

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 *bonafide* 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

I thank GOD, the almighty for giving me strength and knowledge to do this project.

I would like to thank and deep sense of gratitude to my guide **Dr. R.S. MILTON**, Associate Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped me to shape and refine my work.

My sincere thanks to **Dr. T.T. MORNALINEE**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and I would like to thank our project Coordinator **Dr. R.S. MILTON**, Associate Professor, Department of Computer Science and Engineering for her valuable suggestions throughout this project.

I express my deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. I also express my appreciation to our **Dr. V. E. ANNAMALAI**, Principal, for all the help he has rendered during this course of study.

I would like to extend my sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of my project work. Finally, I would like to thank my parents and friends for their patience, cooperation and moral support throughout my life.

RAMAKRISHNAN A M RISHI VARDHAN K SACHIN KRISHAN T

ABSTRACT

Students learning to code can immensely benefit from personalized feedback for their specific codes to improve coding standards. However, evaluating the functionality and design quality to give personalized feedback is a very time consuming task for instructors and teaching assistants. Further, design quality is nuanced and is difficult to concisely express as a set of rules. For these reasons, we propose an end-to-end pipeline to both automatically assess the design of a program and provide personalized feedback to guide students on how to make corrections. The model's effectiveness is evaluated on a corpus of student programs written in Python. The model has an average Mean Absolute Error(MAE) of 1.7 and average Root Mean Square Error(RMSE) of 2.3 when the programs are evaluated out of 10.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
1 INTRODUCTION	1
2 BACKGROUND AND MOTIVATION	3
3 LITERATURE SURVEY	5
4 DESIGN & ARCHITECTURE	6
4.1 TRAINING	6
4.2 DEPLOYMENT	7
5 DATA PREPROCESSING	10
5.1 DATASET ANNOTATION	11
5.1.1 Example Code Submissions	16
5.2 FEATURE EXTRACTION	20
5.3 FEATURE SELECTION	22
5.3.1 Implementation	23
6 ALGORITHMS	27
6.1 SUPPORT VECTOR MACHINE	27
6.1.1 Hyperparameters in SVR	28

6.1.2	Support Vector Regression	29
6.2	RANDOM FOREST	30
6.2.1	Ensemble Learning	30
6.2.2	Decision Tree	30
6.2.3	Bootstrapping	31
6.2.4	Random Forest Regression	32
6.3	MULTI LAYER PERCEPTRON	33
7	IMPLEMENTATION	36
7.1	METRICS	36
7.1.1	Mean Absolute Error (MAE)	36
7.1.2	Root Mean Absolute Error (RMSE)	37
7.2	RESULTS	37
7.2.1	Selection Sort	37
7.2.2	First Negative Item in List	38
7.2.3	Largest Item in List	38
7.2.4	Unique Character count in a string	38
7.3	INFERENCE	39
8	FEEDBACK GENERATION	40
8.1	GOLDEN FEATURE VECTOR	40
8.2	ALGORITHM	41
9	DEPLOYMENT	42
A	CONCLUSION AND FUTURE WORK	47

LIST OF TABLES

5.1	Questions and Description	10
5.2	Grading Parameters	12
5.3	Grading Rubric (Design) for Selection Sort	12
5.4	Grading Rubric (Design) for First Negative Element in List	13
5.5	Grading Rubric (Design) for Largest Element in List task	14
5.6	Grading Rubric (Design) for Unique Character Count in string . .	14
5.7	Grading Rubric (Functionality)	15
5.8	Features extracted from code	21
5.9	Features extracted from AST	21
5.10	F-statistic table : Selection Sort	23
5.11	F-statistic table : First Negative Element in List	24
5.12	F-statistic table : Largest Element in List	25
5.13	F-statistic table : Unique Character Count in List	26
7.1	Results - Selection Sort	37
7.2	Results - First Negative Item in List	38
7.3	Results - Largest Item in List	38
7.4	Results - Unique Characters count in a string	38

LIST OF FIGURES

4.1	Training	7
4.2	Deployment	9
5.1	Data Annotation Flow	11
5.2	Best Program - Selection Sort	16
5.3	Worst Program - Selection Sort	17
5.4	Best Program - First negative element in a list	17
5.5	Worst Program - First negative element in a list	17
5.6	Best Program - Unique characters count in a string	18
5.7	Worst Program - Unique characters count in a string	19
5.8	Best Program - Largest element of a list	19
5.9	Worst Program - Largest element of a list	20
6.1	Simple SVR	28
6.2	Decision tree layout	31
6.3	Simple RF	32
6.4	Perceptron model	33
6.5	MLP model	34
9.1	Snippet of Student Code Submission portal before login	43
9.2	Snippet of Student Code Submission portal after login	43
9.3	Automatic Grading and Feedback Generation - Example 1	44
9.4	Automatic Grading and Feedback Generation - Example 2	45
9.5	Automatic Grading and Feedback Generation - Example 3	46

CHAPTER 1

INTRODUCTION

Static Code Analysis (also known as Source Code Analysis) is usually performed as part of a Code Review (also known as white-box testing) Static Code Analysis commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within ‘static’ (non-running) source code. The process provides an understanding of the code structure and can help ensure that the code adheres to industry standards.

Static analysis is used in software engineering by software development and quality assurance teams. Automated tools can assist programmers and developers in carrying out static analysis. The software will scan all code in a project to check for vulnerabilities while validating the code. Static analysis is generally good at finding coding issues such as: Programming errors, Coding standard violations, security vulnerabilities.

Some benefits of using static code analysis is increasing code quality as it can evaluate all the code in an application. It provides speed in using automated tools compared to manual code review. Paired with normal testing methods, static testing allows for more depth into debugging code. Automated tools are less prone to human error. It will increase the likelihood of finding vulnerabilities in the code, increasing web or application security.

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 do this using a novel pipeline that is

trained on our manually annotated student assignment data-set. The pipeline will provide a design value score between 1 and 10 in our research which will be used to generate individualized feedback explaining the reasoning behind the estimated design value score.

As a result, the review or feedback provided helps students improve the quality of their code as feedback is an important aspect of effective learning. Feedback is more strongly and consistently associated with accomplishment than any other teaching behavior in the context of student assignments, according to research. Regardless of grade, socioeconomic background, color, or school setting, this association exists. Feedback can boost a student's self-esteem, self-awareness, and desire to study. Giving students feedback in this way has been suggested as a way to improve learning and evaluation performance.

CHAPTER 2

BACKGROUND AND MOTIVATION

Feedback is an essential component of scaffolding for learning. Feedback provides insights into the assistance of learners in terms of achieving learning goals and improving self-regulated skills. The results highlight an increased engagement while performing the assessment, the usefulness of the feedback, as well as where the explanation was clear and where improvements are needed.

Especially with the recent compulsion for online learning due the pandemic, feedback becomes even more critical since instructors and students are separated geographically and physically. In this context, feedback allows the instructor to customize learning content according to the students' needs. However, giving feedback is a challenging task for instructors, especially in contexts of large cohorts. As a result, several automatic feedback systems have been proposed to reduce the workload on the part of the instructor. Although these systems have started gaining research attention.

The manual grading of assignments is a tedious and error-prone task, and the problem particularly aggravates when such an assessment involves a large number of students. The use of artificial intelligence can be useful to address these issues by automating the grading process, we can assist teachers in the correction and enable students to receive immediate feedback, thus improving their solutions before the final submission.

Unlike in-person education methods, online education tends to be more affordable. There's also often a wide range of payment options that let you pay in

installments or per class. This allows for financially disadvantaged students to upskill themselves to open avenues for better career opportunities. Money can also be saved from the commute and class materials, which are often available for free. In other words, the monetary investment is less, but the results can be comparable to other options.

Online education enables the student to set their own learning pace, and there's the added flexibility of setting a schedule that fits everyone's agenda. As a result, using an online educational platform allows for a better balance of work and studies, so there's no need to give anything up. Studying online makes finding a good work-study balance easier.

Students of online courses that lack a real-time tutor available can also reap the benefits of feedback with an assisting machine learning based Static Code Analyser that gives personalized feedback to students

CHAPTER 3

LITERATURE SURVEY

David Azcona et al. (2019) [1] used program embeddings to profile students based on their code submissions. Around five hundred thousand code submissions of python2 and bash which were collected over three years were used to extract embeddings. They proved that it is hard to cluster students due to the curse of dimensionality.

J. Walker Orr ,Nathaniel Russell(2021)[2] generated personalised feedback to students to improve the design of their program. The focus was on design quality of programs and not the code's expected functionality. 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.

CHAPTER 4

DESIGN & ARCHITECTURE

We propose a design quality assessment system based on Machine learning models that utilizes an abstract syntax tree (AST) to represent programs. The ML models are regression model that are trained on assessed student programs to predict a score between zero and one. Each feature a model uses is designed to be meaningful to human interpretation and is based on statistics collected from the program's AST. We intentionally do not use deep learning as it would make the representation of the program difficult to understand. Personalized feedback is generated based on each feature of an individual program. By swapping a feature's value for an individual program with the average feature value of good programs, it is possible to determine which changes need to be made to the program to improve design

4.1 TRAINING

Dataset contains the code and corresponding scores for each of the student codes. 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: Support vector regressor, Random forest and Multi layer perceptron. The models trained were all regression models which were trained to predict the scores that should be given to the evaluated codes.

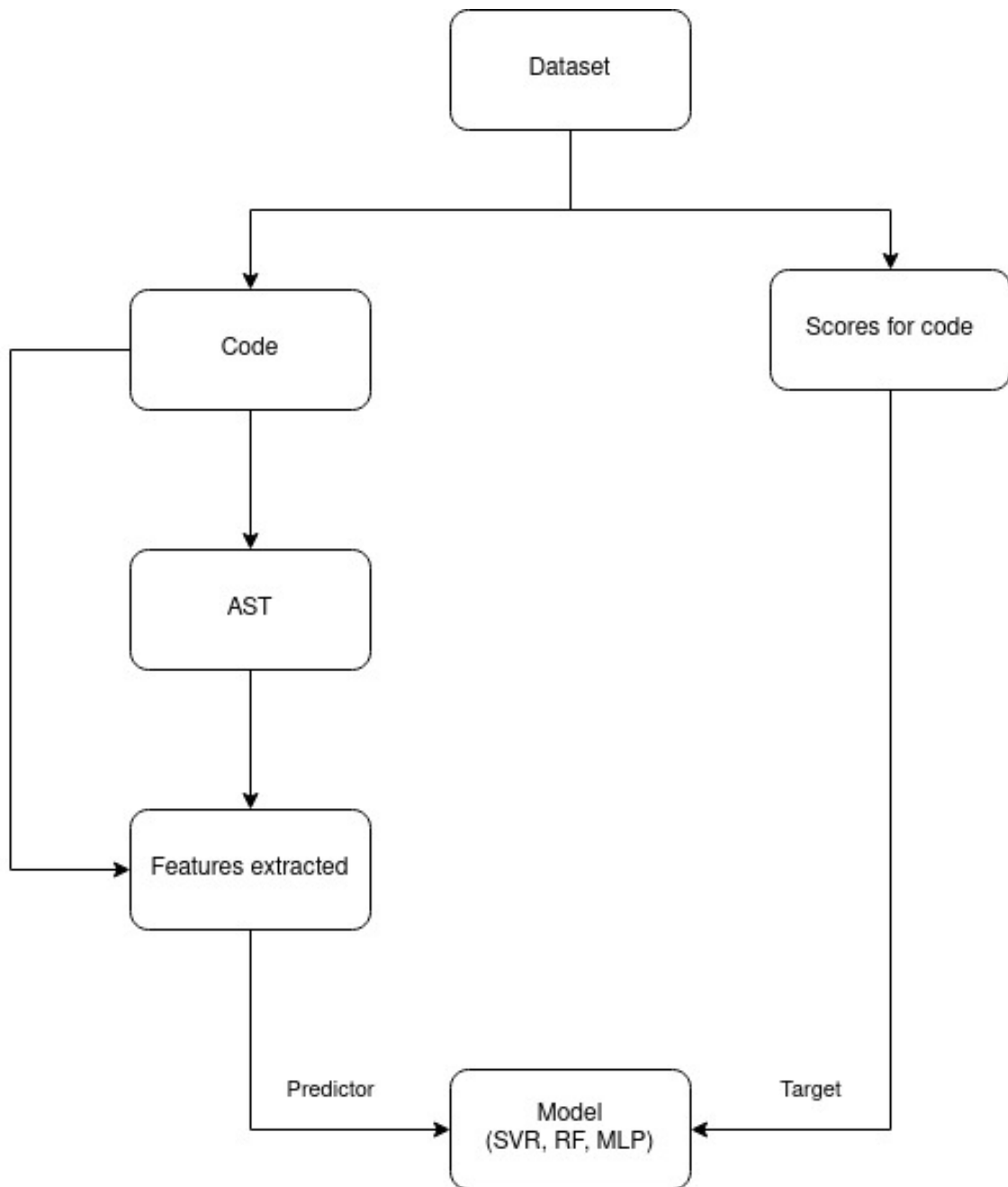


FIGURE 4.1: Training

4.2 DEPLOYMENT

The student code is 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.

In order to avoid the need for a dataset with explicit feedback annotation, we use our model, trained on predicting design score, to evaluate how changes in program features would lead to a higher assessed score. Using the training data, we compute an average feature vector \bar{x} of all the “good” programs. To generate feedback for a program, its feature vector is compared to the average.

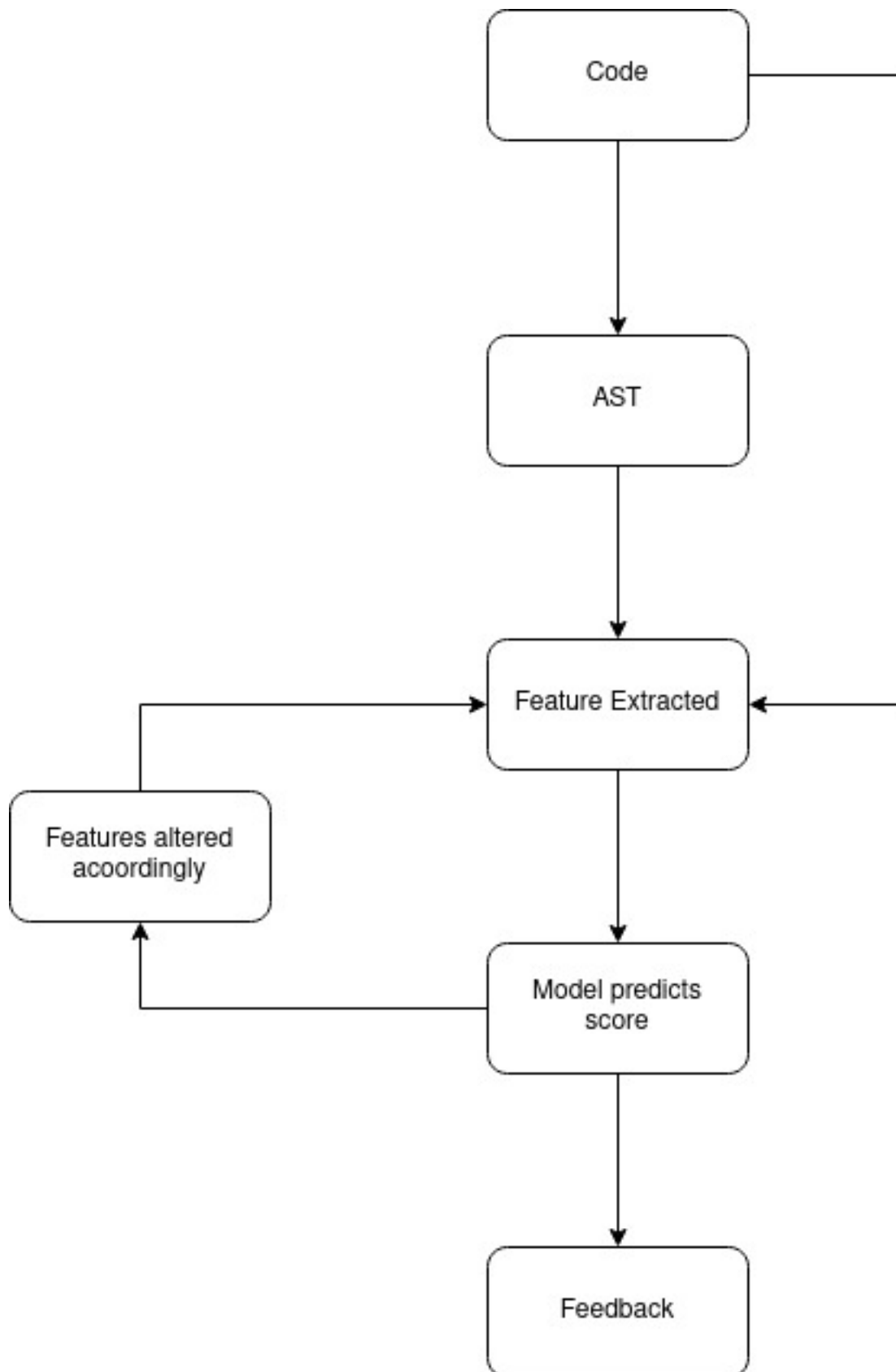


FIGURE 4.2: Deployment

CHAPTER 5

DATA PREPROCESSING

For experimentation, the collection of python codes compiled by the Dublin City University[1] 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 correct (Class 1) or 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 codes 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 5.1. For each task 120 different code solutions were randomly selected. In overall for the 4 different questions, 480 code solutions were compiled for experimentation.

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

TABLE 5.1: Questions and Description

5.1 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 between the range of 0 and 10 and they represent the accuracy of the code solutions to the particular problem. Accordingly higher numeric score represents a correct and optimal code solution to the problem and lower numeric score represents an incorrect or non-optimal code solution. The following figure 5.1 represents the flowchart architecture for dataset annotation.

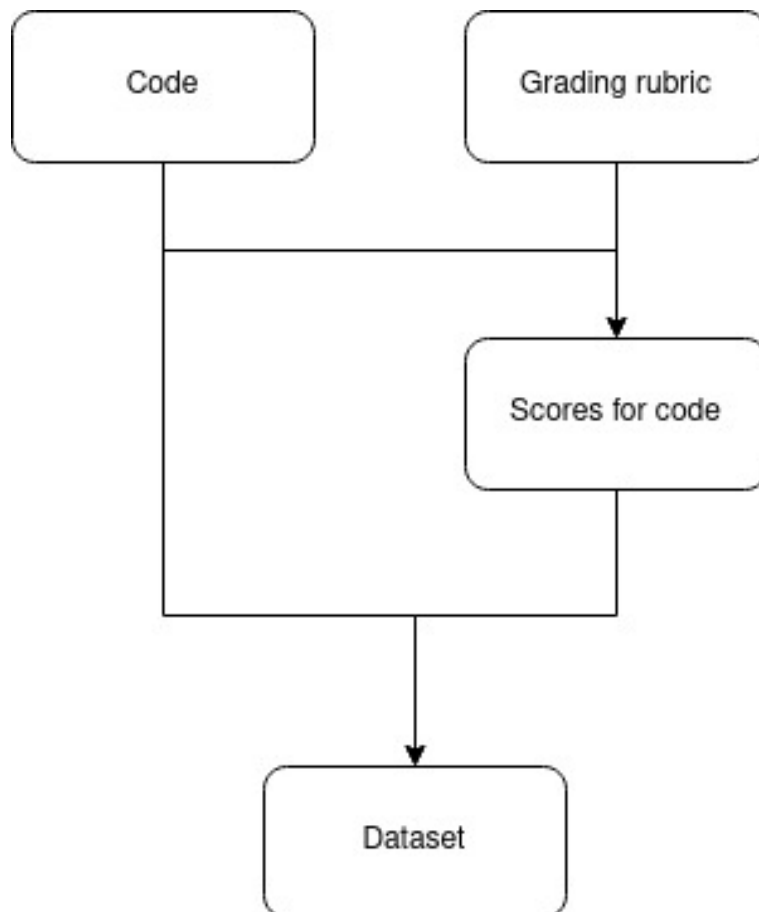


FIGURE 5.1: Data Annotation Flow

The score component for a value of 10 points was divided into two parts, **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 5.2. Tables 5.3 -5.6 details the appropriate score to respective problems based on their necessary individual criteria for each different task.

Parameters
Use of Variables
Modularity
Logic Application
Efficiency

TABLE 5.2: Grading Parameters

- **Selection Sort**

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

TABLE 5.3: Grading Rubric (Design) for Selection Sort

For 'Selection Sort' task, three functions had to be defined - one for swapping, one to find the smallest element of the list from a given index

and the final 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**

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

TABLE 5.4: Grading Rubric (Design) for First Negative Element in List

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 - use of break statement after finding the first negative element).

- **Largest Element in the List**

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.

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

TABLE 5.5: Grading Rubric (Design) for Largest Element in List task

- **Unique Character Count in String**

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

TABLE 5.6: Grading Rubric (Design) for Unique Character Count in string

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. And 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 5.7 was followed to give a score out of 5. The correctness and completeness of the code was considered to grade the code submissions.

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

TABLE 5.7: Grading Rubric (Functionality)

Ultimately a dataset compiled of code solutions and appropriate numeric score values (in a range of 0 to 10) was generated to be used for successive experimentation.

5.1.1 Example Code Submissions

The following figures 5.2 - 5.9 document the examples of best and worst programs for each problems as a result of scoring performed using the developed grading rubric.

5.1.1.1 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 5.2: Best Program - Selection Sort

```
a = ["a","i","j"]
swap (a,1,2)
```

FIGURE 5.3: Worst Program - Selection Sort

5.1.1.2 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 5.4: 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 5.5: Worst Program - First negative element in a list

5.1.1.3 Unique characters count in a string

```
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 5.6: 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 5.7: Worst Program - Unique characters count in a string

5.1.1.4 Largest element of a list

```
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 5.8: Best Program - Largest element of a list

```

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 5.9: Worst Program - Largest element of a list

5.2 FEATURE EXTRACTION

Following annotation of the dataset, feature extraction followed by feature selection is 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 [3].
- **Abstract Syntax Tree (AST) representation:** After converting every program to python 3 format, the AST [4] 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 code. 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 5.9. Other features derived from code are represented in Table 5.8.

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 5.8: Features extracted from code

#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)

TABLE 5.9: Features extracted from AST

Ultimately a total of 17 features were extracted from the code and the AST representation of the code.

5.3 FEATURE SELECTION

After extracting meaningful features from both code and AST representation (Table 5.8 & Table 5.9), 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 [5] 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 a appreciable influence on the target variable(score) are selected and the regressor model for each task is trained with the selected features.

5.3.1 Implementation

The following Tables 5.10 - 5.14 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**

Feature	<i>freg</i> 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

TABLE 5.10: F-statistic table : Selection Sort

From Table 5.10, 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**

Feature	<i>freg score</i>
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

TABLE 5.11: F-statistic table : First Negative Element in List

From Table 5.11, 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**

Feature	<i>freg score</i>
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

TABLE 5.12: F-statistic table : Largest Element in List

From table 5.12, 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**

Feature	<i>freg score</i>
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

TABLE 5.13: F-statistic table : Unique Character Count in List

From table 5.13, 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 6

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.

6.1 SUPPORT VECTOR MACHINE

In machine learning, Support Vector Machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Support Vector Regression [7] is a supervised learning algorithm that is used to predict discrete values. Support Vector Regression uses the same principle as the Support Vector Machine (SVM). The objective of a support vector machine algorithm is to find a hyperplane in an n-dimensional space that distinctly classifies the data points. The data points on either side of the hyperplane that are closest to the hyperplane are called Support Vectors. These

influence the position and orientation of the hyperplane and thus help build the SVM.

6.1.1 Hyperparameters in SVR

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

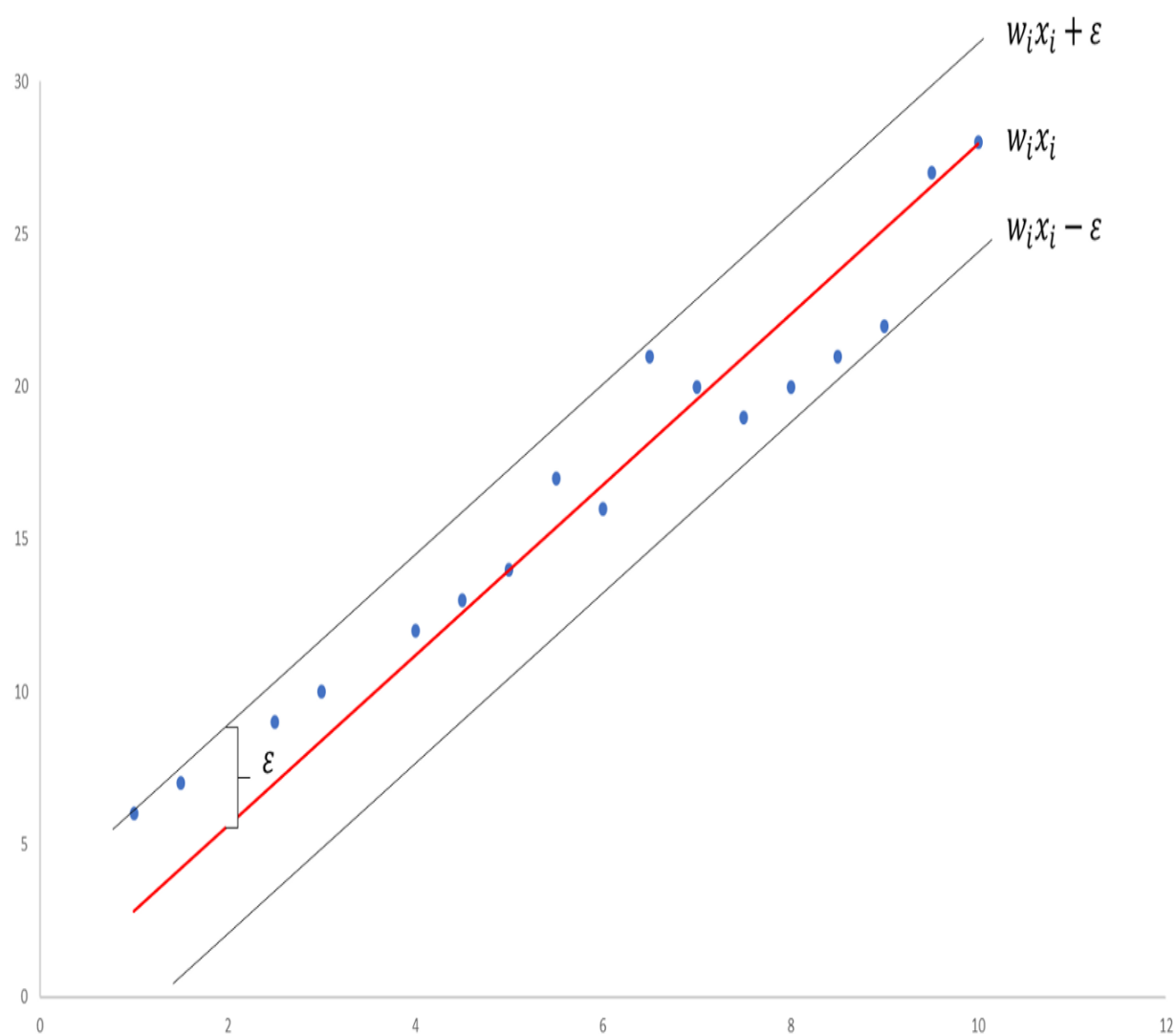


FIGURE 6.1: Simple SVR

- **Hyperplane** (Red line in Figure 6.1)

Hyperplanes are decision boundaries that is used to predict the continuous output. The data points on either side of the hyperplane that are closest to the hyperplane are called Support Vectors. These are used to plot the required line that shows the predicted output of the algorithm.

- **Kernel**

A kernel is a set of mathematical functions that takes data as input and transform it into the required form. These are generally used for finding a hyperplane in the higher dimensional space.

- **Boundary Lines** (Grey lines in Figure 6.1)

These are the two lines that are drawn around the hyperplane at a distance of ϵ (epsilon). It is used to create a margin between the data points.

6.1.2 Support Vector Regression

Support Vector Regression (SVR) finds the best fit line or the hyperplane that has the maximum number of points. Unlike other Regression models that try to minimize the error between the real and predicted value, the SVR tries to fit the best line within a threshold value (the distance between the hyperplane and boundary line).

Hence, we are going to take only those points that are within the decision boundary and have the least error rate, or are within the Margin of Tolerance. This gives us a better fitting model.

6.2 RANDOM FOREST

Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. The functioning of Random Forest is contingent on three main factors,

- Decision Trees
- Ensemble Learning
- Bootstrapping

The following subsections (6.2.1, 6.2.2 and 6.2.3) detail on the mentioned factors description.

6.2.1 Ensemble Learning

Ensemble learning is the process of using multiple models, trained over the same data, averaging the results of each model ultimately finding a more accurate predictive/classification result. The requirement for ensemble learning is that the errors of each model (in this case decision tree) are independent and different from tree to tree.

6.2.2 Decision Tree

Decision Trees are used for both regression and classification problems. The goal is to create a model that predicts the value of a target variable by learning simple

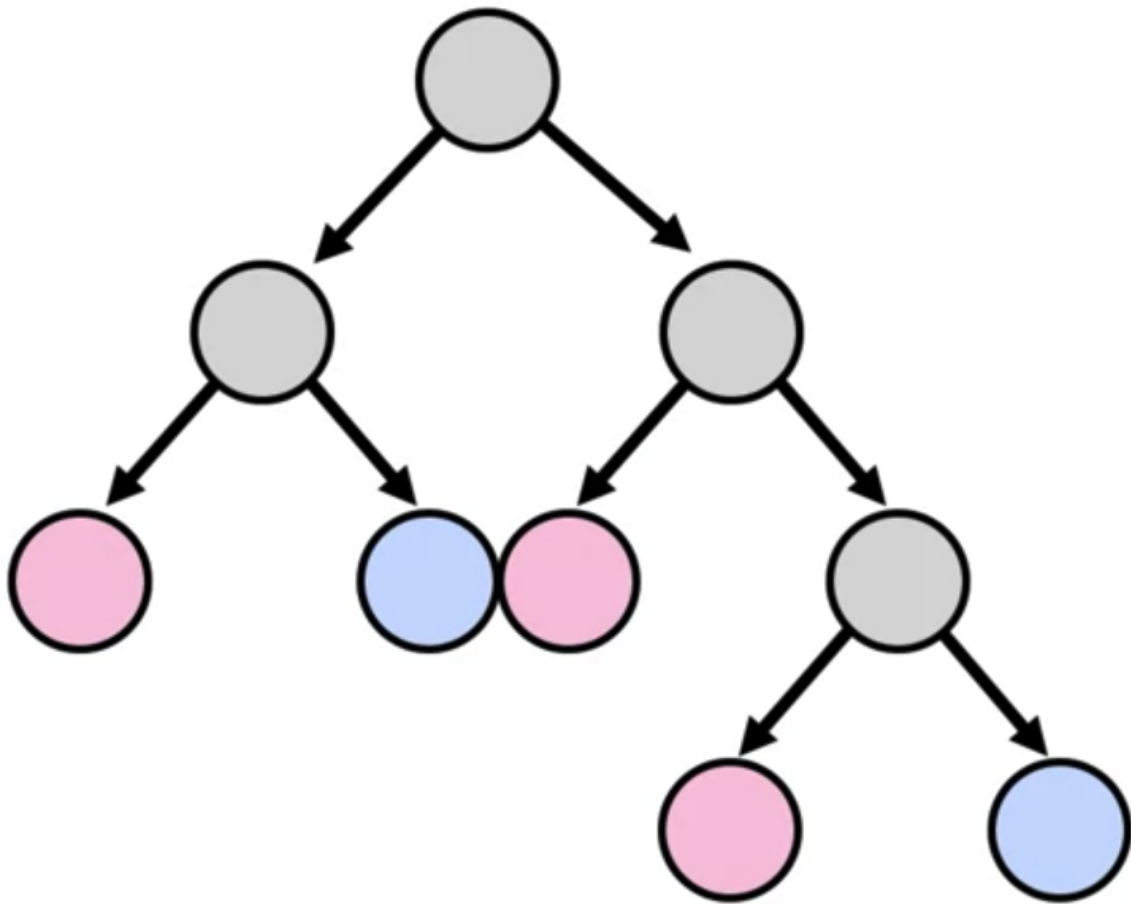


FIGURE 6.2: Decision tree layout

decision rules inferred from the data features. They start with the root of the tree and follow splits based on variable outcomes until a leaf node is reached and the result is given. Figure 6.2 details a visual representation of the decision tree layout.

6.2.3 Bootstrapping

Bootstrapping is the process of randomly sampling subsets of a dataset over a given number of iterations and a given number of variables. These results are then averaged together to obtain a more accurate result.

6.2.4 Random Forest Regression

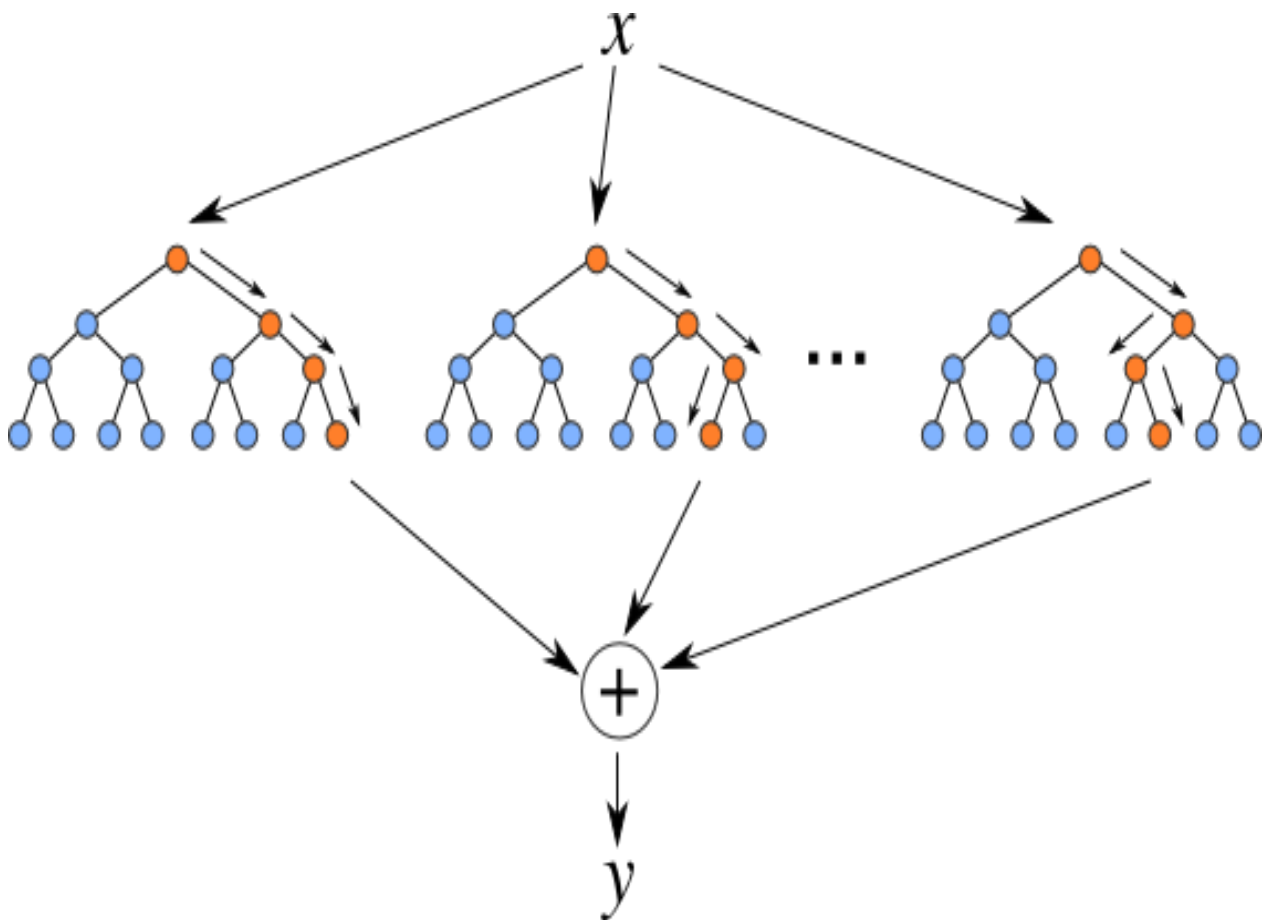


FIGURE 6.3: Simple RF

The bootstrapping Random Forest algorithm [8] combines ensemble learning methods with the decision tree framework to create multiple randomly drawn decision trees from the data, averaging the results to output a new result that often leads to strong predictions/classifications. Figure 6.2 documents a visual representation of random forest wherein \mathbf{X} is the feature vector and \mathbf{y} is the target vector.

6.3 MULTI LAYER PERCEPTRON

The Multi Layer Perceptron bases its fundamental design to the interlinking of several neurons (like structures) representing a Neural Network (NN) architecture. Given $i = 0, 1, \dots, n$ 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 6.4 shows the simplified representation of the above steps.

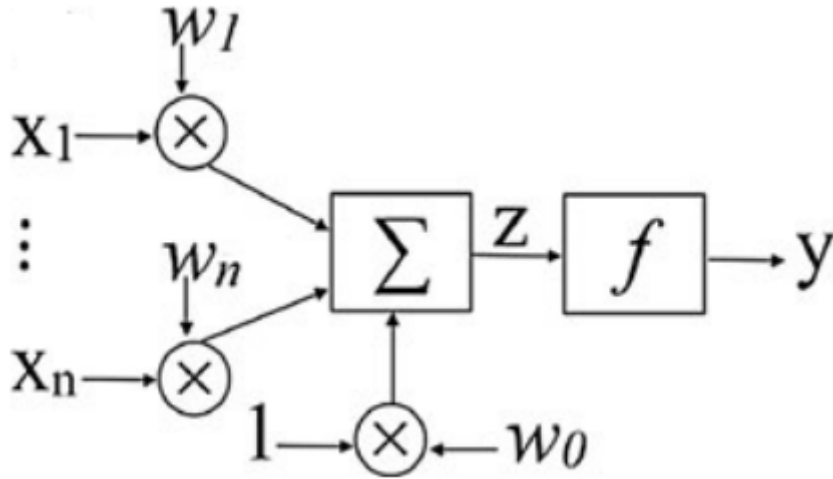


FIGURE 6.4: Perceptron model

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

$$y = f(z) \text{ and } z = \sum_{i=0}^n w_i x_i \quad (6.1)$$

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.

A perceptron can only learn linearly separable functions. For n dimensions, the function is a hyperplane with equation:

$$\sum_{i=0}^n w_i x_i = 0 \quad (6.2)$$

The motive of learning is to optimize the weights by minimizing a cost function, which is usually a square error between the known vector and the estimated vector. Optimization techniques such as gradient descent algorithm can be used to determine the optimum weight vector. Ultimately, the algorithm converges to a solution reaching an operational configuration network. However, the perceptron and the single layer perceptron do not resolve the nonlinearly separable problem.

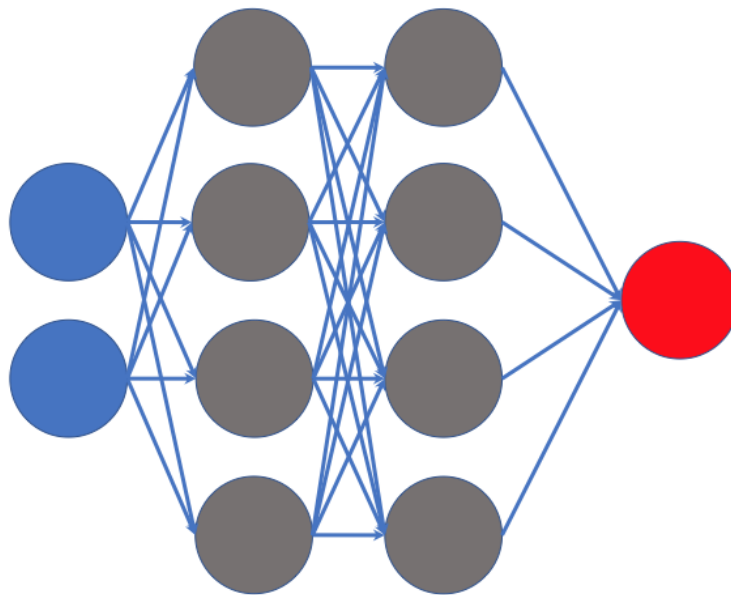


FIGURE 6.5: MLP model

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 [9] is a feedforward neural network that consists of three layers, the input layer, the hidden layer and the output layer. Figure 6.5 presents an MLP with two inputs (blue), two hidden layers (grey) and one output (red).

The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP. Similar to a feed forward network in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the back propagation learning algorithm, where the weights can be corrected by propagating the errors from layer to layer starting with the output layer and working backwards. MLPs are designed to approximate any continuous function and can solve problems which are not linearly separable.

CHAPTER 7

IMPLEMENTATION

The implementation of models mentioned in Chapter 6 was performed using the Scikit-Learn library [6]. For each code submission task, the dataset was split in the ratio 70:30 for training and testing. With respect to the 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 radial basis function kernel.

7.1 METRICS

7.1.1 Mean Absolute Error (MAE)

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

MAE is the mean of magnitude of difference between true value ' t_i ' and prediction ' y_i ' of 'n' observations.

7.1.2 Root Mean Absolute Error (RMSE)

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

RMSE is the square root of the mean of residuals (difference between true value ' t_i ' and prediction ' y_i ') of 'n' observations

7.2 RESULTS

The observed values of MAE and RMSE obtained for the different set of tasks are reported in sub sections 7.2.1 - 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.

7.2.1 Selection Sort

Model	<i>MAE</i>	<i>RMSE</i>
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)

TABLE 7.1: Results - Selection Sort

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

Model	<i>MAE</i>	<i>RMSE</i>
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)

TABLE 7.2: Results - First Negative Item in List

7.2.2 First Negative Item in List

From table 7.24, we observe that Random Forest outperforms SVM and MLP in performance.

7.2.3 Largest Item in List

Model	<i>MAE</i>	<i>RMSE</i>
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)

TABLE 7.3: Results - Largest Item in List

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

7.2.4 Unique Character count in a string

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

TABLE 7.4: Results - Unique Characters count in a string

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

7.3 INFERENCE

YET TO BE FILLED

CHAPTER 8

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 detail on the process of how constructive feedback is generated.

8.1 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/optimization.

8.2 ALGORITHM

Now, to generate feedback for a particular target program x with x_i features where $i = 0, 1, 2, \dots, n$ (n - number of features) - 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 $x_i > X_i$ - Feedback to decrease that particular feature in the program is advised.
 - If $x_i < X_i$ - Feedback to increase the particular feature in the program is advised.
- 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. Thereby the changes that are suggested as feedback are inferred to be optimization changes for the target vector and constructive feedback is provided.

CHAPTER 9

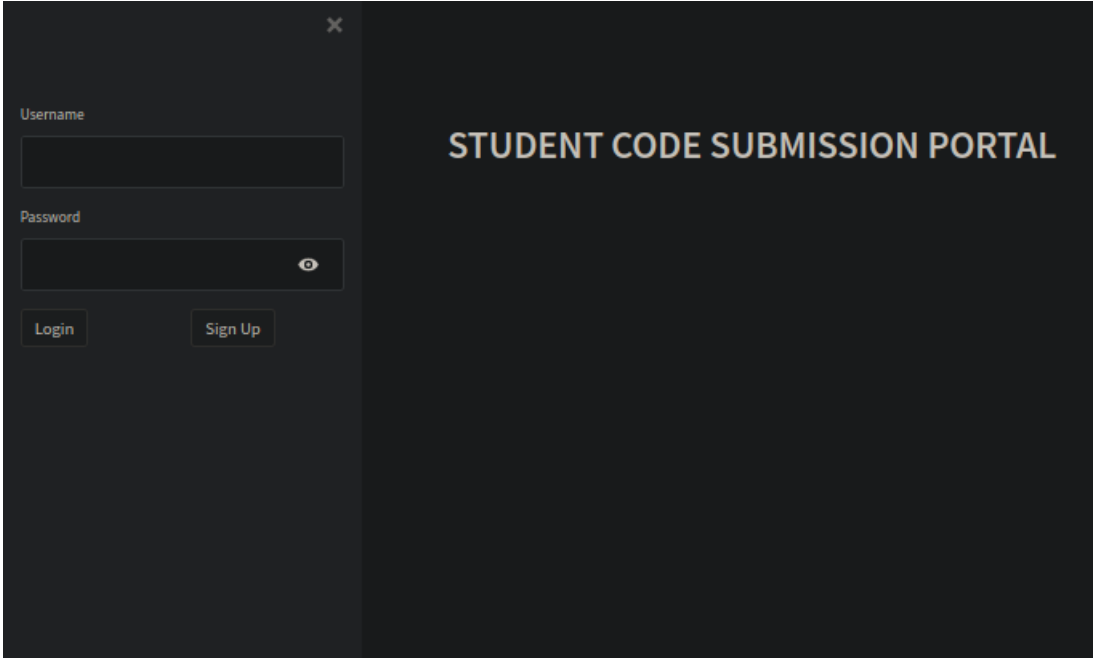
DEPLOYMENT

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 few lines of python code using Streamlit. It supports a lot of python frameworks and libraries like pandas, numpy, scikit-learn, Pytorch, Tensorflow etc.

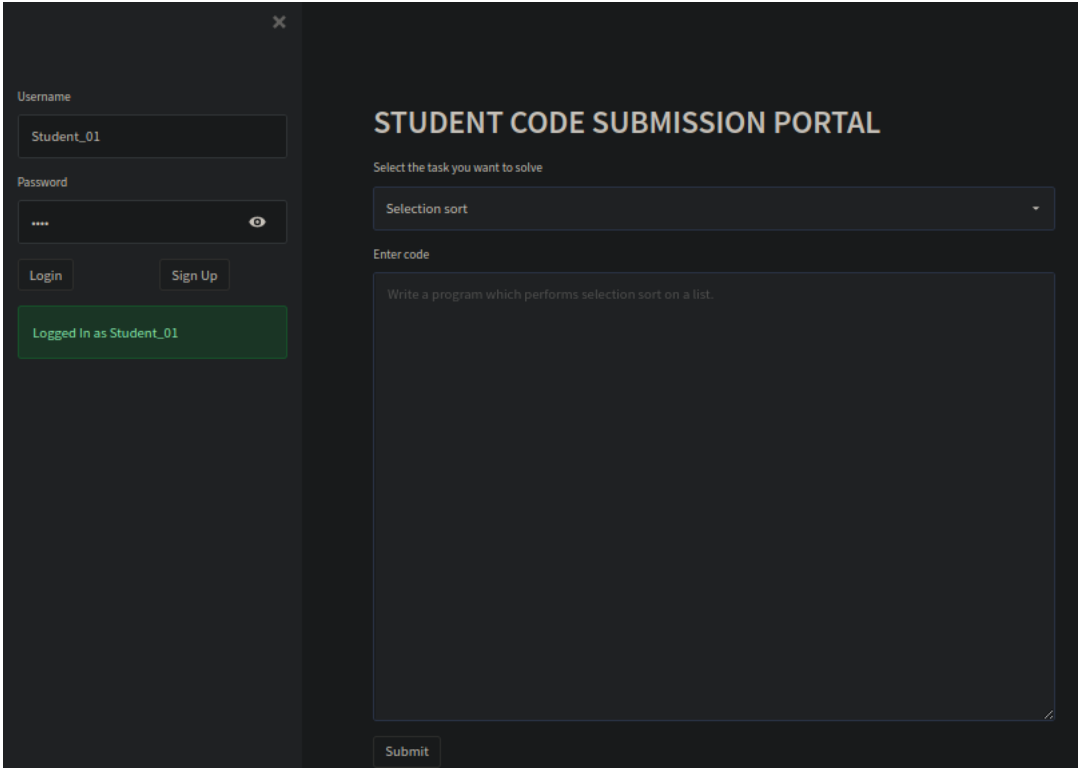
Another big advantage of Streamlit is that it provides a mechanism for caching - so that even when a expensive computation is done or a large dataset is manipulated, the app's performance will not be compromised. This is achieved with the use of the `st.cache` decorator.

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.



A dark-themed web interface for a student code submission portal. On the left, there is a login section with a close button (X) at the top. It contains two input fields: 'Username' and 'Password'. The 'Password' field has a toggle icon (an eye) to its right. Below these fields are two buttons: 'Login' and 'Sign Up'. The main area on the right has the title 'STUDENT CODE SUBMISSION PORTAL' in large, bold, white capital letters.

FIGURE 9.1: Snippet of Student Code Submission portal before login

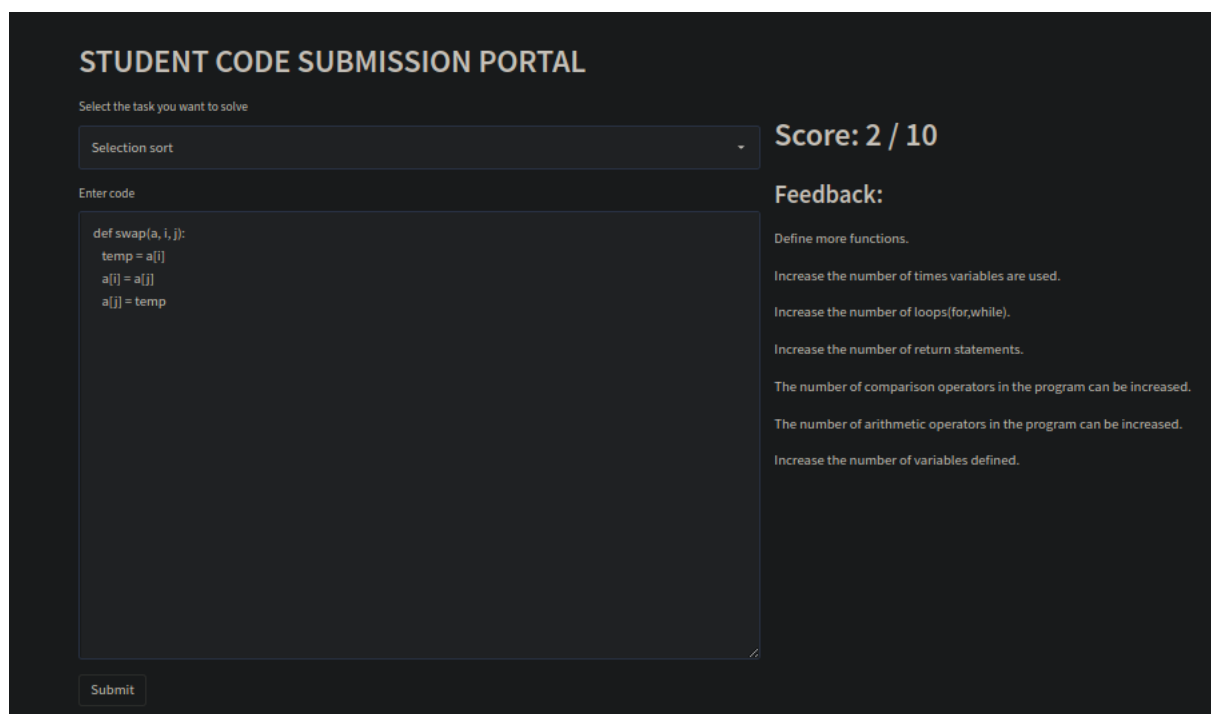


The same dark-themed web interface after a user has logged in. The login section on the left now shows the username 'Student_01' in the 'Username' field and masked characters '****' in the 'Password' field. A green notification box below the login buttons says 'Logged In as Student_01'. The main area on the right now displays the title 'STUDENT CODE SUBMISSION PORTAL' and a prompt 'Select the task you want to solve' above a dropdown menu showing 'Selection sort'. Below this is a section titled 'Enter code' with a text area containing the instruction 'Write a program which performs selection sort on a list.' and a 'Submit' button at the bottom.

FIGURE 9.2: 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.

Figure 9.2 is the snippet of the student code submission portal after 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
```

Submit

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.

FIGURE 9.3: Automatic Grading and Feedback Generation - Example 1

Figure 9.3 shows a 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 like to define more functions, add more looping statements etc.

STUDENT CODE SUBMISSION PORTAL

Select the task you want to solve

Selection sort

Score: 4 / 10

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
```

Submit

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.

FIGURE 9.4: Automatic Grading and Feedback Generation - Example 2

Figure 9.4 shows another example student submission for the election 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 looping statement, the code submission gets a grade of 4 out of 10 and the necessary code review feedbacks.

Figure 9.5 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.

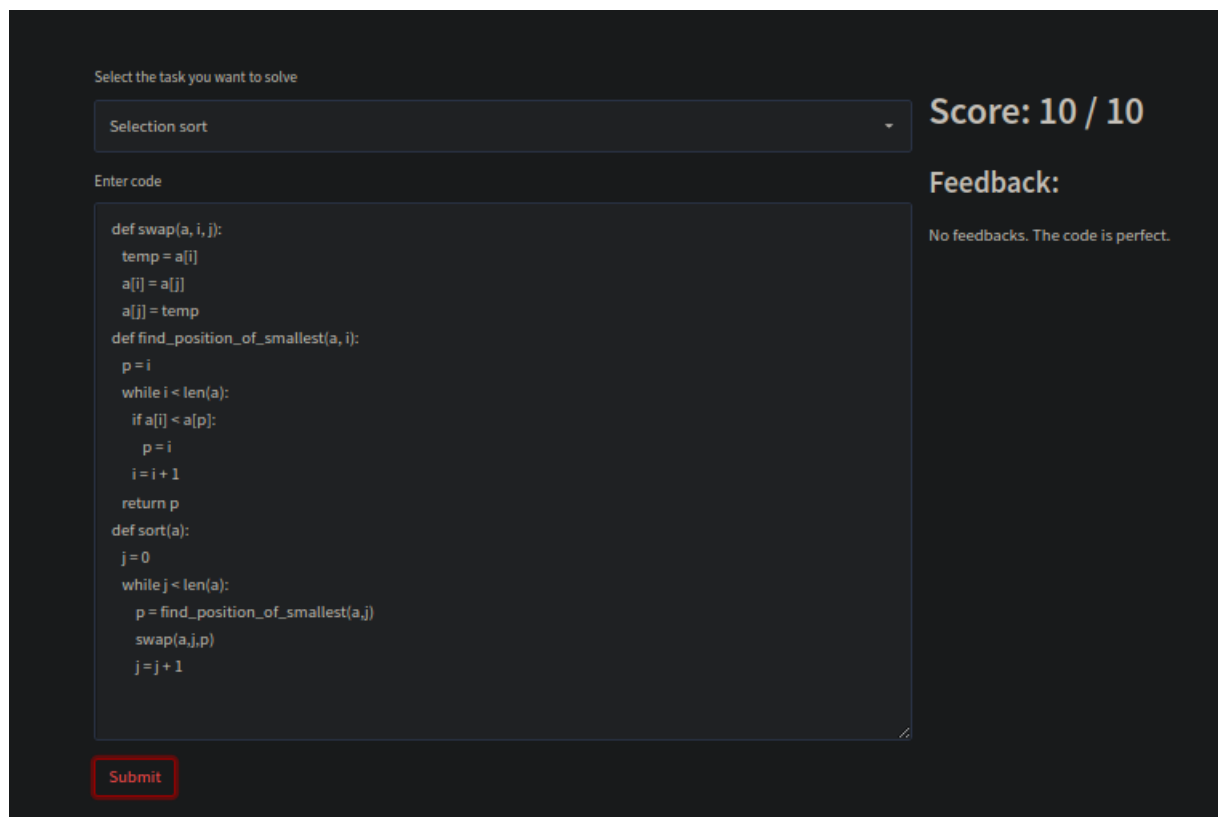


FIGURE 9.5: Automatic Grading and Feedback Generation - Example 3

Appendix A

CONCLUSION AND FUTURE WORK

REFERENCES

1. Azcona, D., Arora, P., Hsiao, I. H., & Smeaton, A. (2019, March). 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).
2. Orr, J. Walker, and Nathaniel Russell. "Automatic Assessment of the Design Quality of Python Programs with Personalized Feedback." arXiv preprint arXiv:2106.01399 (2021).
3. 2to3 - Automated Python 2to3 code translation - <https://docs.python.org/3/library/2to3.html>
4. AST module Python 3.10.4- <https://docs.python.org/3/library/ast.html>
5. Jović, Alan, Karla Brkić, and Nikola Bogunović. "A review of feature selection methods with applications." 2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO). Ieee, 2015.
6. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.
7. Drucker, Harris, et al. "Support vector regression machines." Advances in neural information processing systems 9 (1996).
8. Breiman, L. (2001). "Random forests - Machine Learning" 45, 5-32.
9. Murtagh, Fionn. "Multilayer perceptrons for classification and regression." Neurocomputing 2.5-6 (1991): 183-197.