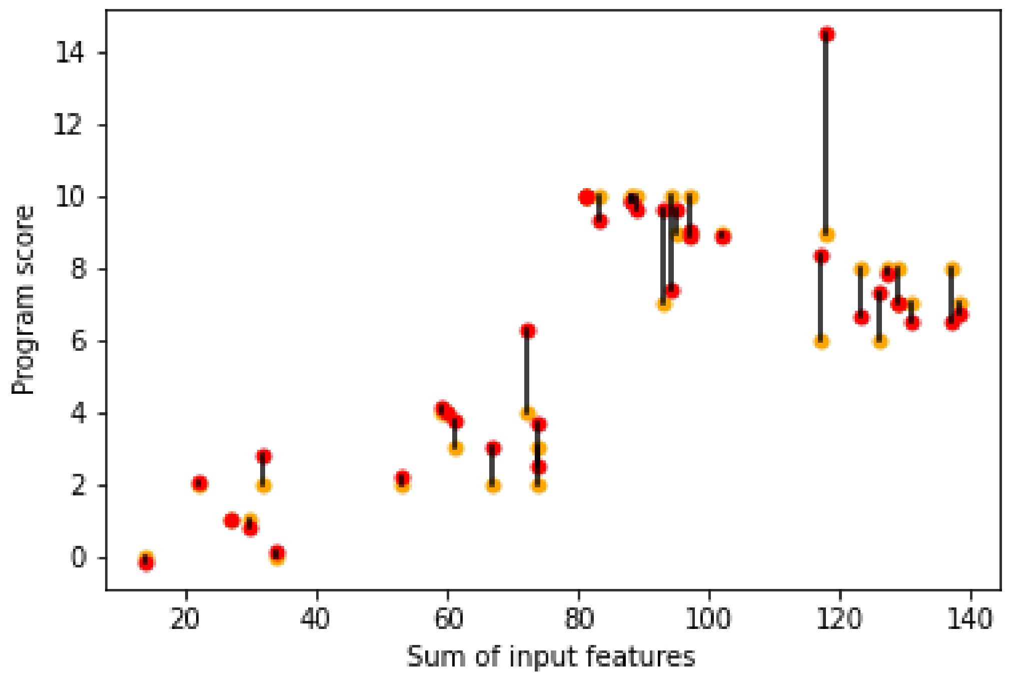


FIGURE 5.3: Actual score vs score predicted by MLP Regressor

While MLP regressor and linear support vector regressor can output a score greater than 10 or lesser than 0, random forest regressor always outputs a score in the range of 0 to 10. This characteristic of random forest regressor is accounted by the decision trees which form the regressor. Since MLP regressor trains through backpropagation and support vector regressor tries to find the best hyperplane which contains maximum number of points, these models may output a score greater than 10. But in the case of random forest regressor, the model outputs a score based on the ensembling of various decision trees. The maximum score that can be awarded will be set as 10 and when a particular code submission passes all conditions (conditions are based on different feature values), that program will be graded with a score of 10.

This can be observed in Figures 7.1 to 7.3. In case of random forest regressor, as shown in Figure 7.1, the scores always lie between 0 and 10. Also, the difference between actual and predicted values is lesser in general. In case of Support Vector Regressor, as shown in Figure 7.2, a couple of samples were given a score of over 10. In case of multi layer perceptron, one code sample was given a score over 14. Thus,it can be inferred that random forest regressor is the best suited choice of regression algorithm to grade student programs automatically.

CHAPTER 6

FEEDBACK GENERATION

After prioritizing the best regression models for each problem, we move on to using the selected model for generating feedback. The feedback generation process follows a cyclic information flow where the model is verified each time for a change depending on the input change and the corresponding change is reflected as feedback. The following subsections explain the process of how constructive feedback is generated.

6.1 GOLDEN FEATURE VECTOR

The golden feature vector F is the best feature vector that is calculated for each problem. In retrospect, the golden feature vector is the average of feature vectors of all code solutions submitted for the program that have score greater than or equal to an empirically decided threshold.

At any instance, the golden feature vector represents the correct and accurate code solutions for a particular problem. The goal of this vector is to act as a comparison tool for feature vectors that require feedback/improvement.

6.2 ALGORITHM

Now, to generate feedback for a particular target program x with x_i features where 'i' ranges between 0 to 17, we perform an iterative process of replacing one of the features x_i with the corresponding feature value X_i from the program's golden feature vector F. Now, the new modified feature vector is passed as input into the regression model.

- If the output score improves after replacement, now we compare x_i and X_i.
  - If $x_i > X_i$, it is advised to decrease that feature in the program.
  - If $x_i < X_i$, it is advised to decrease that feature in the program.
- If the output score decreases after replacement, we consider the next feature in x_i
- The above process is iterated until all the n features have been considered.

In each iteration, the goal of the algorithm is to make the target feature vector closer to the golden feature vector. Thus, the changes that are suggested as feedback are inferred to be improvements for the target vector and constructive feedback is provided.

CHAPTER 7

DEPLOYMENT

After the regressor model for automatic grading and feedback generation was finalised as Random Forest Regressor (which in general outperformed the other two models), the next step was to deploy the model with a user interface where students can login and attempt the tasks. Once the students submit their solution, the score for their submission and feedbacks, if any, had to be displayed.

Streamlit framework was used to deploy the random forest regressor model which would grade students' code solution submissions out of 10 and to generate appropriate feedbacks. Streamlit framework was chosen since it provides a variety of advantages. Firstly, it is simple to create web applications with short Python programs using Streamlit. It supports a lot of Python frameworks and libraries like Pandas, Numpy, Scikit-learn, Pytorch, Tensorflow etc.

Another advantage of Streamlit is that it provides a mechanism for caching so that even when a expensive computation is done or a large dataset is manipulated, the app's performance will not be compromised. This is achieved with the use of the st.cache decorator.

Sqlite3 database was integrated with the web application so that new students can register and existing students can login to the student code submission portal.

Instructor hints/suggestions are also made visible to students after login. With the use of both the instructor hints and the personalised feedback, a student can improve his code solution.

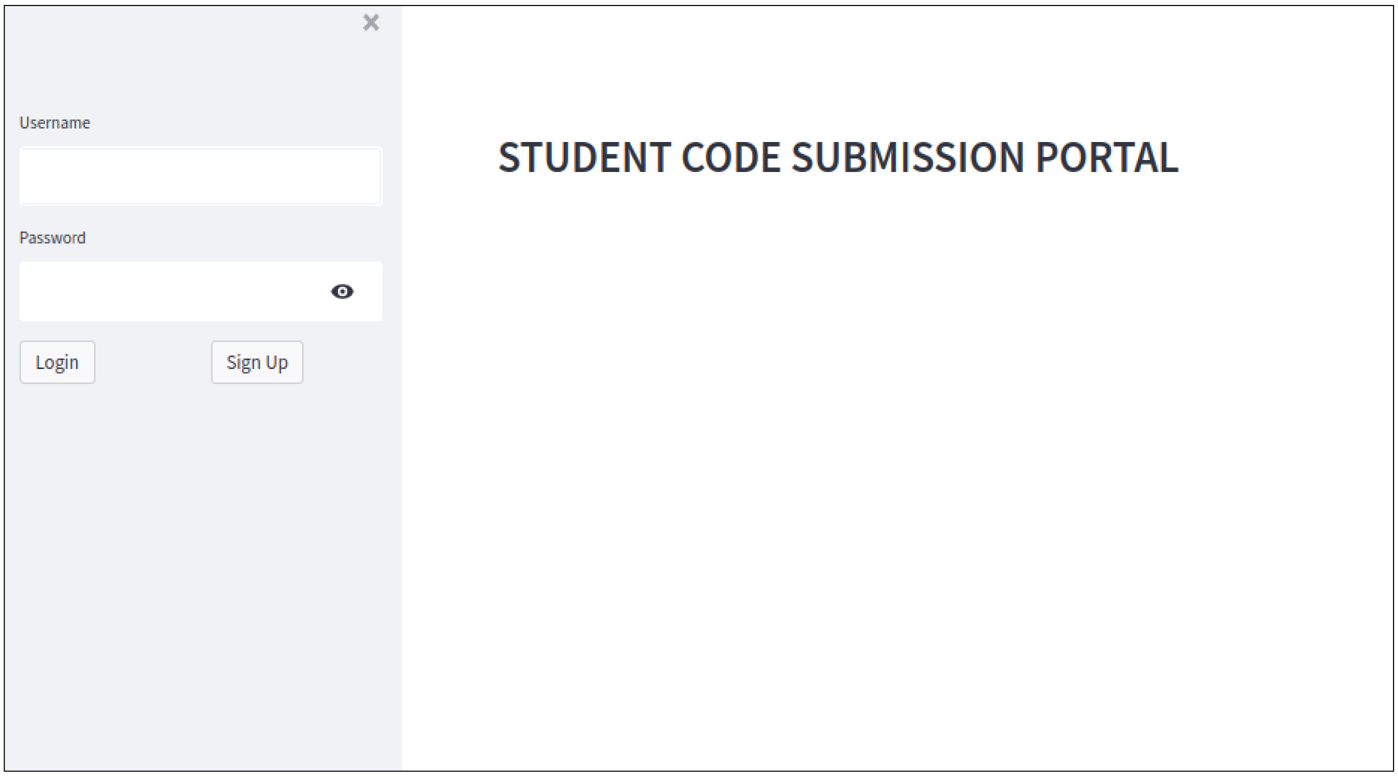
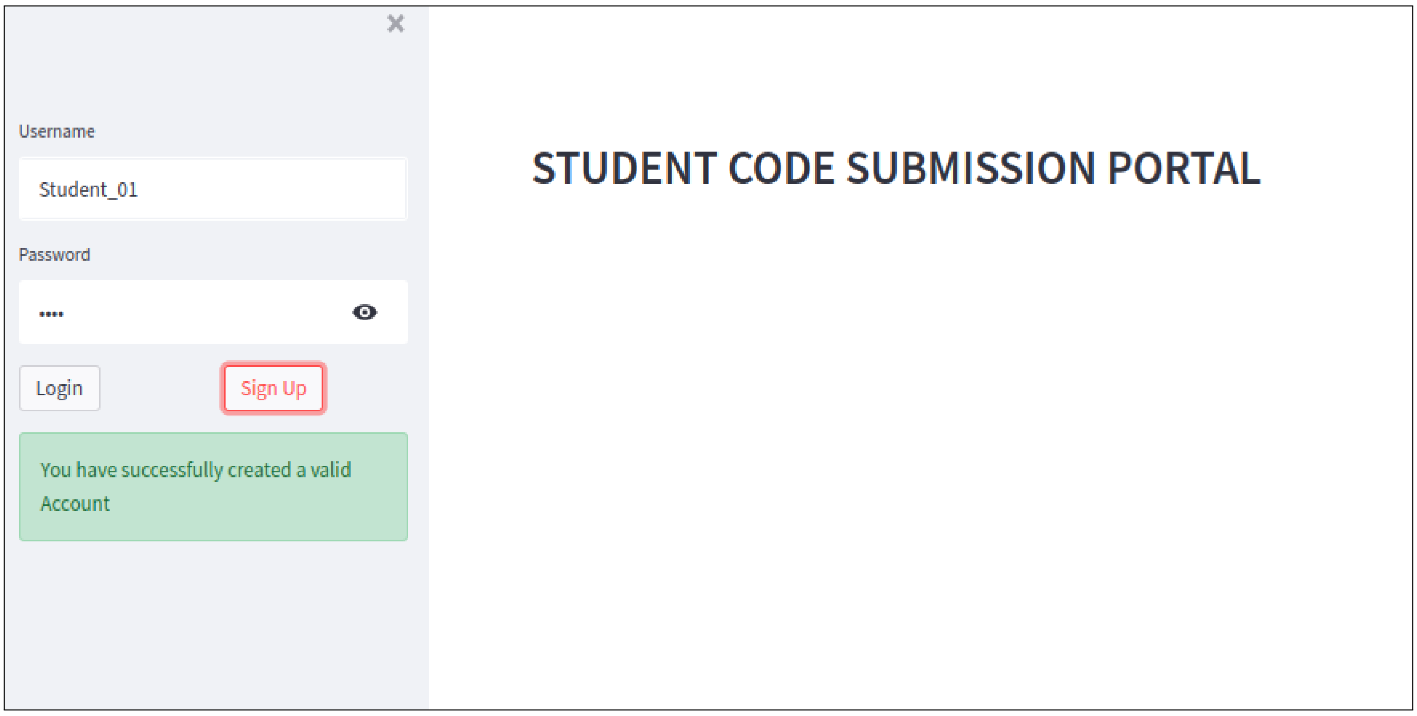


FIGURE 7.1: Snippet of Student Code Submission portal before login/sign-up



You have successfully created a valid Account

FIGURE 7.2: Snippet of Student Code Submission portal after sign-up

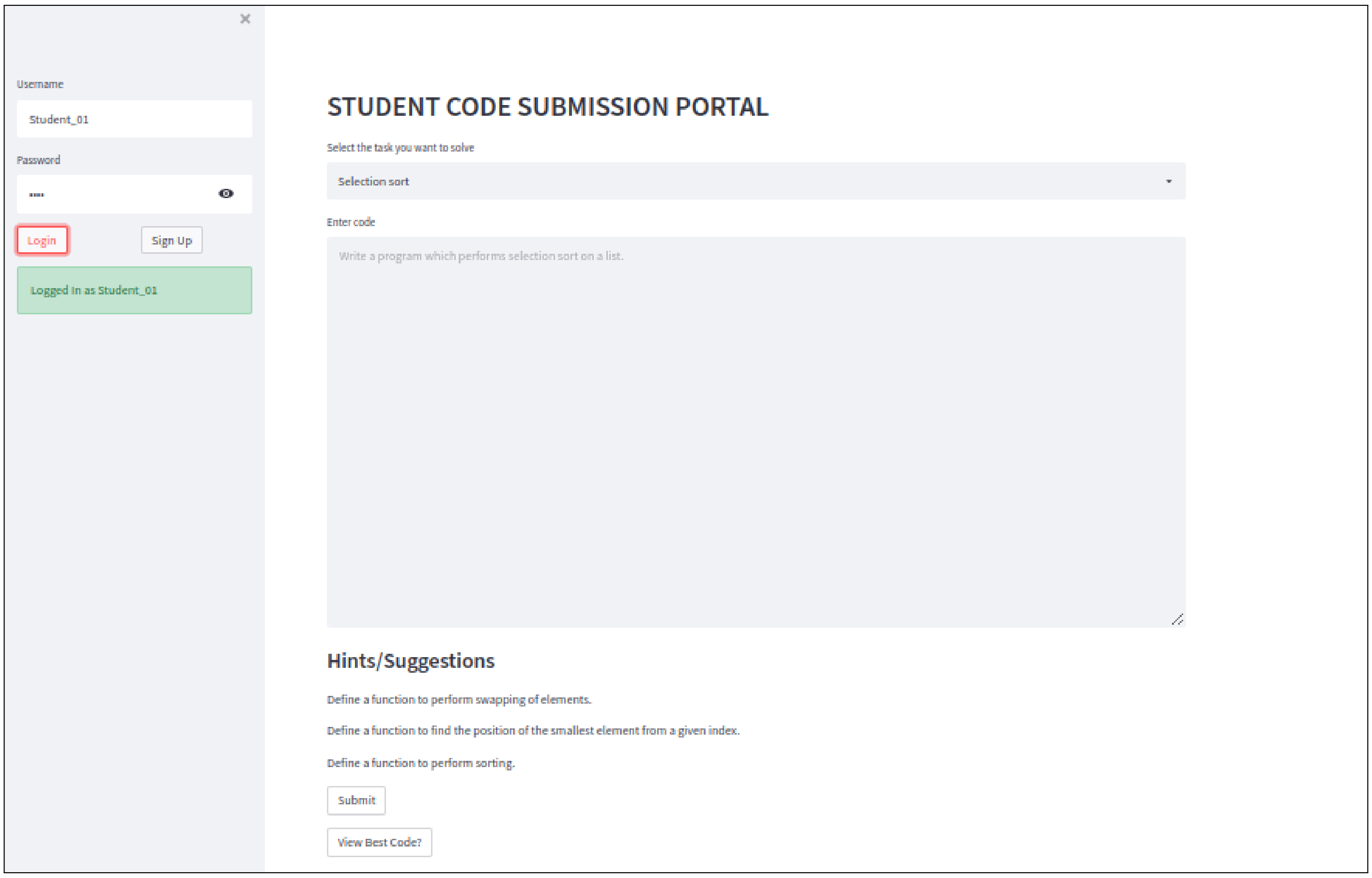
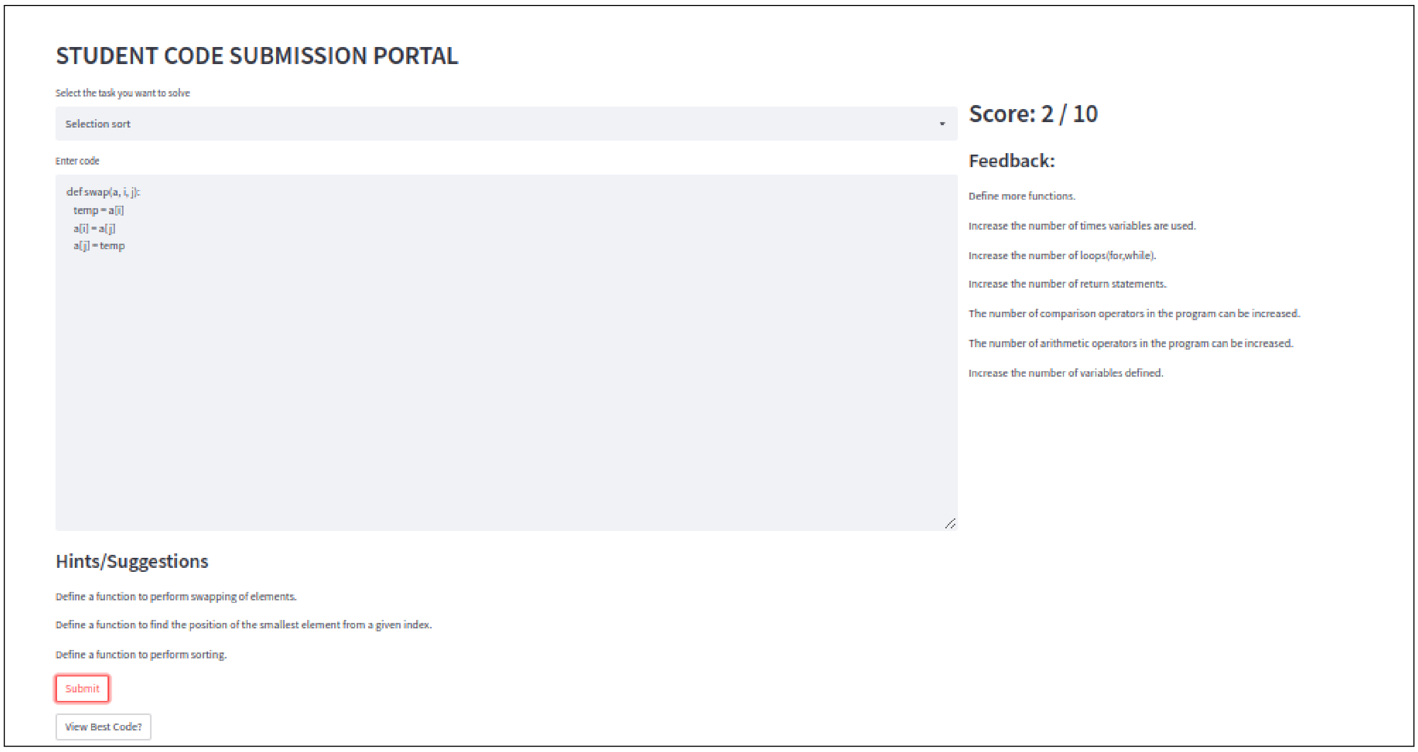

×
Username
Student_01
Password
••••
Login
Sign Up
Logged In as Student_01
STUDENT CODE SUBMISSION PORTAL
Select the task you want to solve
Selection sort
Enter code
Write a program which performs selection sort on a list.
Hints/Suggestions
Define a function to perform swapping of elements.
Define a function to find the position of the smallest element from a given index.
Define a function to perform sorting.
Submit
View Best Code?

FIGURE 7.3: Snippet of Student Code Submission portal after login

Figure 9.1 is the snippet of the student code submission portal before login. Since the student has not logged in yet, the tasks to be attempted are not visible.

Figures 9.2 and 9.3 are the snippets of the student code submission portal after sign-up and login. Once the student enters his username and password, the password is hashed using Secure Hash Algorithm (SHA 256) and the system checks if the user has registered already. Post successful login, the select-box and code area become visible to the student.



STUDENT CODE SUBMISSION PORTAL

Select the task you want to solve

Selection sort

Enter code

```
def swap(a, i, j):
  temp = a[i]
  a[i] = a[j]
  a[j] = temp
```

Score: 2 / 10

Feedback:

Define more functions.

Increase the number of times variables are used.

Increase the number of loops(for,while).

Increase the number of return statements.

The number of comparison operators in the program can be increased.

The number of arithmetic operators in the program can be increased.

Increase the number of variables defined.

Hints/Suggestions

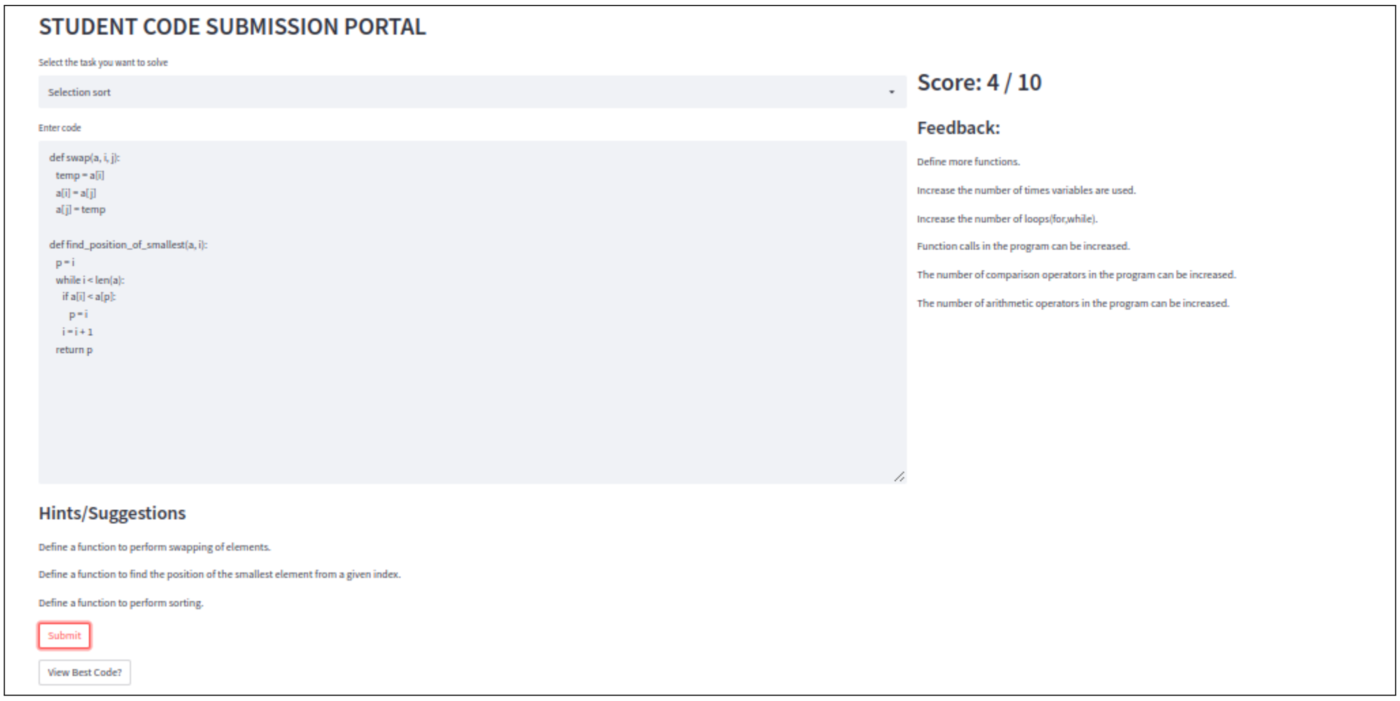
Define a function to perform swapping of elements.

Define a function to find the position of the smallest element from a given index.

Define a function to perform sorting.

FIGURE 7.4: Automatic Grading and Feedback Generation - Example 1

Figure 9.4 shows an example student submission for the selection sort task. Since the code is incomplete and only swap function is defined, the code submission is given a score of 2 out of 10. The code review feedbacks which are generated are appropriate which include suggestions such as “define more functions”, “add more loop statements” and so on.



STUDENT CODE SUBMISSION PORTAL

Select the task you want to solve

Selection sort

Enter code

```
def swap(a, i, j):
  temp = a[i]
  a[i] = a[j]
  a[j] = temp

def find_position_of_smallest(a, i):
  p = i
  while i < len(a):
    if a[i] < a[p]:
      p = i
    i = i + 1
  return p
```

Score: 4 / 10

Feedback:

Define more functions.

Increase the number of times variables are used.

Increase the number of loops(for,while).

Function calls in the program can be increased.

The number of comparison operators in the program can be increased.

The number of arithmetic operators in the program can be increased.

Hints/Suggestions

Define a function to perform swapping of elements.

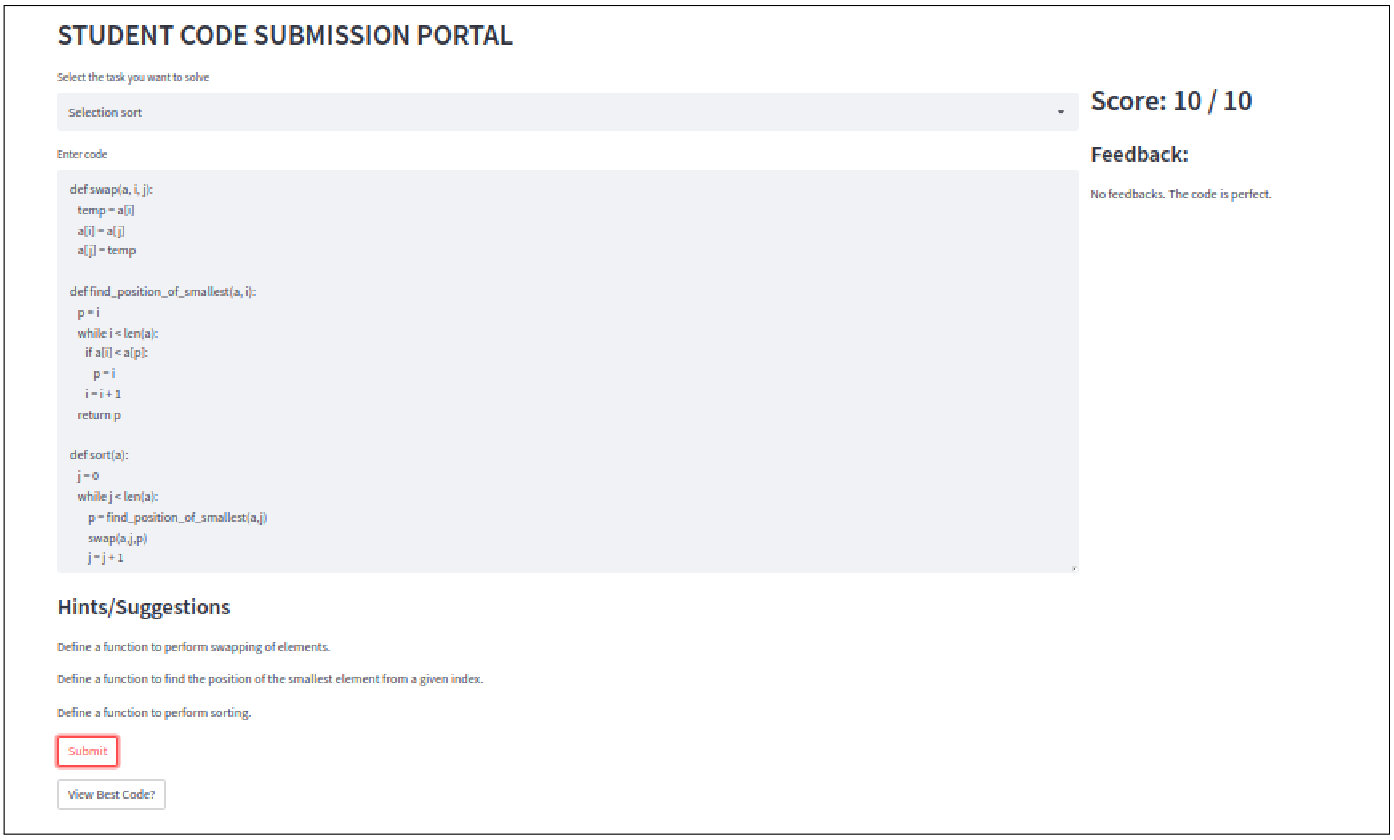
Define a function to find the position of the smallest element from a given index.

Define a function to perform sorting.

FIGURE 7.5: Automatic Grading and Feedback Generation - Example 2

Figure 9.5 shows another example student submission for the selection sort task. In this case, the program is incomplete again, with only two functions defined for swapping and finding the position of the smallest element in the list from a given index. Since the main sort function is missing which would have had function calls to the two defined functions and a loop statement, the code submission gets a grade of 4 out of 10 and the necessary code review feedbacks.

Figure 9.6 is another example student submission for selection sort task. In this case, the model grades the program a perfect 10/10 with no code review feedbacks. The code submission is perfect with three functions defined to perform swapping, finding the position of the smallest element from a given index and finally the sort function. The sort function includes two function calls to the previously defined functions. Thus, the model considers the particular code submission as a perfect code sample.


STUDENT CODE SUBMISSION PORTAL

Select the task you want to solve

Selection sort

Enter code

```
def swap(a, i, j):
  temp = a[i]
  a[i] = a[j]
  a[j] = temp

def find_position_of_smallest(a, i):
  p = i
  while i < len(a):
    if a[i] < a[p]:
      p = i
    i = i + 1
  return p

def sort(a):
  j = 0
  while j < len(a):
    p = find_position_of_smallest(a,j)
    swap(a,j,p)
    j = j + 1
```

Hints/Suggestions

Define a function to perform swapping of elements.

Define a function to find the position of the smallest element from a given index.

Define a function to perform sorting.

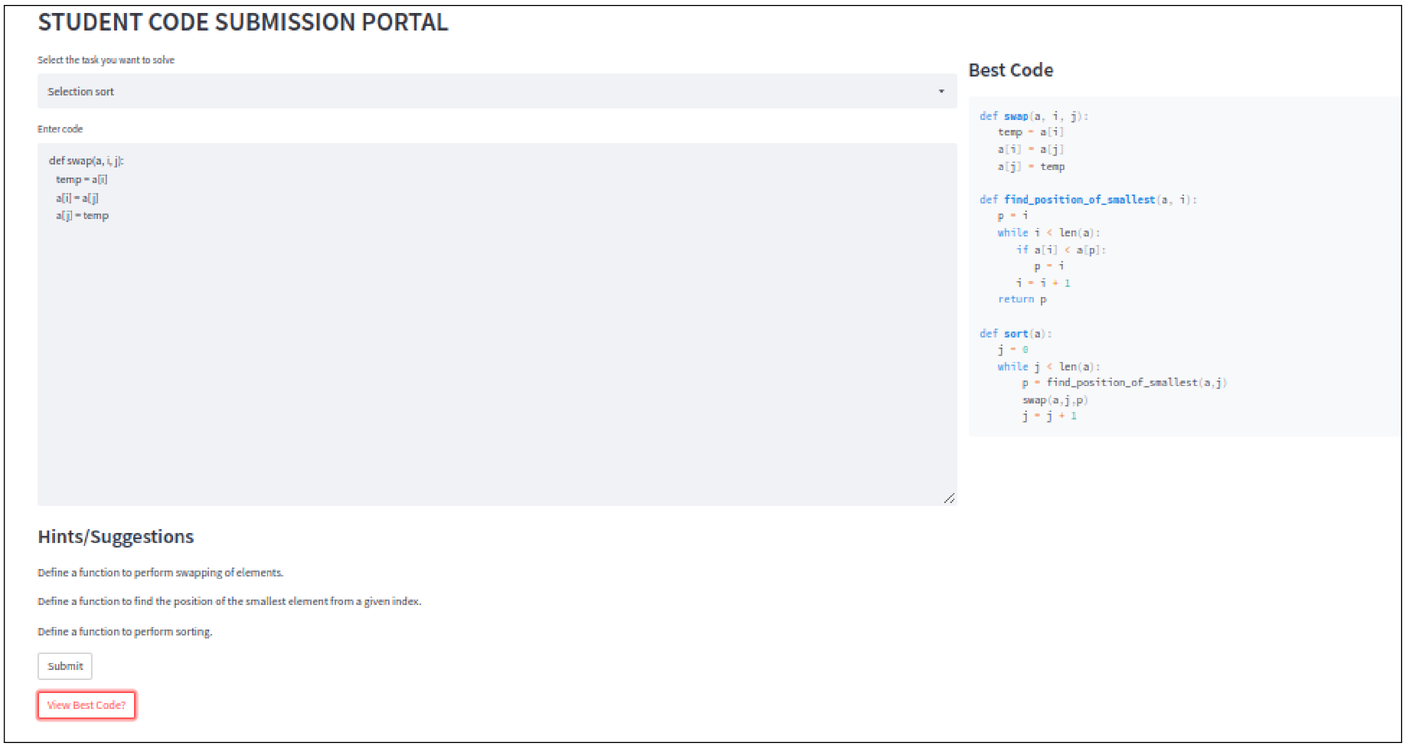
Score: 10 / 10

Feedback:

No feedbacks. The code is perfect.

FIGURE 7.6: Automatic Grading and Feedback Generation - Example 3

There is also a another feature which could prove helpful to students. As every assignments have a deadline, students can be given access to view the best code solution after the deadline. 'View Best Code?' option gives a chance for the students to have a look into the best submission. This is shown in Figure 9.7.



STUDENT CODE SUBMISSION PORTAL

Select the task you want to solve

Selection sort

Enter code

```
def swap(a, i, j):
  temp = a[i]
  a[i] = a[j]
  a[j] = temp
```

Best Code

```
def swap(a, i, j):
  temp = a[i]
  a[i] = a[j]
  a[j] = temp

def find_position_of_smallest(a, i):
  p = i
  while i < len(a):
    if a[i] < a[p]:
      p = i
    i = i + 1
  return p

def sort(a):
  j = 0
  while j < len(a):
     p = find_position_of_smallest(a,j)
     swap(a,j,p)
     j = j + 1
```

Hints/Suggestions

Define a function to perform swapping of elements.

Define a function to find the position of the smallest element from a given index.

Define a function to perform sorting.

FIGURE 7.7: Viewing best code submission

CHAPTER 8

CONCLUSION

We proposed a pipeline for grading a student code submission out of 10 which will also provide appropriate feedback to improve the code. The student programs were represented using 17 features which were retrieved from the submitted codes and from the AST representation of the codes. Feature set for each coding assignment was optimised via feature selection. The initial dataset created by us was used to train three regressor models such as Linear SVR, MLP regressor and Random Forest Regressor. The Random Forest regressor was shown to be the most effective for this problem statement. The Random Forest Regressor model has a MAE of 1.7 and RMSE of 2.3.

Furthermore, our technique delivers personalised feedback for a program by comparing its feature vector with the average feature vector of the ”excellent” programs submitted by students.Whenever a student submits a code, he will be given feedback instantly to help him improve his code. Students can also view the best code submission after the assignment deadline. The Streamlit framework was used to deploy this pipeline as a web app.

REFERENCES

1. “2to3 – Automated Python 2to3 code translation”, https://docs.python.org/3/library/2to3.html

2. “AST module Python 3.10.4”, https://docs.python.org/3/library/ast.html

3. Azcona D, Arora P, Hsiao I H, Smeaton A. (2019) “user2code2vec: Embeddings for profiling students based on distributional representations of source code” , In Proceedings of the 9th International Conference on Learning Analytics & Knowledge, pp. 86-95.

4. Benjamin Paaben, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. (2017) “The Continuous Hint Factory-Providing Hints in Vast and Sparsely Populated Edit Distance Spaces”, arXiv preprint arXiv:1708.06564

5. Breiman L. (2001). “Random forests — Machine Learning” 45, 5-32.

6. Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, Leonidas Guibas. (2015) “Learning program embeddings to propagate feedback on student code” , In Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37. JMLR. org, 1093–1102.

7. Drucker, Harris, et al. (1996) “Support vector regression machines”, Advances in neural information processing systems, 9.

8. Jović, Alan, Karla Brkic, Nikola Bogunovic. (2015) “A review of feature selection methods with applications” , 38th international convention on information and communication technology, electronics and microelectronics (MIPRO). Ieee, 2015.

9. Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin. (2016) "Convolutional Neural Networks over Tree Structures for Programming Language Processing", In AAAI, Vol. 2. 4

10. Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, Lu Zhang. (2014) "Building program vector representations for deep learning", arXiv preprint arXiv:1409.3358.

11. Maxim Rabinovich, Mitchell Stern, Dan Klein. (2017) "Abstract syntax networks for code generation and semantic parsing", arXiv preprint arXiv:1704.07535.

12. Miltiadis Allamanis, Hao Peng, Charles Sutton. (2016) "A convolutional attention network for extreme summarization of source code", In International Conference on Machine Learning. 2091–2100.

13. Murtagh, Fionn. (1991) "Multilayer perceptrons for classification and regression" ,Neurocomputing 2.5-6 : 183-197.

14. Orr, J Walker, Nathaniel Russell. (2021) "Automatic Assessment of the Design Quality of Python Programs with Personalized Feedback", arXiv preprint arXiv:2106.01399 .

15. Pedregosa et al., (2011) "Scikit-learn: Machine Learning in Python", JMLR 12, pp. 2825-2830.

16. Pylint.org. "Pylint - code analysis for python". https://pylint.org.

17. Sebastian Gross, Bassam Mokbel, Benjamin Paaben, Barbara Hammer, Niels Pinkwart. (2014) "Example-based feedback provision using structured solution spaces", International Journal of Learning Technology 10 9, 3 , 248–280.

18. Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini. (2016) “A dataset of simplified syntax trees for C”, In Proceedings of the 13th International Conference on Mining Software Repositories. ACM, 476–479.

19. Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. (2018) “A general path-based representation for predicting program properties”, arXiv preprint arXiv:1803.09544.

20. Veselin Raychev, Martin Vechev, Eran Yahav. (2014). “Code completion with statistical language models”,In Acm Sigplan Notices, Vol. 49. ACM, 419–428.