

---

Masters

Science

---

2023

## Use of Machine Learning Methods in Automatic Assessment Programming Assignments

Botond Tarcsay

*Technological University Dublin, t.botond@yahoo.co.uk*

Follow this and additional works at: <https://arrow.tudublin.ie/scienmas>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Tarcsay, B. (2023). Use of Machine Learning Methods in Automatic Assessment Programming Assignments. Technological University Dublin. DOI: 10.21427/EQW1-3S76

This Theses, Masters is brought to you for free and open access by the Science at ARROW@TU Dublin. It has been accepted for inclusion in Masters by an authorized administrator of ARROW@TU Dublin. For more information, please contact [arrow.admin@tudublin.ie](mailto:arrow.admin@tudublin.ie), [aisling.coyne@tudublin.ie](mailto:aisling.coyne@tudublin.ie), [vera.kilshaw@tudublin.ie](mailto:vera.kilshaw@tudublin.ie).

# Use of Machine Learning Methods in Automatic Assessment Programming Assignments



**Botond Tarcsay**

Supervisor: Jelena Vasić

Fernando Perez Tellez

TU Dublin

This dissertation is submitted for the degree of  
*Master of Philosophy*

January 2023



## **Declaration**

I certify that this thesis which I now submit for examination for the award of Masters by Research, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This thesis was prepared according to the regulations for graduate study by research of the Technological University Dublin and has not been submitted in whole or in part for another award in any other third level Institution or University.

The work reported on in this thesis conforms to the principles and requirements of the TU Dublin's guidelines for ethics in research.

TU Dublin has permission to keep, lend or copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signed:



Botond Tarcsay  
January 2023



## **Acknowledgements**

I would like to express my sincere thanks to Jelena Vasić and Fernando Perez Tellez for their continuous help and hard work throughout the past two years. Their unparalleled guidance and assistance were crucial to this Thesis.

I would also like to thank all the TUD lecturers for their work, along with Dr. Stephen Blott for the platform explanation, David Azcona and Prof. Alan Smeaton for the dataset and Keith Quille for the brainstorming sessions.

Finally, I would like to thank my partner for her patience and support.



## **Abstract**

Programming has become an important skill in today's world and is taught widely both in traditional settings and online. Instructors need to assess increasing amounts of student work. Unit testing can contribute to the automation of the grading process; however, it cannot assess the structures, style and partially correct source code or differentiate between levels of achievement. The topic of this thesis is an investigation into the use of machine learning methods for assessing the correctness and quality of code, with the ultimate goal of assisting instructors in the grading process. In this research, we have used nine different machine learning algorithms, applied to three distinct types of feature sets, created from over five hundred thousand student code submissions. Prediction scores for some of the models show that the content of the submissions can be assessed in an automated manner. Along with unit testing, this approach has the potential to give instructors a source code-based automated way of assigning more finely differentiated grades than is possible by unit testing alone. This dissertation reports on several findings that confirm the validity of using machine learning, with features derived from source code tokens, for the evaluation of computer program correctness. Further, it shows how this approach has the potential to contribute to automated assessment with multi-valued measures of quality (grading schemes), in contrast to the binary pass/fail measure associated with unit testing.



## **List of Publications**

Part of this project was selected to be published in the peer-reviewed conference :

Tarcsay, B., Vasić, J. and Perez-Tellez, F., 2022. Use of Machine Learning Methods in the Assessment of Programming Assignments. In International Conference on Text, Speech, and Dialogue (pp. 151-159). Springer, Cham.

# Table of contents

|  |             |
|--|-------------|
| <b>List of figures</b>                             | <b>xiii</b> |
| <b>List of tables</b>                              | <b>xvii</b> |
| <b>1 Introduction</b>                              | <b>1</b>    |
| 1.1 Research Question . . . . .                    | 1           |
| 1.2 Research Objectives . . . . .                  | 2           |
| 1.3 Scope and Limitations . . . . .                | 2           |
| 1.4 Organization of the Thesis . . . . .           | 3           |
| <b>2 Literature Review</b>                         | <b>5</b>    |
| 2.1 Manual Grading . . . . .                       | 5           |
| 2.2 Automated Grading and Feedback Tools . . . . . | 7           |
| 2.3 Code Evaluation for Grading . . . . .          | 8           |
| 2.3.1 Semantic Comparison . . . . .                | 8           |
| 2.3.2 Unit Testing . . . . .                       | 9           |
| 2.4 Code Evaluation for Other Purposes . . . . .   | 11          |
| 2.5 Summary . . . . .                              | 14          |
| <b>3 Methodology</b>                               | <b>15</b>   |
| 3.1 Data . . . . .                                 | 17          |
| 3.1.1 Raw Data and ByteCode . . . . .              | 17          |
| 3.1.2 Derived Datasets . . . . .                   | 19          |
| 3.1.3 Preliminary Data Analysis . . . . .          | 24          |
| 3.2 Models . . . . .                               | 29          |
| 3.2.1 Models for Line Count Data . . . . .         | 29          |
| 3.2.2 Models for Token Count Data . . . . .        | 29          |
| 3.2.3 Models for Token Sequence Data . . . . .     | 30          |
| 3.2.4 Models for Multi-Class Label Data . . . . .  | 32          |

|          |   |           |
|----------|---|-----------|
| 3.2.5    | Model Type and Data Set Summary . . . . .                             | 33        |
| 3.3      | Generalisation Investigation . . . . .                                | 33        |
| 3.3.1    | Same Questions in Train and Test Subset . . . . .                     | 33        |
| 3.3.2    | Similar Questions in Train and Test Subset . . . . .                  | 34        |
| 3.3.3    | Different Questions in Train and Test Subset . . . . .                | 34        |
| <b>4</b> | <b>Results</b>  | <b>35</b> |
| 4.1      | Feature Selection . . . . .   | 35        |
| 4.2      | Model for Line Count Data . . . . .                                   | 38        |
| 4.3      | Models for Token Count Data . . . . .                                 | 38        |
| 4.3.1    | MNB - Token Count Results . . . . .                                   | 38        |
| 4.3.2    | SVM - Token Count Results . . . . .                                   | 39        |
| 4.3.3    | Logistic Regression - Token Count Results . . . . .                   | 40        |
| 4.3.4    | Decision Tree - Token Count Results . . . . .                         | 41        |
| 4.3.5    | Random Forest - Token Count Results . . . . .                         | 42        |
| 4.4      | Models for Token Sequence Data . . . . .                              | 43        |
| 4.4.1    | ANN Results - Token Sequence Data . . . . .                           | 43        |
| 4.4.2    | LSTM Results - Token Sequence Data . . . . .                          | 44        |
| 4.4.3    | RNN Results - Token Sequence Data . . . . .                           | 46        |
| 4.4.4    | CNN Results - Token Sequence Data . . . . .                           | 47        |
| 4.5      | Models for Multi-Class Label Data . . . . .                           | 48        |
| 4.5.1    | LSTM Results - Token Sequence Data - Multi-Class Prediction . . . . . | 48        |
| 4.5.2    | CNN Results - Token Sequence Data - Multi-Class Prediction . . . . .  | 50        |
| 4.6      | Result of Minimum Instance Requirement . . . . .                      | 52        |
| 4.7      | Result Comparison to Related Work . . . . .                           | 54        |
| <b>5</b> | <b>Conclusion</b>   | <b>55</b> |
| 5.1      | Proposal for Application . . . . .                                    | 57        |
| 5.2      | Future Work . . . . .   | 57        |
|          | <b>References</b>   | <b>59</b> |
|          | <b>Appendix A Resources Used</b>                                      | <b>63</b> |
| A.1      | Resources used . . . . .  | 63        |
|          | <b>Appendix B Algorithm Packages</b>                                  | <b>65</b> |
| B.1      | Versions used . . . . .   | 65        |

---

|  |           |
|--|-----------|
| <b>Appendix C Confusion Matrix Heatmaps</b>              | <b>67</b> |
| <b>Appendix D Source Code</b>                            | <b>79</b> |
| D.1 Tokenizer parameters . . . . .                       | 79        |
| D.2 Multinomial Naïve Bayes code . . . . .               | 79        |
| D.3 Support Vector Machine code . . . . .                | 81        |
| D.4 Logistic Regression code . . . . .                   | 83        |
| D.5 Decision Tree code . . . . .                         | 84        |
| D.6 Random Forest code . . . . .                         | 86        |
| D.7 Deep Learning - Tokenizer code and imports . . . . . | 88        |
| D.8 ANN code . . . . .                                   | 90        |
| D.9 LSTM code . . . . .                                  | 93        |
| D.10 RNN code . . . . .                                  | 96        |
| D.11 CNN code . . . . .                                  | 98        |
| D.12 LSTM Transfer Learning code . . . . .               | 101       |
| D.13 CNN Transfer Learning code . . . . .                | 104       |



# List of figures

|      |  |    |
|------|--|----|
| 2.1  | High level overview of SemCluster (Perry <i>et al.</i> (2019)) . . . . . | 8  |
| 2.2  | The workflow of Autograder (Liu <i>et al.</i> (2019)) . . . . .          | 9  |
| 2.3  | Architecture of ProgEdu system (Chen <i>et al.</i> (2018)) . . . . .     | 10 |
| 2.4  | InferCode knowledge transfer (Bui <i>et al.</i> (2020)) . . . . .        | 12 |
| 2.5  | High level overview of Type4PY (Mir <i>et al.</i> (2021)) . . . . .      | 13 |
| 3.1  | High Level Process Flow . . . . .  | 16 |
| 3.2  | Original Dataset in Json . . . . .                                       | 17 |
| 3.3  | Example of set ByteCode Length Set . . . . .                             | 20 |
| 3.4  | Example of ByteCode Token Count Set . . . . .                            | 20 |
| 3.5  | Example of Python Token Count Set . . . . .                              | 21 |
| 3.6  | Example of Python Keyword Set . . . . .                                  | 21 |
| 3.7  | Example of Python Keyword and Token Count Set . . . . .                  | 21 |
| 3.8  | Example of ByteCode Token Sequence Set Before Tokenization . . . . .     | 22 |
| 3.9  | Example of Python Token Sequence Before Tokenization . . . . .           | 23 |
| 3.10 | Example of Python Graded Token Sequence Before Tokenization . . . . .    | 23 |
| 3.11 | Number of Unigrams per Code Submission in Python code . . . . .          | 25 |
| 3.12 | Number of Bigrams per Code Submission in Python code . . . . .           | 25 |
| 3.13 | Number of Trigrams per Code Submission in Python code . . . . .          | 26 |
| 3.14 | Number of Quadrigrams per Code Submission in Python code . . . . .       | 26 |
| 3.15 | Number of Unigrams per Code Submission in ByteCode code . . . . .        | 27 |
| 3.16 | Number of Bigrams per Code Submission in ByteCode code . . . . .         | 27 |
| 3.17 | Number of Trigrams per Code Submission in ByteCode code . . . . .        | 28 |
| 3.18 | Number of Quadrigrams per Code Submission in ByteCode code . . . . .     | 28 |
| 3.19 | Token Count Process Flow . . . . .                                       | 29 |
| 3.20 | Token Sequence Process Flow . . . . .                                    | 30 |
| 3.21 | Transfer Learning Process Flow . . . . .                                 | 32 |

|      |   |    |
|------|---|----|
| 4.1  | Python Feature Importance - Decision Tree . . . . .   | 37 |
| 4.2  | ByteCode Feature Importance - Decision Tree . . . . .   | 37 |
| 4.3  | Python Minimum Instances Partial Example . . . . .  | 52 |
| 4.4  | Python Minimum Instances to F1 score . . . . .  | 53 |
| C.1  | Heatmap for Table 4.1 ANN - Code Length Results - Python . . . . .  | 67 |
| C.2  | Heatmap for Table 4.2 MNB - Token Count Results - Python Token and<br>Keyword Count Set . . . . .                   | 68 |
| C.3  | Heatmap for Table 4.3 MNB - Token Count Results - ByteCode . . . . .  | 68 |
| C.4  | Heatmap for Table 4.4 SVM - Token Count Results - Python Token and<br>Keyword Count Set . . . . .                   | 68 |
| C.5  | Heatmap for Table 4.5 SVM - Token Count Results - ByteCode . . . . .  | 69 |
| C.6  | Heatmap for Table 4.6 Logistic Regression - Token Count Results - Python<br>Token and Keyword Count Set . . . . .   | 69 |
| C.7  | Heatmap for Table 4.7 Logistic Regression - Token Count Results - ByteCode  | 69 |
| C.8  | Heatmap for Table 4.7 Logistic Regression - Token Count Results - ByteCode  | 70 |
| C.9  | Heatmap for Table 4.8 Decision Tree - Token Count Results - Python Token<br>and Keyword Count Set . . . . .         | 70 |
| C.10 | Heatmap for Table 4.9 Decision Tree - Token Count Results - ByteCode . .  | 70 |
| C.11 | Heatmap for Table 4.10 Decision Tree - Keyword Count Results - Python -<br>Question Independent . . . . .           | 71 |
| C.12 | Heatmap for Table 4.11 Random Forest - Keyword and Token Count Results<br>- Python . . . . .                        | 71 |
| C.13 | Heatmap for Table 4.12 Random Forest - Token Count Results - ByteCode .   | 71 |
| C.14 | Heatmap for Table 4.13 Random Forest - Keyword and Token Count Results<br>- Python - Question Independent . . . . . | 72 |
| C.15 | Heatmap for Table 4.14 ANN - Token Sequence Results - Python . . . . .  | 72 |
| C.16 | Heatmap for Table 4.15 ANN - Token Sequence Results - ByteCode . . . . .  | 72 |
| C.17 | Heatmap for Table 4.16 LSTM - Token Sequence Results - Python . . . . .   | 73 |
| C.18 | Heatmap for Table 4.17 LSTM - Token Sequence Results - ByteCode . . . .   | 73 |
| C.19 | Heatmap for Table 4.18 LSTM - Token Sequence Results - Python - Question<br>Independent . . . . .                   | 73 |
| C.20 | Heatmap for Table 4.19 LSTM - Token Sequence Results - ByteCode -<br>Question Independent . . . . .                 | 74 |
| C.21 | Heatmap for Table 4.20 RNN - Token Sequence Results - Python . . . . .  | 74 |
| C.22 | Heatmap for Table 4.21 RNN - Token Sequence Results - ByteCode . . . . .  | 74 |
| C.23 | Heatmap for Table 4.22 CNN - Token Sequence Results - Python . . . . .  | 75 |

---

|   |    |
|---|----|
| C.24 Heatmap for Table 4.23 CNN - Token Sequence Results - ByteCode . . . . .                             | 75 |
| C.25 Heatmap for Table 4.24 CNN - Token Sequence Results - Python - Question<br>Independent . . . . .     | 75 |
| C.26 Heatmap for Table 4.25 CNN - Token Sequence Results - ByteCode - Ques-<br>tion Independent . . . . . | 76 |
| C.27 Heatmap for Table 4.27 LSTM - Token Sequence Results - Transfer Learning<br>- Python . . . . .       | 76 |
| C.28 Heatmap for Table 4.28 LSTM - Token Sequence Results - Transfer Learning<br>- Python . . . . .       | 76 |
| C.29 Heatmap for Table 4.30 CNN - Token Sequence Results - Transfer Learning<br>- Python . . . . .        | 77 |
| C.30 Heatmap for Table 4.31 CNN - Token Sequence Results -Transfer Learning -<br>Python . . . . .         | 77 |





# List of tables

|      |  |    |
|------|--|----|
| 3.1  | Algorithm and Dataset Relation Matrix . . . . .  | 33 |
| 4.1  | ANN - Code Length Results - Python . . . . .   | 38 |
| 4.2  | MNB - Token Count Results - Python Token and Keyword Count Set . . . .                       | 39 |
| 4.3  | MNB - Token Count Results - ByteCode . . . . .   | 39 |
| 4.4  | SVM - Token Count Results - Python Token and Keyword Count Set . . . .                       | 39 |
| 4.5  | SVM - Token Count Results - ByteCode . . . . .   | 40 |
| 4.6  | Logistic Regression - Token Count Results - Python Token and Keyword<br>Count Set . . . . .  | 40 |
| 4.7  | Logistic Regression - Token Count Results - ByteCode . . . . .                               | 40 |
| 4.8  | Decision Tree - Token Count Results - Python Token and Keyword Count Set                     | 41 |
| 4.9  | Decision Tree - Token Count Results - ByteCode . . . . .                                     | 41 |
| 4.10 | Decision Tree - Keyword Count Results - Python - Question Independent .                      | 42 |
| 4.11 | Random Forest - Keyword and Token Count Results - Python . . . . .                           | 42 |
| 4.12 | Random Forest - Token Count Results - ByteCode . . . . .                                     | 42 |
| 4.13 | Random Forest - Keyword and Token Count Results - Python - Question<br>Independent . . . . . | 43 |
| 4.14 | ANN - Token Sequence Results - Python . . . . .  | 43 |
| 4.15 | ANN - Token Sequence Results - ByteCode . . . . .  | 44 |
| 4.16 | LSTM - Token Sequence Results - Python . . . . .   | 44 |
| 4.17 | LSTM - Token Sequence Results - ByteCode . . . . .   | 45 |
| 4.18 | LSTM - Token Sequence Results - Python - Question Independent . . . . .                      | 45 |
| 4.19 | LSTM - Token Sequence Results - ByteCode - Question Independent . . . .                      | 45 |
| 4.20 | RNN - Token Sequence Results - Python . . . . .  | 46 |
| 4.21 | RNN - Token Sequence Results - ByteCode . . . . .  | 46 |
| 4.22 | CNN - Token Sequence Results - Python . . . . .  | 47 |
| 4.23 | CNN - Token Sequence Results - ByteCode . . . . .  | 47 |
| 4.24 | CNN - Token Sequence Results - Python - Question Independent . . . . .                       | 48 |

|   |    |
|---|----|
| 4.25 CNN - Token Sequence Results - ByteCode - Question Independent . . . . . | 48 |
| 4.26 LSTM - Token Sequence Results - Transfer Learning - Python . . . . .     | 49 |
| 4.27 LSTM - Token Sequence Results - Transfer Learning - Python . . . . .     | 49 |
| 4.28 LSTM - Token Sequence Results - Transfer Learning - Python . . . . .     | 50 |
| 4.29 CNN - Token Sequence Results - Transfer Learning - Python . . . . .      | 50 |
| 4.30 CNN - Token Sequence Results - Transfer Learning - Python . . . . .      | 51 |
| 4.31 CNN - Token Sequence Results -Transfer Learning - Python . . . . .       | 51 |

# Chapter 1

## Introduction

This dissertation reports on the research of machine learning methods for the evaluation of source code, specifically source code submitted by students as solutions to first and second-year Python programming assignments. The aim is to provide an in-depth analysis of several methods that may assist instructors with the grading of Python student coding submissions. Our vision is to identify the possibility of a universal grading method using source code submissions only, where the instructors will not have to provide good solutions or built lengthy unit tests.

### 1.1 Research Question

Most widely used existing systems evaluate code submissions in terms of whether the code is runnable, whether the syntax is correct, whether the output matches the expected values and whether they are similar to the gold standard solution provided by the instructor. This research is aiming to go a step further, to find partially correct code in programs that failed unit tests and to differentiate between programs that passed, by building machine learning models of the source code, which would facilitate a useful refinement of automated grading systems.

The theory underlying this research is that the information about code quality contained in a program can be learnt by machine learning algorithms and translated into a quantitative measure (the grade) straight from the source code.

The question that the research aims to answer is whether it is possible to measure the level of correctness of code using machine learning and, if so, under what constraints.

## 1.2 Research Objectives

Research Objectives define the most important tasks completed throughout the research.

Main objectives are:

1. Prepare different datasets and transform them for further processing by the Machine Learning (ML) models
2. Build ML models for dichotomous prediction and determine which model is the most suitable to predict student submission outcome
3. Build ML models for fine-graded prediction by using and transforming the best-performing dichotomous models
4. Investigate generalisability, in terms of the required level of functional similarity between the assessed code and the code used for model training
5. Investigate performance issues and how they can be addressed

## 1.3 Scope and Limitations

The research uses the dataset from Azcona *et al.* (2019) but for a different purpose. The data contain only pass/fail labels, while the goal of the research is to establish if fine grades can be predicted. The dataset, although it is very large, spanning across 3 years with more than 600 questions, was generated by the same unit testing program with test cases created by teachers from the same university.

Another limitation was in the computing resources available for the research, which are listed in Appendix A.1. As the machines on which the algorithms were trained were not industrial scale, the number of features had to be reduced for the training processes to execute successfully, due to memory, CPU or time limitations. This might have led to information loss, which cannot be validated independently.

## 1.4 Organization of the Thesis

The dissertation is organised as follows.

- Chapter 1 - Introduction: This chapter gives an overview of the research, introducing the research questions and objectives. We discuss why the question is important and what were we aiming to achieve.
- Chapter 2 - Literature Review: This chapter surveys the literature related to this research and discusses how and why these papers are important. We outline the current state of the art and the existing approaches and processes. We aim to discuss more recent developments in more detail.

The chapter is broken down into sections by code evaluation purposes, separating the literature for grading and other purposes. manual grading, automated grading and feedback tools are also discussed before the chapter is summarised.

- Chapter 3 - Methodology: Here we discuss the models and datasets used and present how our approach evolved during the research from binary to multi-class predictions. The first of the three main sections discuss the raw data in detail, introducing the datasets created and the preprocessing steps taken. The second section breaks down the models used, grouped by data set type, while the third section discusses the investigation into the generalisation of learning in this context.
- Chapter 4 - Results: This chapter provides the results of the research. Apart from an explanation on feature selection, it discusses the prediction success of all the fitted models, grouped by data set type. Tables containing the best-achieved results are presented, discussing relevant details and findings.
- Chapter 5 - Conclusion: Here the results are summarised, and discussed in the context of the research question. A possible application of the work is proposed, as well as future research possibilities based on this topic.



# Chapter 2

## Literature Review

This chapter presents a review of the literature. The interest in evaluating and characterising code is as old as programming itself, with publications discussed here spanning a period of almost fifty years up to the present day.

Manually marking student coding submissions is a tedious process and it is based on the teacher's individual teaching style. This area would greatly benefit from automation, even if it is partial, to speed up and standardise the steps. Coding itself is based on deterministic goals, therefore the topic is a great candidate for automation.

The first papers on automatic code assessment were published in 1960 by Hollingsworth (1960), but we are going to concentrate only on some of the most recent papers and articles, to give an overview of the ever-changing landscape and the latest progress.

A very recent paper by Combéfis (2022) reviews the related literature on how automatic grading works and what effect it has on the education system. The author discussed how automated tools can be categorised, and what challenges are there and summarised 127 papers on this topic. It was concluded that based on the literature there is no evidence that automated code assessment can or can not produce as good results as human graders.

### 2.1 Manual Grading

Manual grading is the traditional method for evaluating code written by students. Student submissions are reviewed by the instructors and graded based on the teacher's own knowledge, expectations and the way they have taught that particular subject. Manual submissions for



manual grading can have different issues on both the teacher's and the student's side. Either a prescribed format needs to be used for consistency, which has to be implemented by the students, or the teacher has to devise a method to collect submissions in a consistent manner. Therefore some universities have set rules regarding submissions and grading those submissions. The main gradable elements are generally the same, such as how well the program is designed, or if it is running, but most of the teachers have their own approach, based on the task or exercise at hand. Depending on the module and the material that is being tested, the approach can involve only logical testing, excluding syntax, or very basic unit testing, where the program is tested based on variables and outcomes, by a script written by the instructor. Although the functional and non-functional requirements are well specified, the percentage of how and what is graded can be different from instructor to instructor. These differences exist even if automated coding submissions are in place, as instructors can use different methods within a university, as concluded by Wilcox (2015).

To understand formalised grading systems, we have researched several institutions, grading methods (Stanford (2017), UWO (2018), NCCEDU (2017) NCL (2020), UT (2017) and UIUC (2020) ) and established a baseline for grading. The below list summarises the most common approaches. The main observation that can be concluded based on these guidelines is that even if the method is well defined (however different it is between teachers and universities), we could not find any guidelines regarding partial correctness. This seems to be left for the individual instructors to decide.

#### 1. Method

- Working Program
- Program Structure
- Program documentation
- Program Style

#### 2. Method

- Program Design
- Program execution
- Specification Satisfaction
- Coding Style
- Comment

#### 3. Method

- Instructor defined
- Keyword strategy
- Ordered Keyword strategy

#### 4. Method

- Code Analysis
- Plagiarism check
- Functionality check
- Code quality check

## 5. Method

- Correctness
- Execution Time performance
- Memory Consumption
- Robustness
- Elegance

## 6. Method

- Successful compilation
- Matching code construct to the topic being assessed
- Passing the test cases

## 2.2 Automated Grading and Feedback Tools

Automated grading has become common based on the reviewed papers, especially for online courses, where the number of student submissions is extremely high. Using an automatic grading system reduces the resources required throughout the entire process of grading, from collecting submissions to evaluating them, without negatively impacting academic performance. This was concluded by Wilcox (2015) based on their research at their own university. In this section, we discuss the available tools which generate feedback along with a grade.

Piech *et al.* (2015) proposes deep learning for grading and simultaneously providing feedback to students on coding submissions. They use NPM-RNNs with embedding and encoders to train their algorithms on a vast dataset (over 21 million submissions). They have achieved a high precision however the recall was 44 % which is below the 50 % mark. Their feedback was based on similarly correct code annotations extracted from correct code, which seems to be a reliable approach based on the reported results showing consistent correct classification. They have also found that KNNs on AST edits are computationally very expensive and not warranted to use due to their relatively poor precision.

There are systems in place which are well-developed and used by a wide range of students, instructors and professionals. One of the most widely used is WEB-Cat, developed by Edwards and Perez-Quinones (2008), an open-source system for a variety of programming languages. The web-based system is used by more than a hundred universities and is extremely versatile due to its plug-ins. OverCode by Glassman *et al.* (2015) is also used for student code checking based on similarity clustering and GraderScope by GraderScope (2020) is an excellent choice for unit testing, all of them providing instant feedback for the students in an iterative manner.

## 2.3 Code Evaluation for Grading

In this section we discuss papers that investigate student submission grading. The section is further divided into two subsections, based on how the code evaluation for grading is achieved.

### 2.3.1 Semantic Comparison

This subsection discusses papers, that deal with the semantic comparison of code.

SemCluster (by Perry *et al.* (2019)) uses two different clustering algorithms to understand how the pieces of source code are partitioned into classes and how these classes address the final outcome as Figure 2.1 shows. Their work attempted to address the problem of clustering correct code with incorrect code in the same group.

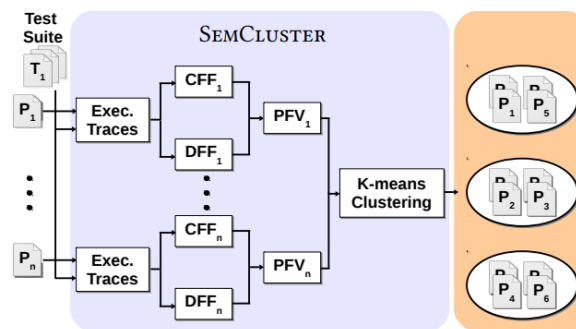


Fig. 2.1 High level overview of SemCluster (Perry *et al.* (2019))

CFF in Figure 2.1 refers to Control Flow Features, representing the aspect of a solution strategy. DFF is the Data Flow Feature and these are merged into a final program feature vector (PFV) which is then fed into the clustering algorithm. Their proposal was evaluated by using programming submissions from various online sources. The tool created significantly fewer clusters than other similar tools, such as Glassman *et al.* (2015), which we discuss later. Perry *et al.* (2019)'s approach takes into account the degree of similarity within clusters and uses a different representation of the submission, which led to better results according to their own validation, as they used a vector representation of programs, based on semantic program features.

Liu *et al.* (2019) use a formal-semantics-based approach to grade submissions based on a reference implementation and the semantic difference between the reference and the student code.

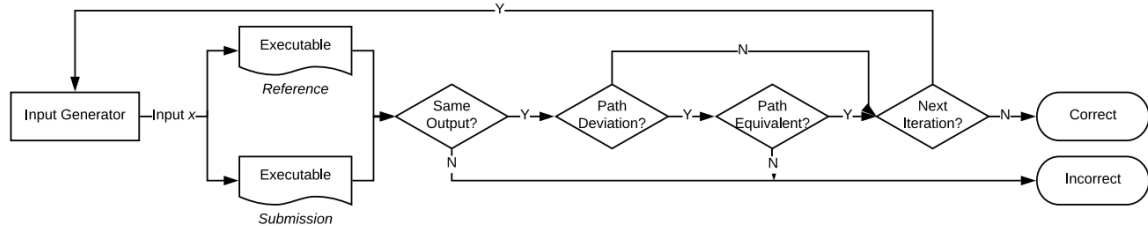


Fig. 2.2 The workflow of Autograder (Liu *et al.* (2019))

As Figure 2.2 shows, their iterative approach generates inputs (using an open-source white-box fuzzer (Pathgrind by Codellion (2012))) for the code to run and they investigate the inputs that lead to path deviation using Symbolic Execution. Their approach is grading the functional correctness of the submission. The research used 10 test cases which is a low sample size to create a fully generalizable solution.

There are also question-independent solutions to predict the correct grade proposed by Rai *et al.* (2019) using best practices and closeness to the expected response by logic, data dependencies and bag of words. The authors used Control Flow Graphs and Data Flow Graphs, which were translated to a common format using Bag of Words. This was divided into smaller sections and compared to the good examples using different ML models, with Linear Kernel Support performing the best. Although the results were promising, predicting the correct outcome with high consistency, the number of questions and samples used was low.

An even more complicated task was taken on by Drummond *et al.* (2014). The team was looking for a solution to grade student submissions of an online course for interactive game design in Python. They had to consider not only the correctness of the code submission but also if the game was playable. They achieved the goal with statistical analysis, breaking down the code into code fragments and carefully setting up the data which was then used in Bayesian models and kNN regression models.

### 2.3.2 Unit Testing

Analyzing source code is a vital element in assisting instructors in grading it. Solving the problem of grading students' coding submissions at scale has also been a widely researched subject, due to the increasing number of online programming courses. Based on the number of papers reviewed, the most popular approach is still unit testing.

A good example of a unit testing framework is introduced in a paper proposing a method called ProgEdu by Chen *et al.* (2018). This framework deals with an automated code quality assessment and student feedback through a web-based portal, catering to the end-to-end needs of both teachers and students.

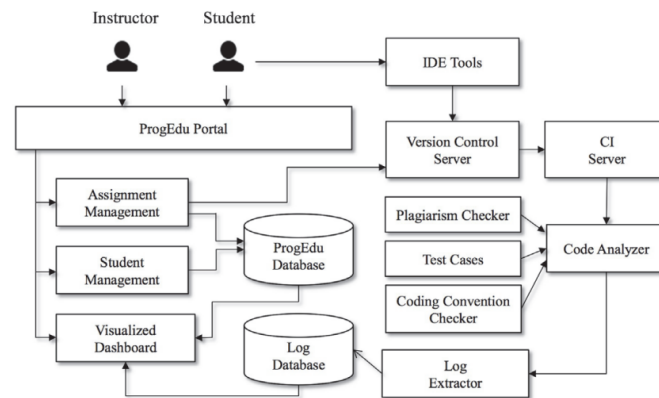


Fig. 2.3 Architecture of ProgEdu system (Chen *et al.* (2018))

As Figure 2.3 shows, both instructors and students use the same portal. This enables the teachers to set up unit testing and students to submit their own solutions in an iterative manner. The system checks for plagiarism and checks whether the code is executable, passes the unit tests and complies with coding conventions and best practices. The system is using best practices in Java along with the instructor's unit tests.

Teaching programming with automated feedback is also explored by Fangohr *et al.* (2015), using a special unit testing framework where the teachers can create their own unit testing process. This automated unit testing approach requires students to send an email with the code in a specific format, which then gets tested in a sandbox environment, running through the scripts prepared by the teacher. The students then receive the results and feedback on the correctness of the code. The framework highlights the pieces of code where the students can improve, based on how the code performed in the unit testing phase.

In another paper, Hegarty-Kelly and Mooney (2021) discuss techniques to evaluate, run and grade code using sophisticated scripts by checking syntax, outcome and closeness to the gold standard answer. The authors' approach was similar to the topic of this thesis, as they were interested in finding partially gradable parts of the code, even if the overall code failed. They used grep (which is a bash scripting language process searching the code for

patterns) with regular expressions (RegEx) to capture concepts in Java that corresponds to the questions proposed. This way, even if the syntax was slightly incorrect and the code didn't compile, the students got awarded a partial grade. Their addition to the regular unit testing framework generated positive feedback in their university both from teachers and students according to their questionnaire which measured the success of the project.

Interestingly enough the topic of grading student submissions is well researched not just in the coding space, but in relation to STEM subjects as well, as outlined by Lan *et al.* (2015). They used Mathematical Language Processing (MLP) (which is very similar to NLP but out of scope for our Thesis) and clustering to determine the closeness to the correct solution.

There is further research on the field of submission grading, which has been conducted for many years and notable papers are by Huang *et al.* (2013), by Srikant and Aggarwal (2013), by Srikant and Aggarwal (2014), by Glassman *et al.* (2015) and by Singh *et al.* (2016) dealing with similar approaches on automatic grading of student submissions.

## 2.4 Code Evaluation for Other Purposes

How a program should be written and how to manage program complexity has been researched for almost half a century, with articles such as the one written by McCabe (1976), who used graph-theoretic complexity measure to determine how complex a given piece of source code is. Since then coding, in general, has changed significantly, however, the main principles still stand.

Bui *et al.* (2020) examines the use of self-supervised machine learning algorithms with NLP, for processing Abstract Syntax Trees of pieces of code. This approach uses Tree-based Convolutional Neural Networks (TBNNs) as an encoder to process the ASTs and can be useful for several applications, such as code clustering, code clone detection, cross-language code-to-code search in unsupervised form and code classification and method name prediction in supervised form. This paper devises a process that can be replicated for any language with well-defined ASTs, as they use the sub-branches of the AST as a label.

As Figure 2.4 shows, once the source code features are learned through the proposed TBCNN encoder, these can be transferred to a supervised model for further fine-tuning to increase accuracy. The results are very promising, consistently reported to reach a higher accuracy than Code2Vec by Alon *et al.* (2019).

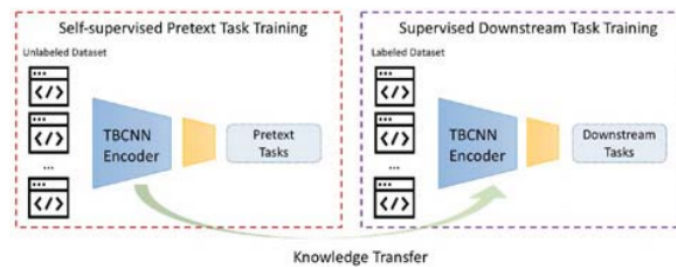


Fig. 2.4 InferCode knowledge transfer (Bui *et al.* (2020))

Different approaches were employed to solve different but related problems, from finding security vulnerabilities directly in source code using Neural Network Language Model-guided Engine Fuzzer with Montage by Lee *et al.* (2020) to information retrieval techniques for plagiarism detection by Jayapati and Venkitaraman (2019), and by Setoodeh *et al.* (2021) and code review to predict if a code change will be approved, with the use of CNN (Li *et al.* (2019)).

Looking at code and discovering mistakes to establish if a rework is required is explored by Madera and Tomoń (2017). They are using several data points including the code, the related project description and version management details extracted from the whole project document with WEKA by Hall *et al.* (2009). This overarching approach investigated the coding projects as a whole and used every available data provided by the company they worked with. They aimed to alleviate the need for rework before it happened, saving time and resources in the application development life-cycle.

A more recent paper related to Python code proposes a Deep Similarity Learning approach by Mir *et al.* (2021) using word2vec by Mikolov (2015) and word embeddings. The authors created type hints, based on natural information and code context, using ASTs, names of functions and dependency graphs.

As Figure 2.5 shows, they extracted the information straight from the source code. After pre-processing, they used a hierarchical neural network with bidirectional LSTM to implement two RNNs along with a type clustering process with Triplet Loss by Cheng *et al.* (2016). After creating the model, the approach is able to provide contextual and natural type hints along with argument, variable and return predictions. Based on their performance evaluation, this approach performs significantly better than Typilus by Allamanis *et al.* (2020) or Typewriter by Pradel *et al.* (2020), which they measured the performance against.

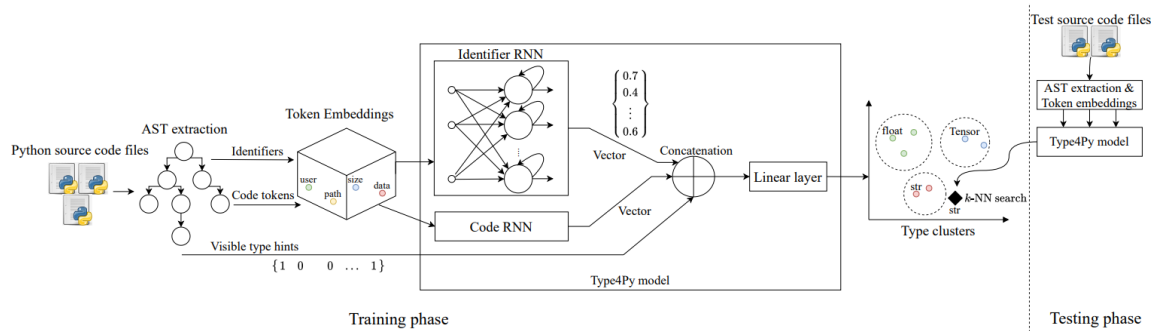


Fig. 2.5 High level overview of Type4PY (Mir *et al.* (2021))

Kharat *et al.* (2019) deals with a systematic source code review solution using static code analysis, rules and classifiers to identify the coding irregularities based on peer review and student submissions.

A paper presenting a comprehensive review of thirty different statistical analysis methods and tools explored by Ragkhitwetsagul *et al.* (2018) addresses the issue of plagiarism and code cloning by comparing the different statistical methods and discussing their strength and weaknesses.

An interesting method by Kapur *et al.* (2021) using Paragraph Vector Algorithm to compare source code by finding similar code on StackOverflow and compare the code snippets to each other in order to improve their own (previously developed) tool, CRUSO. With their new tool, CURSO-P, the authors aim to predict if a code will be defective more accurately using fewer resources. According to their evaluation, CURSO-P achieved 5% more accuracy (from 94% to 99%) reducing response time and storage needs by almost 90%.

While source code grading is extremely important, writing correct code does not stop at the learning stage. Most of the integrated development environments (IDE) contain their own code correctness check, such as PyCharm by Nguyen (2019), a desktop-based IDE and there are online solutions such as SonarQube by Campbell and Papapetrou (2013), Checkmarkx by Ye *et al.* (2016) or Checkstyle by Checkstyle (2017) to help programmers detect bugs or security vulnerabilities by uploading their code to the cloud.



## 2.5 Summary

To achieve the research goals, and understand the current area of research, more than 150 papers and articles were reviewed. The relevant ones were highlighted in this chapter, discussing their results or implications. It was found that the majority of current systems and proposals for grading of code are unit testing based, but there is also a vast amount of research in Machine Learning based proposals using ASTs and example solutions provided by the teachers. There is little research that draws on source code tokens and their sequences directly as the basis for ML features, even though for simple problems this would seem to be a viable approach.

Our review also highlighted a gap in research into measuring partial correctness. Most manual grading methods do not specifically cater to partial correctness in a well-defined manner, leaving the decision to the teachers, while automated grading does not seem to have solved this task, based on our current knowledge.

Recently there has been significant progress in the world of automated grading using machine learning and deep learning based on the literature. Using machine learning or deep learning algorithm libraries such as *keras* as described in the book by Gulli and Pal (2017) is a sophisticated way to check code. This might lead to a generalizable solution, which can assist teachers in a language-independent and unbiased way to grade coding submissions.

# Chapter 3

## Methodology

In this chapter the methodology used to conduct the research is described and discussed. The aim of our research is to investigate whether it is possible to measure the level of correctness of code directly from source code tokens with the use of using machine learning and, if so, under what constraints. To explore which of several model types is best for this goal, we first had to prepare data in several ways so as to have suitable features for the training of each of these model types. Our choice of models was driven by the models' unique capabilities as discussed throughout this section. In the first instance, the models were trained for binary prediction with labels available from a unit testing system. The most successful models from this stage were built upon through additional training with multi-class-labelled data, for finer-grained grading. In this section we present the data sets and models trained on these data sets.

A high-level overview of the process can be seen in Figure 3.1. This flow includes data analysis, code compilation and data preparation for the different datasets and shows all the algorithms used to reach the final results including bi-nominal and multi-class predictions.

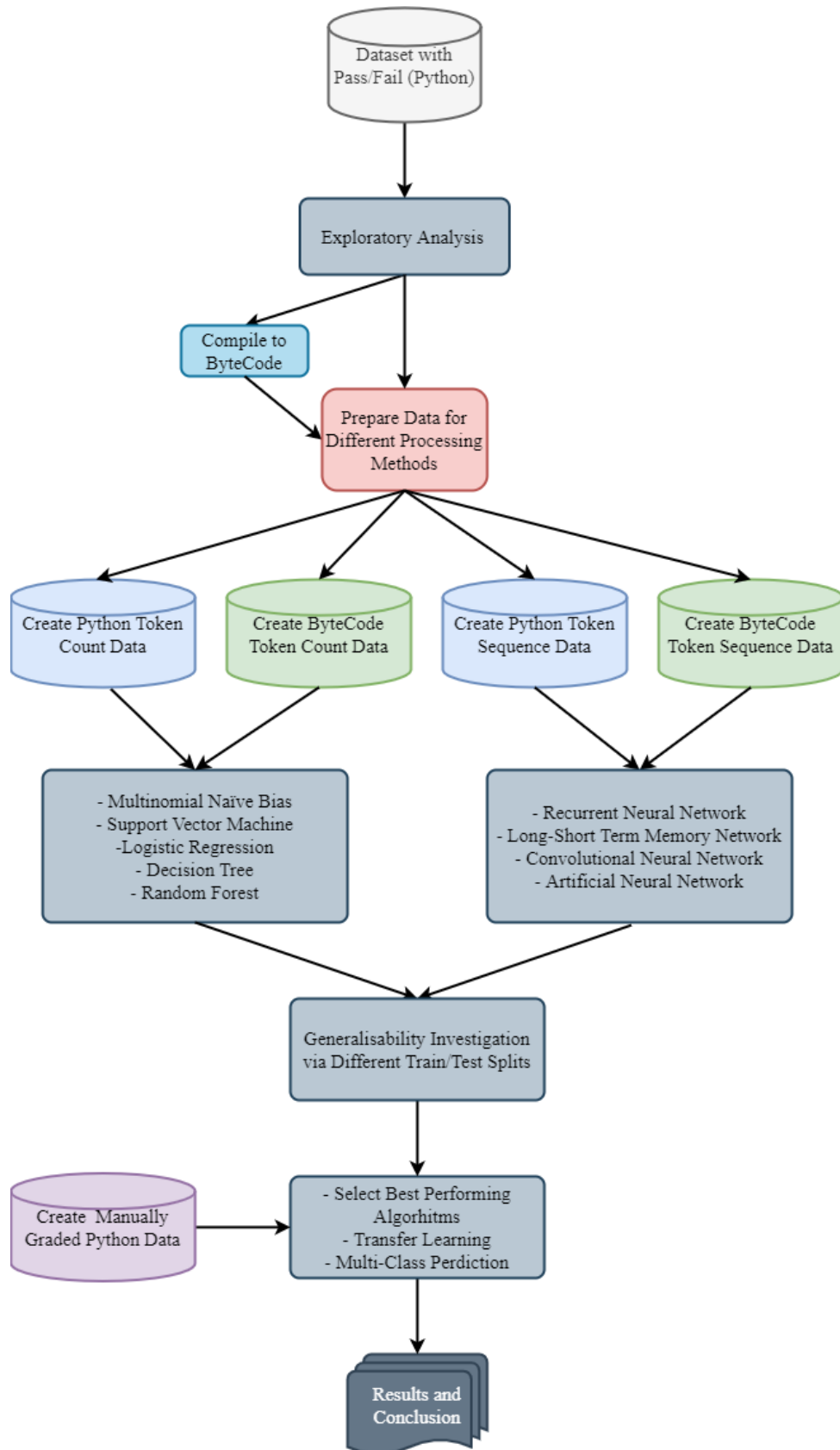


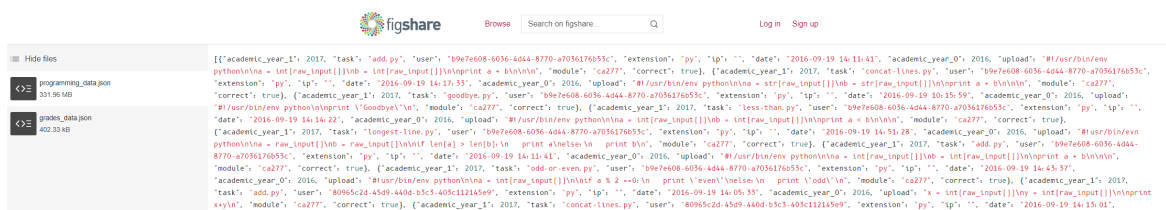
Fig. 3.1 High Level Process Flow

## 3.1 Data

The data had to be processed in several ways, producing sets of features that fit different types of models. We have used Machine Learning and Deep Learning methods, which can be divided into two main categories, models that work with independent features and models that work with sequences of items. Corresponding data set types were prepared. These are referred to as token count data and token sequence data throughout this document. In general, we refer to token count based data or models when each feature represents the number of occurrences of a word in a submission. We refer to token sequence-based data in cases where we tokenized the individual submissions to train the most suitable algorithms in a similar way as Natural Language Processing, to learn context and relations from the word sequences.

### 3.1.1 Raw Data and ByteCode

The data we have used was collected as part of research conducted by Azcona *et al.* (2019). Their research aimed at profiling students based on source code to identify learning development potential. The data contained half a million student source code submissions from more than 660 students across 3 years and 5 different courses, answering 665 different questions. The dataset was extracted from a unit testing platform where students were able to submit multiple solutions for the same question until the deadline, resubmitting their work until it was the best possible code they could create. This led to many instances of slightly different variations between the answers to the same question. All submissions were graded by an automatic unit testing system, removing any human bias from the results. An indicator of pass/fail status, for each submission, based on relevant unit tests, was included. The full dataset can be found at: Azcona and Smeaton (2019). A snapshot of the original JSON file can be found at Figure 3.2.



```
[{"academic_year_1": 2017, "task": "add.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:11:41", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = int([raw_input()])\nb = int([raw_input()])\nprint a + b\n\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "concat-lines.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:17:33", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = str([raw_input()])\nb = str([raw_input()])\nprint a + b\n\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "goodbye.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 10:15:59", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\nprint 'Goodbye!'\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "less-than.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:14:22", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = int([raw_input()])\nb = int([raw_input()])\nprint a < b\n\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "longest-line.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:51:28", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = raw_input()\nb = raw_input()\nif len(a) > len(b):\n    print a\nelse:\n    print b\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "add.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:11:41", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = int([raw_input()])\nb = int([raw_input()])\nprint a + b\n\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "odd-or-even.py", "user": "b9e7e608-6036-4d44-8770-a7036176053c", "extension": ".py", "ip": "", "date": "2016-09-19 14:43:37", "academic_year_0": 2016, "upload": "#!usr/bin/evn\npython\n\na = int([raw_input()])\nif a % 2 == 0:\n    print 'even'\nelse:\n    print 'odd'\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "add.py", "user": "80965c2d-45d9-440d-b3c3-403c112145e9", "extension": ".py", "ip": "", "date": "2016-09-19 14:05:35", "academic_year_0": 2016, "upload": "x = int([raw_input()])\ny = int([raw_input()])\nprint x+y\n", "module": "ca277", "correct": true}, {"academic_year_1": 2017, "task": "concat-lines.py", "user": "80965c2d-45d9-440d-b3c3-403c112145e9", "extension": ".py", "ip": "", "date": "2016-09-19 14:15:01",
```

Fig. 3.2 Original Dataset in Json

The Python submissions contained more than seventy thousand tokens when tokenized by the algorithms, therefore we decided to investigate if ByteCode would consist of fewer tokens while keeping the same prediction success levels. ByteCode is a lower-level programming language, which is an intermediate stage between Python and machine code

when the compiler processes the Python source code. The Python Virtual Machine translates the code to ByteCode, to be further processed by the operating system to machine instructions.

The submissions varied significantly in length. The average code length in ByteCode format was 4,000 characters, with a maximum of 21,000 and a minimum of 1, while the average length of Python code was 350 characters with a maximum of 35,000 and a minimum of 1. The data contain programming assignments in various languages; however, all other languages were removed apart from Python code. The data was anonymised by discarding all information except the source code, question name and the auto-grading results. Only code with a maximum length of 15,000 characters was used, to reduce the noise in the data created by outliers. The removed data accounted for 3.8 % of the whole dataset. Transformed code, consisting of Python ByteCode was also used. ByteCode consists of a set of about 200 instructions, of which the students were using 86. The pieces of code that could not be transformed or had empty content were discarded, leaving us with a dataset of 202,000 submissions. Of these 58% were labelled 'fail' and 42 % 'pass'.

### ByteCode definition

ByteCode sits below Python, as it is a lower-level representation of the Python code itself. The purpose of compiling Python to ByteCode at execution is to make Python platform-independent. ByteCode decompiles the Python code to a set of instructions, which is used by the Python Virtual Machine (PVM). Machine code is specific to the operating system and hardware the PVM is run on and will be compiled to binary code as a final step. Listing 3.1 shows a Python code, which is then compiled into ByteCode on Listing 3.2.

```
1 n = int(raw_input())
2 if n % 2 == 0:
3     print "even"
4 else:
5     print "odd"
```

Listing 3.1 Basic Python Code

```
1 4 0 LOAD_GLOBAL 0 (int)
2 2 LOAD_GLOBAL 1 (raw_input)
3 4 CALL_FUNCTION 0
4 6 CALL_FUNCTION 1
5 8 STORE_FAST 0 (n)
6
7 5 10 LOAD_FAST 0 (n)
8 12 LOAD_CONST 1 -2
```

```
9 14 BINARY_MODULO
10 16 LOAD_CONST 2 0
11 18 COMPARE_OP 2 (==)
12 20 POP_JUMP_IF_FALSE 28
13
14 6 22 LOAD_GLOBAL 2 (print)
15 24 POP_TOP
16
17 7 26 JUMP_FORWARD 4 (to 32)
18
19 9 >> 28 LOAD_GLOBAL 2 (print)
20 30 POP_TOP
21
22 10 >> 32 LOAD_CONST 0 (None)
23 34 RETURN_VALUE
```

Listing 3.2 Python Code compiled to ByteCode

### Purpose of using ByteCode

We have chosen to use ByteCode along with Python source code to investigate if the cleaner and more restricted version of the source code can lead to better results in predicting the outcome of the student source code submissions. As Python contains different variable names, different personal notes and logic, it can lead to increased noise in the data, and can greatly increase the requirements needed to process this data.

### 3.1.2 Derived Datasets

Eight sets of features were derived from the raw data i.e. the files containing code. Here we introduce the sets and describe how these were processed.

#### 1. Line Count Data Set

- (a) A set with a single attribute, the Python code length (line by line)

The single attribute set contains only the length of each code compiled to ByteCode. This was used to check if the length of the student code by itself would have an indication of the pass/fail result. A sample can be found in Figure 3.3:

|   | A     | B |
|---|-------|---|
| 1 | 11098 | 1 |
| 2 | 9614  | 1 |
| 3 | 6918  | 1 |
| 4 | 4694  | 1 |
| 5 | 5335  | 1 |

Fig. 3.3 Example of set ByteCode Length Set

## 2. Token Count Sets

### (a) ByteCode Token Count Set

ByteCode was investigated to reduce resource requirements and to explore if a more structured programming language can predict the same success. ByteCode token count set was derived from the full Python set, by iterating through the `.csv` file and creating a `.py` from each submission. These Python source files were then run through a code, creating a ByteCode version of the files. Files that did not compile or had 0 bytes were discarded, reducing the set from approx. 500 thousand samples to 200 thousand. These ByteCode files were then run through another python code to create a set where the number of times the ByteCode keywords appeared in the code was counted. For the set we used all available ByteCode keywords, however, we discovered that only 86 are used by the whole dataset (86 keywords appeared at least once), therefore these were the features for this set. A sample can be found in Figure 3.4

|   | A       | B       | C         | D        | E       | F           |
|---|---------|---------|-----------|----------|---------|-------------|
| 1 | POP_TOP | ROT_TWO | ROT_THREE | ROT_FOUR | DUP_TOP | DUP_TOP_TWO |
| 2 | 17      | 0       | 0         | 0        | 0       | 0           |
| 3 | 13      | 0       | 0         | 0        | 0       | 0           |
| 4 | 14      | 0       | 0         | 0        | 0       | 0           |
| 5 | 8       | 0       | 0         | 0        | 0       | 0           |

Fig. 3.4 Example of ByteCode Token Count Set

### (b) Python Tokenized Token Count Set

In this set, Python code was processed in its existing format by the Python built-in `tokenizer` command and the number of token type occurrences was counted per submission. The set contains the Python token types used in the code, the task number and the outcome of the unit testing. This set was generated from all available submissions, approximately 500,000. A sample can be found in Figure 3.5.

|   | A    | B      | C      | D       | E      | F      |
|---|------|--------|--------|---------|--------|--------|
| 1 | Name | Number | String | NewLine | Intent | Dedent |
| 2 | 28   | 3      | 1      | 11      | 2      | 2      |
| 3 | 128  | 12     | 1      | 26      | 10     | 10     |
| 4 | 7    | 5      | 0      | 4       | 1      | 1      |
| 5 | 15   | 4      | 1      | 6       | 2      | 2      |

Fig. 3.5 Example of Python Token Count Set

## (c) Python Keyword Token Count Set

In the reserved words set, we were interested to observe the number of occurrences of the following Python Keywords per submission: *False, def, if, raise, None, del, import, return, True, elif, in, try, and, else, is, while, as, except, lambda, with, assert, finally, nonlocal, yield, break, for, not, class, from, or, continue, global, pass*. Keywords with 0 occurrences were removed. The two additional features were the task number and the result of the unit test. This set was generated using all available submissions, approximately 500,000. A sample can be found in Figure 3.6

|   | A     | B   | C  | D     | E    | F   |
|---|-------|-----|----|-------|------|-----|
| 1 | FALSE | def | if | raise | None | del |
| 2 | 1     | 6   | 4  | 0     | 0    | 0   |
| 3 | 0     | 0   | 1  | 0     | 0    | 0   |
| 4 | 0     | 0   | 0  | 0     | 0    | 0   |
| 5 | 5     | 12  | 7  | 0     | 0    | 0   |

Fig. 3.6 Example of Python Keyword Set

## (d) Python Token and Keyword Count Set

This set combines the Python token count 3.5 and the Python Keyword 3.6 set however as some Python submissions were not tokenized by the internal tokenizer, a new script was used to first count the keywords, then tokenize the source code. Keywords and tokens with 0 occurrences were removed. Task number and final results were the two additional features here as well. This set was generated using all available submissions, approximately 500,000. A sample can be found in Figure 3.7

|   | P     | Q  | R      | S      | T    | U      | V       |
|---|-------|----|--------|--------|------|--------|---------|
| 1 | while | as | except | lambda | with | assert | finally |
| 2 | 0     | 0  | 0      | 0      | 0    | 0      | 0       |
| 3 | 1     | 0  | 0      | 0      | 0    | 0      | 0       |
| 4 | 1     | 0  | 0      | 0      | 0    | 0      | 0       |
| 5 | 0     | 0  | 0      | 0      | 0    | 0      | 0       |

Fig. 3.7 Example of Python Keyword and Token Count Set



### 3. Token Sequence Sets

#### (a) ByteCode Token Sequence Set

For the ByteCode Token Sequence set we left ByteCode in its instruction-based format, which was later used along with the *keras.tokenize* operation to create a tokenized representation of the ByteCode source code. For this set, every comment was removed, leaving only the instructions and the numbers there to preserve the sequential nature of the data. Numbers were later discarded as part of the tokenization process. The three features were the task number, which is related to the question, the code as a sequence of tokens and the result of the unit testing. The set used all the available data, and every submission which was compiled into ByteCode, approximately 200,000 rows. A sample can be found in Figure 3.8

|   | A    | B                | C      |
|---|------|------------------|--------|
| 1 | task | code             | result |
|   |      | 1 0 LOAD_CONST 0 |        |
|   |      | 2 LOAD_CONST 1   |        |
|   |      | 4 IMPORT_NAME 0  |        |
|   |      | 6 IMPORT_FROM 1  |        |
|   |      | 8 STORE_NAME 1   |        |
|   |      | 10 POP_TOP       |        |
| 2 | 492  | 1 0 LOAD_CONST 0 | 0      |
|   |      | 2 LOAD_CONST 1   |        |
|   |      | 4 IMPORT_NAME 0  |        |
|   |      | 6 STORE_NAME 0   |        |
|   |      | 4 8 BUILD_MAP 0  |        |
| 3 | 407  | 10 STORE_NAME 1  | 1      |

Fig. 3.8 Example of ByteCode Token Sequence Set Before Tokenization

#### (b) Python Token Sequence Set

The Python Token Sequence set was built by using Python code in its existing format, only the comments were removed. This set was further processed by the algorithm using *keras.tokenize*. This set used all available submissions, approximately 500,000. A sample can be found in Figure 3.9

|   | A              | B  | C      |
|---|----------------|--|--------|
| 1 | task           | code   | result |
| 2 | add.py         | a = int(raw_input())<br>b = int(raw_input()) | TRUE   |
| 3 | concatlines.py | a = str(raw_input())<br>b = str(raw_input()) | TRUE   |
| 4 | goodbye.py     | print "Goodbye"                              | TRUE   |
| 5 | lessthan.py    | a = int(raw_input())<br>b = int(raw_input()) | TRUE   |

Fig. 3.9 Example of Python Token Sequence Before Tokenization

(c) Manually Graded Python Token Sequence Set

The Python Graded Token Sequence set was built by using Python code in its existing format, only the comments were removed. This set used a manually graded section of the original data, containing 2 different questions and almost 600 submissions. This set was further processed by the algorithm using *keras.tokenize*. A sample can be found in Figure 3.10

|   | A                 | B  | O                                |
|---|-------------------|--|----------------------------------|
| 1 | Question name     | Code   | Grade out of 7 (manually graded) |
| 3 | countnumbersagain | a = []<br>x = input<br><br>while x != 0:<br>a.append(x)<br>x = input()<br><br>counter = 0<br>for i in a:                 | 2                                |
| 4 | countnumbersagain | a = []<br>x = input<br><br>while x != 0:<br>a.append(x)<br>x = input()<br><br>counter = 0<br>for i in a:<br>if i%2 == 0: | 2                                |

Fig. 3.10 Example of Python Graded Token Sequence Before Tokenization

### 3.1.3 Preliminary Data Analysis

An initial data analysis was conducted on the full dataset (section 3.1.1) to investigate n-grams in the code. We were interested in establishing the importance of word sequences. The number of times the same token appears across the whole dataset was counted and the same was done for bigrams, trigrams, and quadrigrams. The below figures (Figures 3.11 to 3.14 pertaining to Python, Figures 3.15 to 3.18 pertaining to ByteCode) show how the most used words and word sequences are distributed between pass and fail submissions, labelled TRUE and FALSE, broken down by 1, 2, 3 and 4 words. The figures contain incidence per code submission of different word sequences and contain only the top 15 sequences as the numbers of incidences had below 0.02 occurrences after the 15th sequence.

This analysis was conducted separately on the Failed and Passed submissions and these were compared in terms of n-gram incidence. During the analysis, we realised that some words and word sequences have strikingly different incidences of passing and failing code. For example, Figure 3.11 and Figure 3.12 show that *self* and *def* appear more in Python code that Passes the unit test. This was found to be the case in both Python and ByteCode and gave the first indication that there is some relationship between code correctness on the one hand and word incidence and word order on the other, which is promising with regard to the possibility of predicting correctness from content.

This exploration gave an insight into the importance of words and the sequence in which they appear. Individual words showed differences in incidence between pass and fail, indicating that token count could be a valid approach.

An interesting observation that can be made from these data is that passing (i.e. good) programs contain more function-related keywords *load\_const*, *make\_function* in ByteCode and *def*, *self* Python than code that fails.

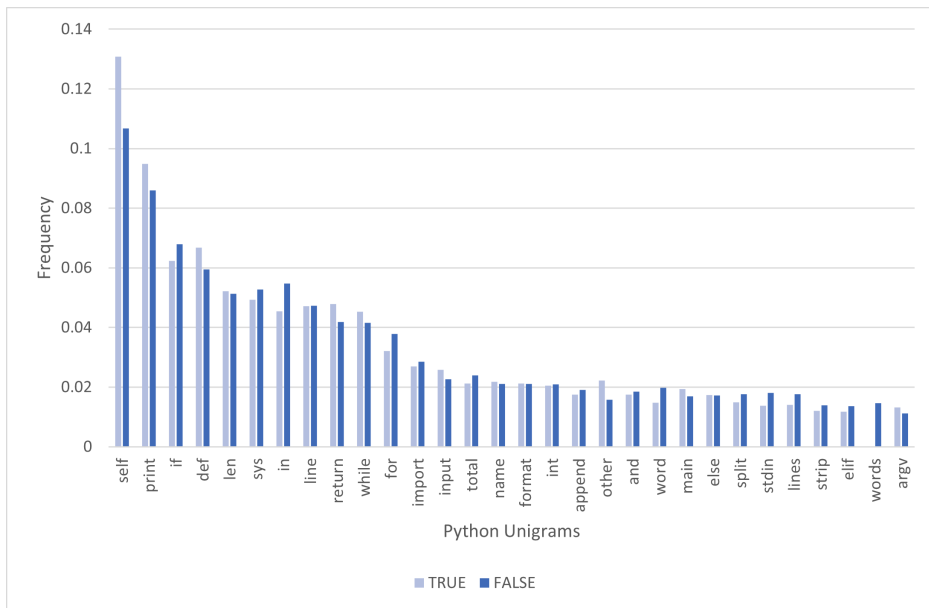


Fig. 3.11 Number of Unigrams per Code Submission in Python code

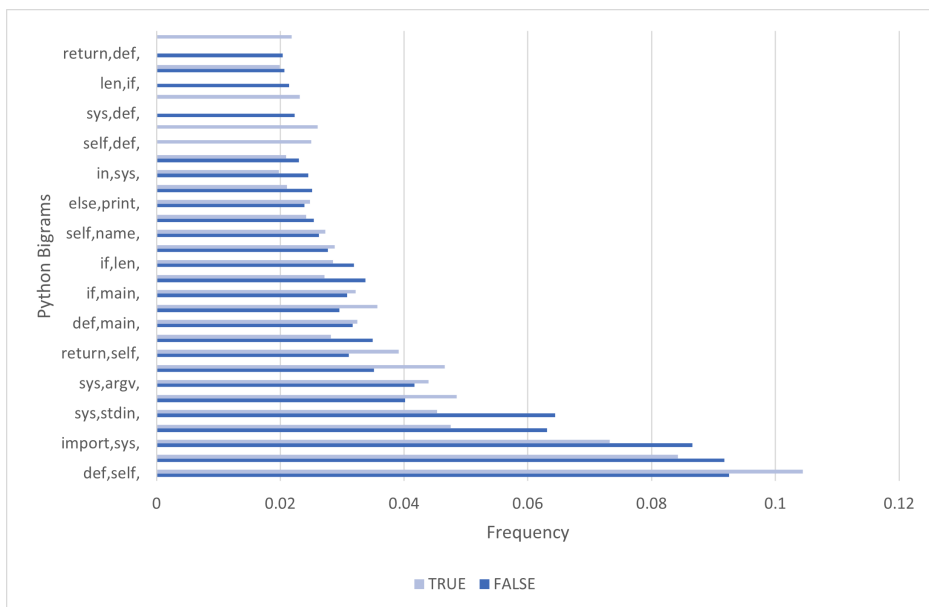


Fig. 3.12 Number of Bigrams per Code Submission in Python code

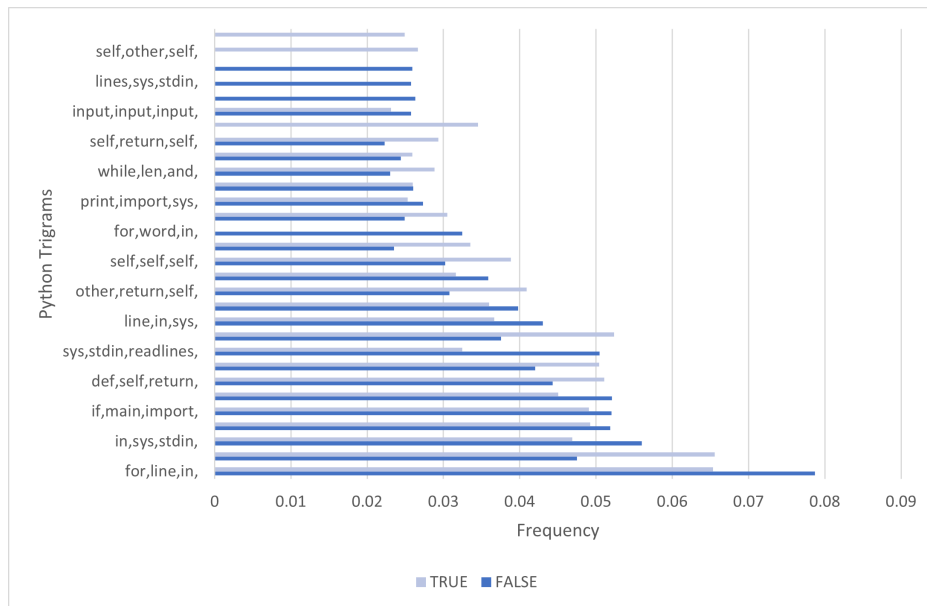


Fig. 3.13 Number of Trigrams per Code Submission in Python code

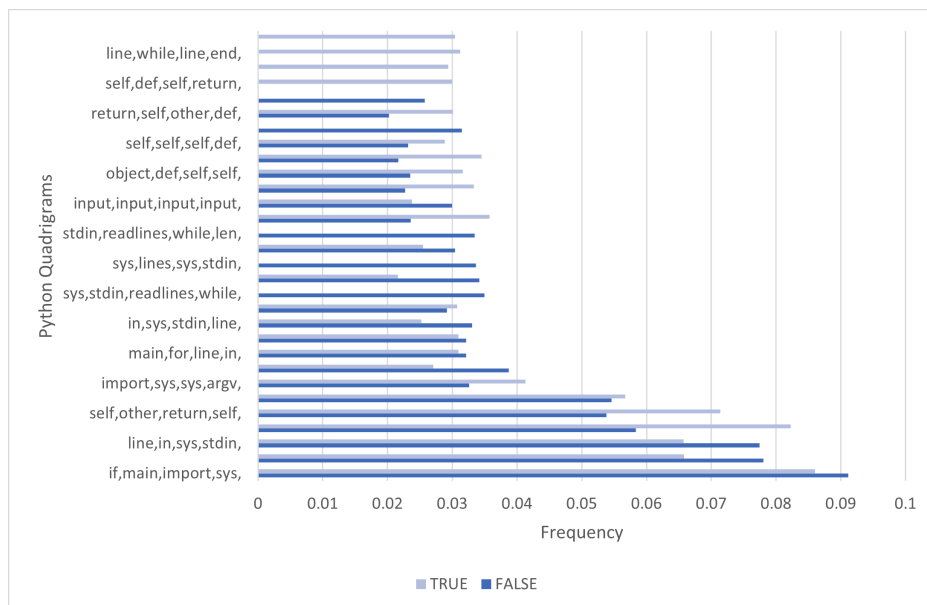


Fig. 3.14 Number of Quadrigrams per Code Submission in Python code

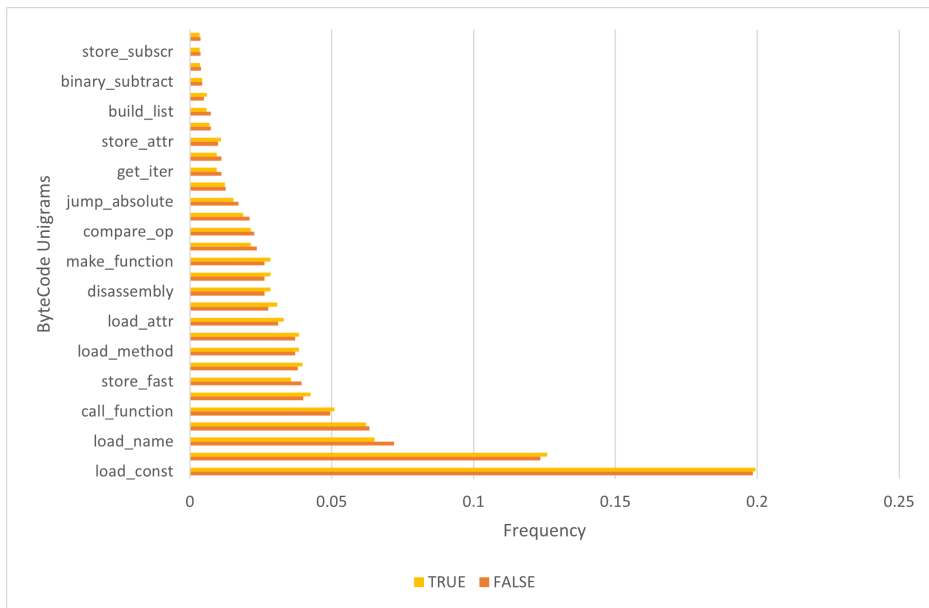


Fig. 3.15 Number of Unigrams per Code Submission in ByteCode code

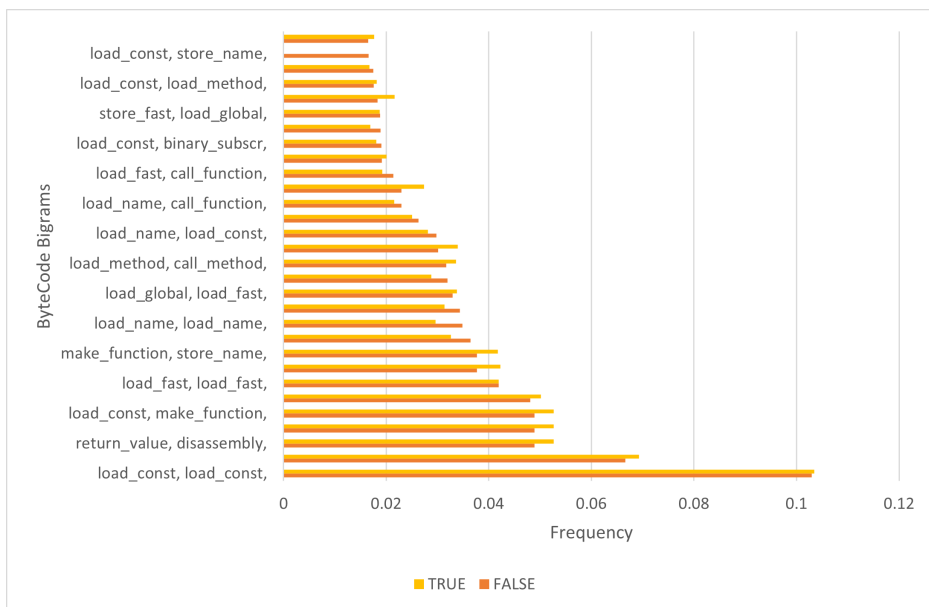


Fig. 3.16 Number of Bigrams per Code Submission in ByteCode code

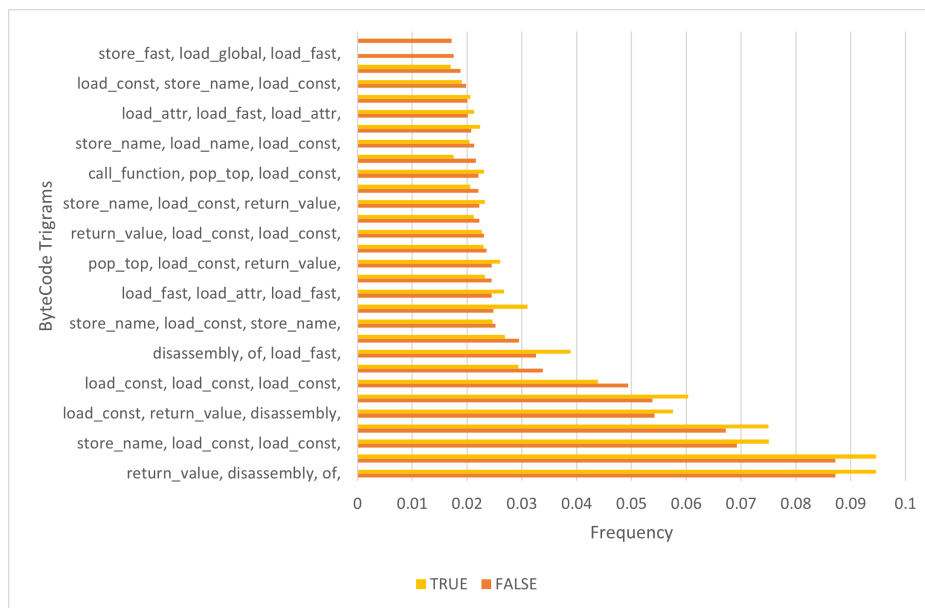


Fig. 3.17 Number of Trigrams per Code Submission in ByteCode code

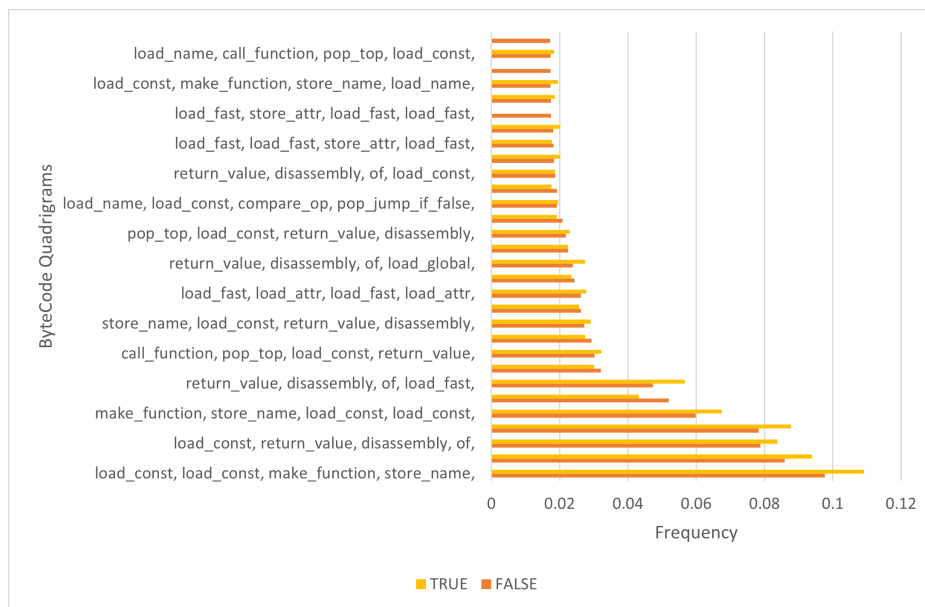


Fig. 3.18 Number of Quadrigrams per Code Submission in ByteCode code

## 3.2 Models

In this section the models used are discussed in detail grouped by data set type. Each model is presented individually in the relevant subsection.

### 3.2.1 Models for Line Count Data

Artificial Neural Network - The first step in our deep learning research was a simple Artificial Neural Network. To exclude the possibility that code length by itself is a determinant of pass/fail outcome, we performed basic training with ANN using only Code Length as a feature. This has not resulted in any reliable results.

### 3.2.2 Models for Token Count Data

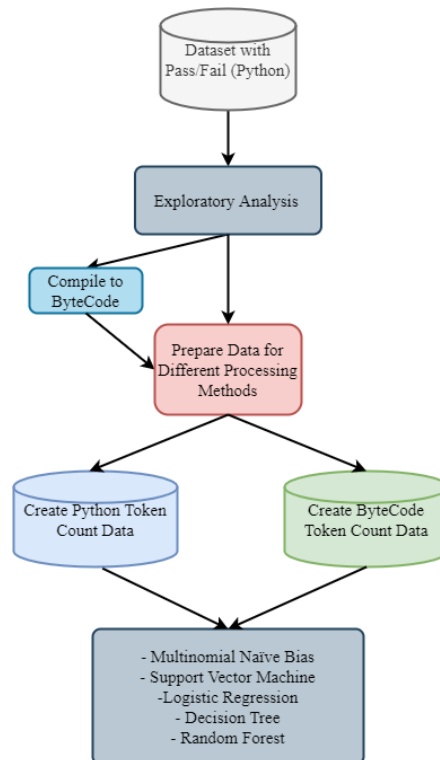


Fig. 3.19 Token Count Process Flow

As Figure 3.19 shows, first we used the Python Pass/Fail data and created different sets with token count information as presented in Subsection 2. The base dataset was also compiled to ByteCode, and those which were able to compile went through another set of



pre-processing to create the token count set for ByteCode. The following model types were fitted with token count data.

- Multinomial Naïve Bayes - Multinomial Naïve Bayes (MNB) is the simplest model and the fastest to fit.
- Support Vector Machine - Support Vector Machine (SVM) was used in linear form.
- Logistic Regression - Logistic Regression (LR) was used in a linear form.
- Decision Tree - Decision Trees (DT) was used to model non-linear dependencies.
- Random Forest - Random Forest (RF) was used to model non-linear dependencies.

### 3.2.3 Models for Token Sequence Data

Deep Learning is the most complex and resource-intensive form of machine learning. It is widely used in Natural Language Processing where context and sequence are as important as words. This leads us to hypothesise that deep learning algorithms would work well with code, helping to solve the automated grading process.

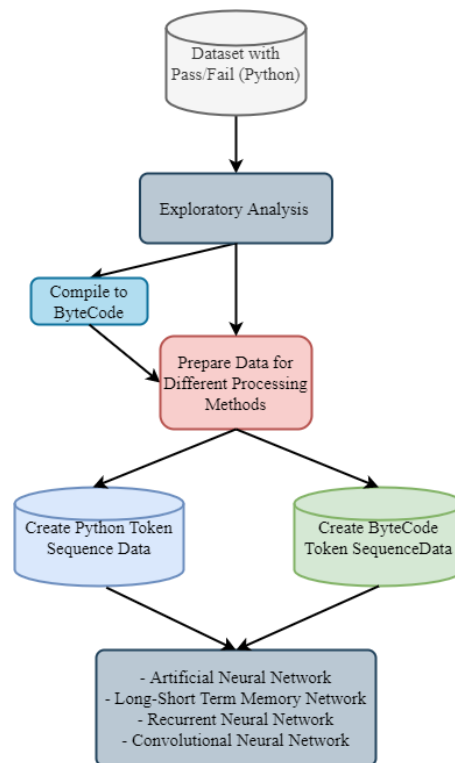


Fig. 3.20 Token Sequence Process Flow

Figure 3.20 shows that the datasets were created from the main Python set, and processed to create Token Sequence-based output, which was used as the input for the Deep Learning algorithms. The following model types were trained on token sequence data.

- **Artificial Neural Network - Artificial Neural Networks (ANN)** are one of the most simple Deep Learning algorithms, yet they generally achieve better consistency and show more generalizable features than the aforementioned Machine Learning Models. ANN was the only algorithm that was suitable to be tested on both Token Count and Token Sequence Datasets.
- **Long Short-Term Memory Network - Long Short-Term Memory Networks (LSTM)** are different than Artificial Neural Networks, as LSTMs preserve the sequence of the words and due to the use of the forget gate, they can learn the context in a more effective way than ANNs. LSTM's are widely used in Natural Language Processing and implemented for NLP purposes by a vast amount of peer-reviewed papers.
- **Recurrent Neural Network - Recurrent Neural Networks (RNN)** can store long-term information, however, they lack the ability to "forget", therefore the information is stored in the long term which in some cases can have an adverse effect on the outcome. These basic RNN models are very effective in Natural Language Processing and speech recognition.
- **Convolutional Neural Network (CNN) - Convolutional Neural Networks (CNN)** are mainly used in image processing, however, due to their pattern recognition capabilities, these are also excellent in NLP classification tasks. CNNs were initially developed to recognise patterns within pixels in a 2-dimensional space. They create their own feature map by scanning through the picture and learning to recognise shapes and forms. In NLP sense, they are capable of recognising connections between words (or even sentences). The only drawback of a CNN's in our case, where we are trying to determine if a student code is correct, is that CNNs are capable of learning from the structure of the data. This means that if a student solves a problem with an "If" and a "For" loop, and the second student solves it with "For" coming first, then CNN will consider them highly different. To overcome this, we have used a GlobalMaxPooling1D layer, to understand global patterns, instead of local ones.

### 3.2.4 Models for Multi-Class Label Data

For our initial exploration explained previously we used binary prediction as the available dataset contained only true/false unit test results. However, our goal was to research if we can use an automated way on finely graded submissions and predict those correctly. Therefore we manually graded almost 600 submissions across 2 different questions to provide a deeper understanding of the minimal differences between different marks. The binary prediction research gave us a clear guideline in which algorithms perform the best in determining the correct results of the submissions, therefore, building on that knowledge, we have used CNNs and LSTMs on the manually graded datasets.

Given the same domain and dataset, we were interested in investigating if Transfer Learning as explored by Pan and Yang (2009) would be applicable in grading the submissions. We have used the best-performing LSTM and CNN algorithms and froze the first layers, where the training has taken place on the full dataset, including all the questions. To refine the solution, we added an extra layer which was directly trained on the manually graded dataset. We have created 7 categories based on a custom-made grading matrix and the percentage of the grade awarded for each submission, which was then trained using the knowledge already present in the frozen layers.

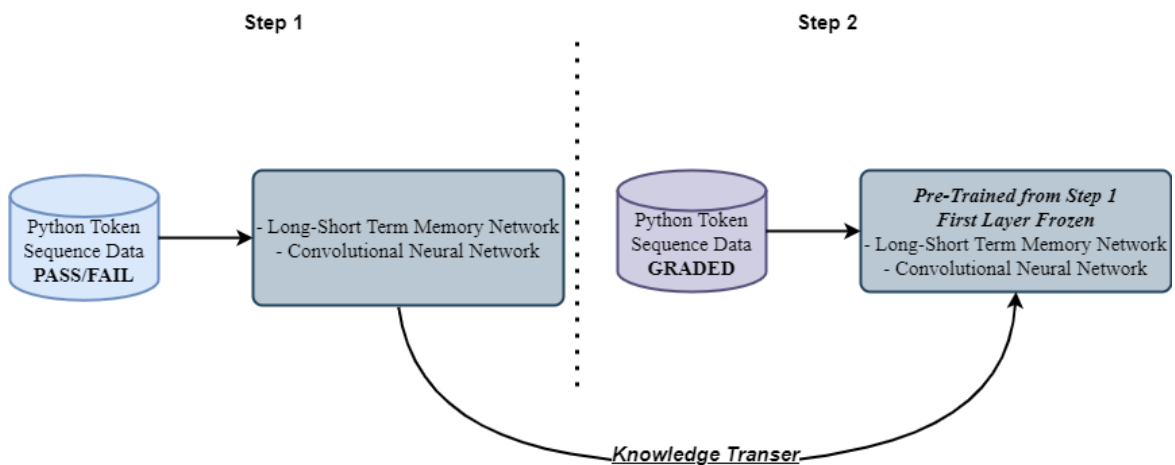


Fig. 3.21 Transfer Learning Process Flow

Figure 3.21 summarises the high-level process flow of Transfer Learning, where the algorithms trained on the Pass/Failed set retain the weights and "knowledge" learnt and were refined using the Graded dataset.

### 3.2.5 Model Type and Data Set Summary

As there are several models using different datasets, Table 3.1 summarises the relationship between the models and datasets discussed in Section 3.1.2 and Section 3.2.

Table 3.1 Algorithm and Dataset Relation Matrix

| Data Set Type              | Models               |
|----------------------------|----------------------|
| Line Count Data            | ANN                  |
| Token Count Data           | MNB, SVM, LR, DT, RF |
| Token Sequence Data        | ANN, LSTM, RNN, CNN  |
| Graded Token Sequence Data | LSTM, CNN            |

## 3.3 Generalisation Investigation

The word generalisation is used here to express how a model handles decreasing levels of similarity between training and validation data. Three levels of similarity were investigated: (1) For any question answered by a piece of validation code, there are pieces of training code that answer the same question (2) For any question answered by a piece of validation code, there are no pieces of training code that answer the same question but there are pieces of training code that answer similar questions (3) For any question answered by a piece of validation code, there are no pieces of training code that answer the same or similar questions.

We created controlled experiments to validate if the results are question-independent. We clearly defined which questions to train on, and which questions to hold back for testing. This will be discussed further in the results section for CNN (Section 4.4.4) and LSTM (Section 4.4.2).

### 3.3.1 Same Questions in Train and Test Subset

Most of the models in this research were trained on the full data set, with 80% of all the instances (code submissions) used for training and 20% for testing. The allocation of instances to these subsets was random. Given that in the set there are a large number (several hundred) of submissions per question, random allocation creates a statistical expectation that for each question some submissions will fall into the training set and some into the test set. Thus this approach to model fitting can be categorised under 'Same Questions in Train and Test Subset'.

However, models were also trained on individual questions, with all the submissions answering a single question involved in training and testing. For this to mature into a real-life

solution, we had to establish the minimum required samples for the training set. We ran our CNN algorithm through all questions one by one, with 400, 900 and 2000 examples each and explored the minimum amount of data required to train a model. The dataset didn't have the same amount of submissions for each question and there were significantly fewer with 900 and 2000 examples.

### **3.3.2 Similar Questions in Train and Test Subset**

During the second experiment, the questions were manually grouped into similar categories. The categories were related, solving very similar problems. As an example, question source codes such as: *count numbers*, *count items*, *count up*, *count even*, *count up to odd* were used as training set, and *count down*, *count odd* question source codes were used as validation set.

### **3.3.3 Different Questions in Train and Test Subset**

During the third experiment to explore if any of the models are further generalizable, we have created models using different questions in the train and the validation set. The questions representing the validation data were completely unseen by the algorithm.

# Chapter 4

## Results

In this section we discuss the results of our research. We have trained over 3000 models in Machine and Deep Learning on 3 different platforms. The results in this section represent the best-performing model setups with the best hyperparametrisation we were able to achieve. These tables contain the average of a 5-fold cross-validation across 3 different runs using the same setup. Each section explains the individual details and introduces the dataset used. To be able to compare the results objectively, we aimed to keep the parameters as close to each other as possible, using the most basic setup of the models. We fine-tuned only the best-performing models to gauge their potential. We believe that these results can be further refined with more robust and in-depth fine-tuning of the best-performing model, however, this is out of the scope of our project. Appendix C contains the heatmaps related to the tables.

### 4.1 Feature Selection

The feature selection process is crucial for every Machine Learning model, as the quality of data determines several factors of the models, such as training time, final results and complexity. Selecting the most relevant features is very important, to make sure that the underlying data is not just representative of the question the model is aiming to address, but also well-balanced between pass and fail. Imbalanced data can lead the algorithm to learn features that are only pertaining to one class. A good example of this is the number of passes or fails in the dataset in the binary prediction models. In case one is significantly more than the other then the algorithm will only learn how to predict that class.

Our feature selection was done in different phases, depending on the model and set we used. In fact, some of the sets were already feature-selected sets at the data derivation phase,

such as the sets used in Section 3.2.2. These sets were only containing the features that were aimed to represent the underlying data and are discussed in detail in the relevant section explaining the data set creation process.

Feature selection was also handled by the algorithms themselves, using one-hot encoding and embedding. These will be discussed later.

The Python Token Keyword and Tokenized Token Count data (dataset can be found in subsection 2d) on Figure 4.1 shows that there are 6 tokenized counts which have an importance of more than 0.06 as reported by the Decision Tree feature importance. These were *OP*, *NAME*, *NUMBER*, *NL*, *NEWLINE*, *STRING*.

*OP* is a generic token type for all operators, delimiters, and the ellipsis literal. This does not include characters and operators that are not recognised by the parser; the *NAME* token type is used for any Python identifier, as well as every keyword; the *NUMBER* token type is used for any numeric literal, including (decimal) integer literals, binary, octal, and hexadecimal integer literals, floating point numbers (including scientific notation), and imaginary number literals; the *NEWLINE* token type represents a newline character (`\n` or `\r\n`) that ends a logical line of Python code. Newlines that do not end a logical line of Python code use *NL*; the *STRING* token type matches any string literal, including single-quoted, double-quoted strings, triple- single and double-quoted strings (i.e., multi-line strings, or “docstrings”), raw, “unicode”, bytes, and f-strings (Python 3.6+). These Python Tokenizer details are extracted from an explanation by Meurer (2018).

The feature importance analysis for individual tokens in ByteCode (further information on ByteCode tokens by PythonSoftwareFoundation (2022)) on the Token Count data (dataset can be found in subsection 2a) using feature importance reported by the Decision Tree on Fig 4.2 shows that there are 8 words which have an importance of equal to or more than 0.04, with top 5 being *pop\_top*, *load\_name*, *load\_fast*, *call\_function*, *load\_const*. This information assisted in the hyperparameter tuning of the ML algorithms used.

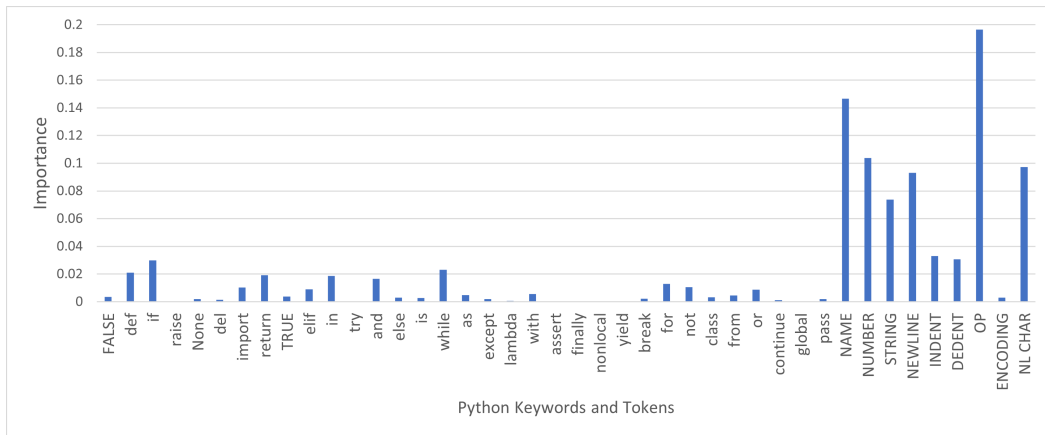


Fig. 4.1 Python Feature Importance - Decision Tree

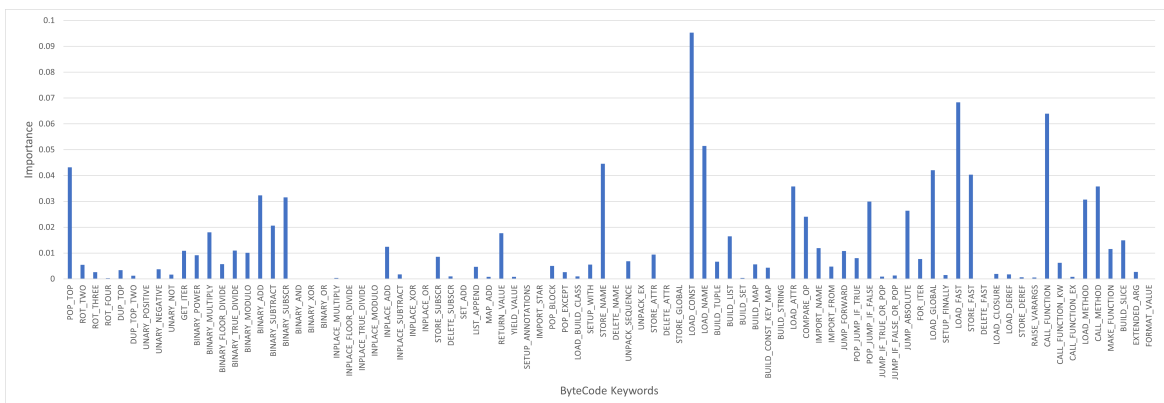


Fig. 4.2 ByteCode Feature Importance - Decision Tree



## 4.2 Model for Line Count Data

The investigation to prove or disprove that the length of the code by itself can be a predictor confirmed the hypothesis that the code length by itself cannot correctly predict the pass/fail outcome. A simple one Dense layer ANN with 200 Neurons wasn't able to predict the categories based on this one feature alone. As Table 4.1 shows, the F1 score of 0.45 confirms that the prediction was random.

Table 4.1 ANN - Code Length Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.51         | 0.84   | 0.63     |
| TRUE             | 0.52         | 0.17   | 0.26     |
| Accuracy         |              |        | 0.51     |
| Macro Average    | 0.51         | 0.51   | 0.45     |
| Weighted Average | 0.51         | 0.51   | 0.45     |

## 4.3 Models for Token Count Data

In this section results for models using Token Count Data are presented. All of these models were run through all of the different dataset in subsection 2 and the best-performing ones are presented in detail here. All token count models were balanced between pass and fail, containing the same amount of samples for each category.

### 4.3.1 MNB - Token Count Results

The *MultinomialNB* model was used from *sklearn.naive\_bayes* without any modifications across Python and ByteCode token count datasets, using the default setup for this experiment. We expected to get better results, as the probability of the same words occurring the same number of times might have been a good indicator of the correct code. However, this result would only indicate that the student used the correct keywords, but might not be used in the correct sequence, with the correct logic.

Although the Python (Table 4.2) model achieved the same F1 score as the ByteCode (Table 4.3), the precision and recall are better on Python, which can be explained by the fact that the Python count contained not only the keywords, but the operator counts, therefore more accurately mirroring the underlying data for probability-based prediction.

Table 4.2 MNB - Token Count Results - Python Token and Keyword Count Set

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.54         | 0.50   | 0.51     |
| TRUE             | 0.53         | 0.63   | 0.56     |
| Accuracy         |              |        | 0.53     |
| Macro Average    | 0.54         | 0.53   | 0.53     |
| Weighted Average | 0.54         | 0.53   | 0.53     |

Table 4.3 MNB - Token Count Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.53         | 0.54   | 0.53     |
| TRUE             | 0.53         | 0.49   | 0.52     |
| Accuracy         |              |        | 0.53     |
| Macro Average    | 0.53         | 0.53   | 0.53     |
| Weighted Average | 0.53         | 0.53   | 0.53     |

### 4.3.2 SVM - Token Count Results

Support Vector Machine research on Token Count data used *SVC* from *sklearn.svm* without any modification. This provided better F1 scores for ByteCode (Table 4.5) than Python (Table 4.4). This can be explained by the fact that ByteCode only contained compilable code with a significantly smaller set of two hundred thousand submissions, compared to the five hundred thousand in Python. It made it more suitable for the SVMs to create a suitable hyperplane between the two classes. It is also possible that due to the fact that the ByteCode data only used the compiled submissions, it already created a filter for better-quality code.

Table 4.4 SVM - Token Count Results - Python Token and Keyword Count Set

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.57         | 0.46   | 0.51     |
| TRUE             | 0.55         | 0.65   | 0.6      |
| Accuracy         |              |        | 0.56     |
| Macro Average    | 0.56         | 0.56   | 0.55     |
| Weighted Average | 0.56         | 0.56   | 0.55     |

Table 4.5 SVM - Token Count Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.63         | 0.57   | 0.60     |
| TRUE             | 0.61         | 0.66   | 0.63     |
| Accuracy         |              |        | 0.61     |
| Macro Average    | 0.62         | 0.62   | 0.61     |
| Weighted Average | 0.62         | 0.61   | 0.61     |

### 4.3.3 Logistic Regression - Token Count Results

A maximum iteration of 20,000 was used for Logistic Regression with the *LogisticRegression* model from *sklearn.linear\_model*, as this resulted in the best F1 score based on 20 test runs. Using the full Python and ByteCode dataset, ByteCode (Table 4.7) reached an F1 score of 0.57, which was slightly better than the 0.55 in Python (Table 4.6). Although the precision and recall didn't increase significantly, this F1 score is still close to .50, showing that the algorithm cannot consistently differentiate between the two classes. Due to the linearity of the Logistic Regression model, this was somewhat expected.

Table 4.6 Logistic Regression - Token Count Results - Python Token and Keyword Count Set

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.55         | 0.52   | 0.53     |
| TRUE             | 0.54         | 0.62   | 0.57     |
| Accuracy         |              |        | 0.55     |
| Macro Average    | 0.55         | 0.55   | 0.55     |
| Weighted Average | 0.55         | 0.55   | 0.55     |

Table 4.7 Logistic Regression - Token Count Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.57         | 0.55   | 0.56     |
| TRUE             | 0.56         | 0.62   | 0.58     |
| Accuracy         |              |        | 0.57     |
| Macro Average    | 0.57         | 0.57   | 0.57     |
| Weighted Average | 0.57         | 0.57   | 0.57     |

### 4.3.4 Decision Tree - Token Count Results

Decision Trees are excellent algorithms however very prone to overtraining. This can be clearly seen using the whole dataset for Python (Table 4.8) and ByteCode (Table 4.9) as they achieved very high F1 scores of 0.81 and 0.85 respectively. However looking into the results deeper, we were able to see that the Accuracy on the training split was between 91-92 % in both cases, while the test set achieved only 81-82 % which indicates that the set was almost perfectly learned. For this experiment, *DecisionTreeClassifier* algorithm was used without modification from *sklearn.tree*.

Table 4.8 Decision Tree - Token Count Results - Python Token and Keyword Count Set

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.83         | 0.79   | 0.81     |
| TRUE             | 0.78         | 0.87   | 0.82     |
| Accuracy         |              |        | 0.81     |
| Macro Average    | 0.82         | 0.81   | 0.81     |
| Weighted Average | 0.82         | 0.81   | 0.81     |

Table 4.9 Decision Tree - Token Count Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.86         | 0.83   | 0.85     |
| TRUE             | 0.82         | 0.89   | 0.85     |
| Accuracy         |              |        | 0.85     |
| Macro Average    | 0.85         | 0.85   | 0.85     |
| Weighted Average | 0.85         | 0.85   | 0.85     |

To investigate if any of the learning achieved by this model is based on general properties of code solving small problems, we have created a different data split and fed the algorithm different questions for training and testing, instead of using a general built-in split function, which might contain the same questions. The result in Table 4.10 clearly shows that in this case, the F1 score drops significantly to 0.53. Since this is close to random guessing, we can conclude that all the information learnt as evidenced by results in Table 4.8 is question-specific.

Table 4.10 Decision Tree - Keyword Count Results - Python - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.53                     | 0.52   | 0.52     |
| TRUE             | 0.53                     | 0.53   | 0.53     |
| Accuracy         |                          |        | 0.53     |
| Macro Average    | 0.53                     | 0.53   | 0.53     |
| Weighted Average | 0.53                     | 0.53   | 0.53     |

### 4.3.5 Random Forest - Token Count Results

As Random Forests build a set of decision trees using different samples of data, these are extremely versatile. *RandomForestClassifier* algorithm was used from *sklearn.ensemble* in its basic form. Using this approach on the full dataset also creates the same challenge as the Decision Tree, the issue of overtraining. Although as the results in Table 4.11 and 4.12 show, Random Forest is performing better with higher F1 scores and better Recall and Precision than the Decision Trees, these are still overtrained and only able to be used in prediction of the very same question or in the same set of questions.

Table 4.11 Random Forest - Keyword and Token Count Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.85         | 0.81   | 0.83     |
| TRUE             | 0.80         | 0.89   | 0.84     |
| Accuracy         |              |        | 0.83     |
| Macro Average    | 0.84         | 0.83   | 0.83     |
| Weighted Average | 0.84         | 0.83   | 0.83     |

Table 4.12 Random Forest - Token Count Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.88         | 0.85   | 0.86     |
| TRUE             | 0.84         | 0.91   | 0.87     |
| Accuracy         |              |        | 0.87     |
| Macro Average    | 0.87         | 0.87   | 0.87     |
| Weighted Average | 0.87         | 0.87   | 0.87     |

The Random Forest modelling was investigated in the same way as the Decision Tree, using completely different questions and answers for train/test split with a similar result, a big drop in F1 Score in Table 4.13. Since this is close to random guessing, we can conclude that all the information learnt as evidenced by results in Table 4.11 is question-specific.

Table 4.13 Random Forest - Keyword and Token Count Results - Python - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.54                     | 0.55   | 0.54     |
| TRUE             | 0.54                     | 0.48   | 0.52     |
| Accuracy         |                          |        | 0.54     |
| Macro Average    | 0.54                     | 0.54   | 0.54     |
| Weighted Average | 0.54                     | 0.54   | 0.54     |

## 4.4 Models for Token Sequence Data

This section presents the results from the models built upon the Token Sequence Dataset from subsection 3. The word sequences were tokenized using *Tokenizer* from *keras.preprocessing.text* and the embedding dimensions were kept as close as possible to each other.

### 4.4.1 ANN Results - Token Sequence Data

Once it was clear that the single Code Length in Section 4.2 is not the main factor in prediction based on the poor 0.24 F1 score, ANN was tested on token sequence data for ByteCode and Python. Due to the fact that ByteCode is already pre-filtered by the compiling algorithm, we have expected to see better results, which can be seen in the difference of F1 score of 0.72 in Python (Table 4.14) and 0.84 in ByteCode (Table 4.15).

Table 4.14 ANN - Token Sequence Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.77         | 0.63   | 0.69     |
| TRUE             | 0.59         | 0.81   | 0.74     |
| Accuracy         |              |        | 0.72     |
| Macro Average    | 0.73         | 0.72   | 0.72     |
| Weighted Average | 0.73         | 0.72   | 0.72     |

Table 4.15 ANN - Token Sequence Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.90         | 0.80   | 0.85     |
| TRUE             | 0.76         | 0.88   | 0.82     |
| Accuracy         |              |        | 0.83     |
| Macro Average    | 0.83         | 0.84   | 0.83     |
| Weighted Average | 0.84         | 0.83   | 0.84     |

#### 4.4.2 LSTM Results - Token Sequence Data

We had high expectations as LSTM's are capable of learning "context" using the Forget Gate. This algorithm is widely used in NLP settings and we have hypothesised that it will be useful to analyze programming submissions. As the results show, the algorithm did manage to adjust the weights to achieve a reliable prediction. A comfortable 0.70 F1 score on Python (Table 4.16) and 0.75 on ByteCode (Table 4.17) shows that the LSTM might be capable of correctly predicting the outcome. However the difference in recall on the Python dataset shows that similarly to the ANN, the algorithm was significantly better in predicting FALSE results. This difference was reduced on the ByteCode dataset, which already has the pre-filter of only compilable code being used. The LSTM algorithms had an *Embedding* layer, a *SpatialDropout1D* layer, and two *LSTM* layers before the final *Dense* layer for binary prediction.

Table 4.16 LSTM - Token Sequence Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.72         | 0.82   | 0.77     |
| TRUE             | 0.66         | 0.53   | 0.59     |
| Accuracy         |              |        | 0.71     |
| Macro Average    | 0.69         | 0.68   | 0.68     |
| Weighted Average | 0.70         | 0.71   | 0.70     |

Table 4.17 LSTM - Token Sequence Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.77         | 0.79   | 0.78     |
| TRUE             | 0.70         | 0.68   | 0.69     |
| Accuracy         |              |        | 0.75     |
| Macro Average    | 0.74         | 0.74   | 0.74     |
| Weighted Average | 0.75         | 0.75   | 0.75     |

To investigate if the above results are question-independent, we have used the same train/test split as in the Decision Tree and Random Forest, providing unseen questions to test the accuracy on. This led to a similar drop in F1 score both in Python (Table 4.18) and ByteCode (Table 4.19) with final F1 scores almost the same as the Random Forest (Table 4.13). It is also notable that the ByteCode dataset performed slightly better with an F1 score of 0.54 than the Python with an F1 of 0.53. The recall in both cases shows that the failed submissions were predicted better.

Table 4.18 LSTM - Token Sequence Results - Python - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.53                     | 0.73   | 0.61     |
| TRUE             | 0.56                     | 0.35   | 0.43     |
| Accuracy         |                          |        | 0.54     |
| Macro Average    | 0.55                     | 0.54   | 0.52     |
| Weighted Average | 0.55                     | 0.54   | 0.52     |

Table 4.19 LSTM - Token Sequence Results - ByteCode - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.53                     | 0.72   | 0.61     |
| TRUE             | 0.57                     | 0.37   | 0.45     |
| Accuracy         |                          |        | 0.54     |
| Macro Average    | 0.55                     | 0.54   | 0.53     |
| Weighted Average | 0.55                     | 0.54   | 0.53     |



### 4.4.3 RNN Results - Token Sequence Data

RNN's are very similar to LSTMs, without the ability to "forget" the information learned. This is represented in the results, as the weighted F1 score of 0.67 with a Recall of 0.48 on the failed submissions is not reliable enough to predict the class fully on the Python dataset (Table 4.20). The ByteCode dataset (Table 4.21) performed significantly better, with an F1 score of 0.74 and a more balanced recall on both classes. This result shows the overarching trend of Bytecode (due to the more restricted structure and the pre-filter of only compiled code) being more beneficial than processing Python source code as is. The RNN model had an *embedding* layer, a *SpatialDropout1D* layer, a *SimpleRNN* layer and two *Dense* layers.

Table 4.20 RNN - Token Sequence Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.70         | 0.81   | 0.75     |
| TRUE             | 0.63         | 0.48   | 0.54     |
| Accuracy         |              |        | 0.68     |
| Macro Average    | 0.67         | 0.65   | 0.65     |
| Weighted Average | 0.67         | 0.68   | 0.67     |

Table 4.21 RNN - Token Sequence Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.77         | 0.79   | 0.78     |
| TRUE             | 0.70         | 0.68   | 0.69     |
| Accuracy         |              |        | 0.74     |
| Macro Average    | 0.73         | 0.73   | 0.73     |
| Weighted Average | 0.74         | 0.74   | 0.74     |

#### 4.4.4 CNN Results - Token Sequence Data

CNNs are used widely in NLP as they can achieve very high accuracy in learning both global and local contexts, making them analyzing text an outstanding choice. This was very obvious after running the experiments on the full dataset, with the highest F1 score of 0.81 on the Python set (Table 4.22) which was the best deep learning results across all models. ByteCode (Table 4.23) performed worse with an F1 score of 0.76, which can be explained by how the source code is constructed.

Table 4.22 CNN - Token Sequence Results - Python

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.90         | 0.76   | 0.83     |
| TRUE             | 0.71         | 0.88   | 0.79     |
| Accuracy         |              |        | 0.81     |
| Macro Average    | 0.81         | 0.82   | 0.81     |
| Weighted Average | 0.83         | 0.81   | 0.81     |

Table 4.23 CNN - Token Sequence Results - ByteCode

|                  | Full Dataset |        |          |
|------------------|--------------|--------|----------|
|                  | Precision    | Recall | F1-Score |
| FALSE            | 0.79         | 0.80   | 0.79     |
| TRUE             | 0.72         | 0.71   | 0.71     |
| Accuracy         |              |        | 0.76     |
| Macro Average    | 0.75         | 0.75   | 0.75     |
| Weighted Average | 0.76         | 0.76   | 0.76     |

CNNs were the best-performing models on the full dataset in deep learning models. These were also investigated from a question-independent perspective, similar to the LSTM, Decision Tree and Random Forest models. The questions in the test set were completely unseen in this case. The significant drop was the same as in the previous algorithms, reaching only 0.50 F1 score on Python (Table 4.24) and 0.54 on ByteCode (Table 4.25).

Table 4.24 CNN - Token Sequence Results - Python - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.52                     | 0.76   | 0.62     |
| TRUE             | 0.55                     | 0.29   | 0.38     |
| Accuracy         |                          |        | 0.53     |
| Macro Average    | 0.53                     | 0.53   | 0.50     |
| Weighted Average | 0.53                     | 0.53   | 0.50     |

Table 4.25 CNN - Token Sequence Results - ByteCode - Question Independent

|                  | Question Independent Set |        |          |
|------------------|--------------------------|--------|----------|
|                  | Precision                | Recall | F1-Score |
| FALSE            | 0.54                     | 0.57   | 0.55     |
| TRUE             | 0.54                     | 0.51   | 0.52     |
| Accuracy         |                          |        | 0.54     |
| Macro Average    | 0.54                     | 0.54   | 0.54     |
| Weighted Average | 0.54                     | 0.54   | 0.54     |

## 4.5 Models for Multi-Class Label Data

Results presented so far were using only pass/fail labels. However, to progress toward answering the research question of whether it is possible to measure the level of correctness of code using machine learning, a fine-graded, multi-class prediction is required. For this purpose, the best-performing deep learning algorithms discussed in this section use the manually graded data presented in subsection 3.1.2 Token Sequence Sets (c).

### 4.5.1 LSTM Results - Token Sequence Data - Multi-Class Prediction

To be able to use the knowledge acquired previously, a Transfer learning method was investigated, where the first 2 LSTM layers trained on the full dataset were frozen and a manually graded smaller set was used to train the last layer on just one question.

This resulted in a 0.64 F1 score, however, the fact that we had such a small number of training examples and 7 different categories showed that there might be a potential in investigating further. The 7 categories were 0-35% as fail, going up in 10 % increments. As table 4.26 shows, there were 2 categories that were not predicted at all, while one category was predicted perfectly. This might have been caused by the categories being so close to

each other in grading, that the algorithm wasn't able to differentiate.

Table 4.26 LSTM - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset |        |          |
|------------------|----------------------------------|--------|----------|
|                  | Precision                        | Recall | F1-Score |
| 0                | 0.68                             | 0.97   | 0.80     |
| 1                | 0.81                             | 0.76   | 0.79     |
| 2                | 0.25                             | 0.17   | 0.20     |
| 3                | 0.56                             | 0.38   | 0.45     |
| 4                | 0.00                             | 0.00   | 0.00     |
| 5                | 0.00                             | 0.00   | 0.00     |
| 6                | 1.00                             | 1.00   | 1.00     |
| Accuracy         |                                  |        | 0.68     |
| Macro Average    | 0.47                             | 0.47   | 0.46     |
| Weighted Average | 0.61                             | 0.68   | 0.64     |

To resolve the issue, we have collapsed the categories into a smaller number of categories to test if that would lead to better predictions. In this case, the categories were 0-40%, 40-50%, 50-60%, 70-90% and 100% which resulted in more reliable predictions in terms of recall and precision on each category as Table 4.27 shows. However, the sample size was different for the individual categories, which might affect the outcome.

Table 4.27 LSTM - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset - Reduced Categories |        |          |
|------------------|---|--------|----------|
|                  | Precision   | Recall | F1-Score |
| 0                | 0.67  | 1.00   | 0.81     |
| 1                | 0.76  | 0.76   | 0.76     |
| 2                | 0.20  | 0.17   | 0.18     |
| 3                | 0.67  | 0.31   | 0.42     |
| 4                | 1.00  | 0.22   | 0.36     |
| Accuracy         |   |        | 0.67     |
| Macro Average    | 0.66  | 0.49   | 0.51     |
| Weighted Average | 0.69  | 0.67   | 0.63     |

To refine the results even further, we have made the dataset equal, 66 samples of each 5 categories. Although the weighted average F1 score was slightly less than in table 4.27, the

individual categories performed significantly better in Precision and Recall (Table 4.28) confirming that equal sample size is an improvement in correctly predicting partial correctness.

Table 4.28 LSTM - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset - Equal 66 samples |        |          |
|------------------|---|--------|----------|
|                  | Precision   | Recall | F1-Score |
| 0                | 0.70  | 1.00   | 0.82     |
| 1                | 0.75  | 0.50   | 0.60     |
| 2                | 0.57  | 0.44   | 0.50     |
| 3                | 0.50  | 0.43   | 0.46     |
| 4                | 0.50  | 0.75   | 0.60     |
| Accuracy         |   |        | 0.61     |
| Macro Average    | 0.60  | 0.62   | 0.60     |
| Weighted Average | 0.61  | 0.61   | 0.59     |

#### 4.5.2 CNN Results - Token Sequence Data - Multi-Class Prediction

Similarly to the LSTM approach, CNN Transfer Learning was investigated using the same approach (freezing the pre-trained layer which used the full data with binary prediction and adding an extra layer to train on the graded set). This has achieved better results than LSTM with a Weighted F1 Score of 0.71 on 7 classes (Table 4.29) however had the same problem by not predicting 2 classes at all.

Table 4.29 CNN - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset |        |          |
|------------------|----------------------------------|--------|----------|
|                  | Precision                        | Recall | F1-Score |
| 0                | 0.79                             | 1.00   | 0.89     |
| 1                | 0.79                             | 0.88   | 0.83     |
| 2                | 0.25                             | 0.17   | 0.20     |
| 3                | 0.70                             | 0.54   | 0.61     |
| 4                | 0.00                             | 0.00   | 0.00     |
| 5                | 0.00                             | 0.00   | 0.00     |
| 6                | 1.00                             | 1.00   | 1.00     |
| Accuracy         |                                  |        | 0.75     |
| Macro Average    | 0.50                             | 0.51   | 0.50     |
| Weighted Average | 0.68                             | 0.75   | 0.71     |

Similarly to LSTM, a reduced number of categories was used to check if this can result in a better prediction in terms of recall and precision. This also performed better than the LSTM with a higher weighted F1 score as presented in Table 4.30.

Table 4.30 CNN - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset - Reduced Categories |        |          |
|------------------|---|--------|----------|
|                  | Precision   | Recall | F1-Score |
| 0                | 0.78  | 1.00   | 0.87     |
| 1                | 0.75  | 0.88   | 0.81     |
| 2                | 0.33  | 0.17   | 0.22     |
| 3                | 0.70  | 0.54   | 0.61     |
| 4                | 0.67  | 0.22   | 0.33     |
| Accuracy         |   |        | 0.74     |
| Macro Average    | 0.64  | 0.56   | 0.57     |
| Weighted Average | 0.71  | 0.74   | 0.70     |

To investigate how the number of samples might affect the results, the same reduced set with 66 samples from each category was used on CNN Transfer Learning (Table 4.31) similar to the LSTM. This has resulted in similar, but better numbers than the LSTM and the categories were predicted precisely with a simple CNN algorithm, opening up the way to build on this promising result in the future.

Table 4.31 CNN - Token Sequence Results - Transfer Learning - Python

|                  | Full Dataset with Graded Dataset - Equal 66 samples |        |          |
|------------------|---|--------|----------|
|                  | Precision   | Recall | F1-Score |
| 0                | 0.78  | 1.00   | 0.88     |
| 1                | 1.00  | 0.50   | 0.67     |
| 2                | 1.00  | 0.44   | 0.62     |
| 3                | 0.33  | 0.43   | 0.38     |
| 4                | 0.50  | 1.00   | 0.67     |
| Accuracy         |   |        | 0.64     |
| Macro Average    | 0.72  | 0.67   | 0.64     |
| Weighted Average | 0.75  | 0.64   | 0.64     |

## 4.6 Result of Minimum Instance Requirement

To illustrate the results of the training algorithms, we have used different colours in Figure 4.3 and Figure 4.4, showing the related F1 score (Figure 4.3 is a partial example, zoomed in for the last 15 questions). The blue bars represent the F1 score by the question on the instances where we had at least 400 examples. The orange bars represent the F1 score of the questions with at least 900 examples, while the grey bars show the questions with at least 2000 examples. As one can observe, there are only just a few (highly complicated) questions which cannot use a model trained solely on that one question with only 400 instances available as the training set. This can be determined by the length of the bars, and those which are below 0.70 F1 scores are rare. This opens up the possibility of an online service, where lecturers can submit the minimum amount of their own questions and answers, and use their individually trained algorithms to predict the results of that question.

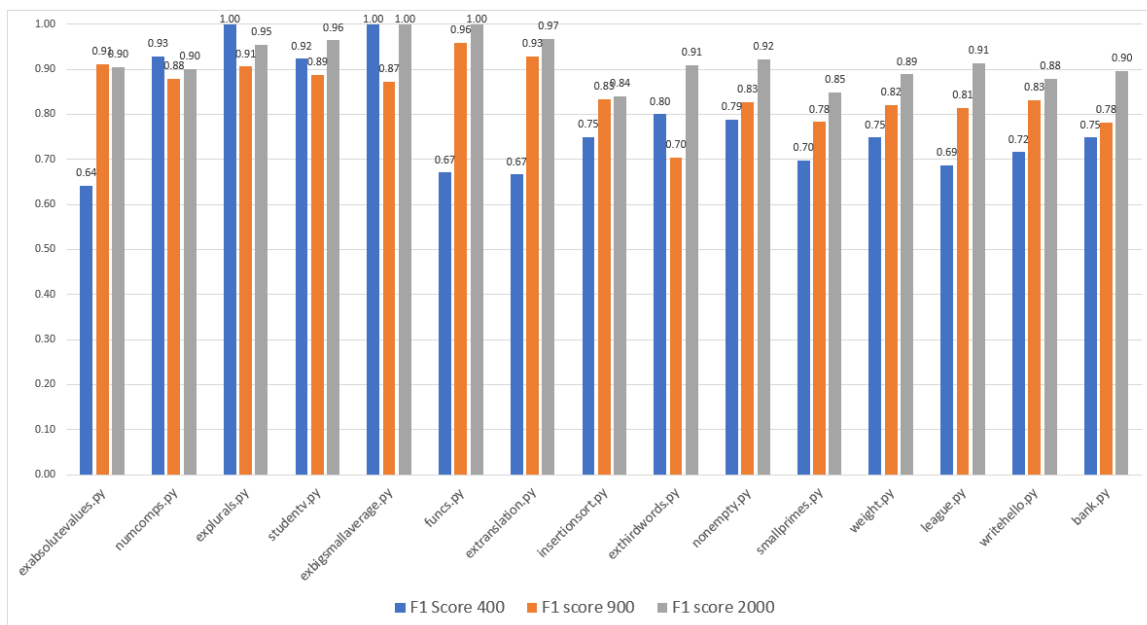


Fig. 4.3 Python Minimum Instances Partial Example

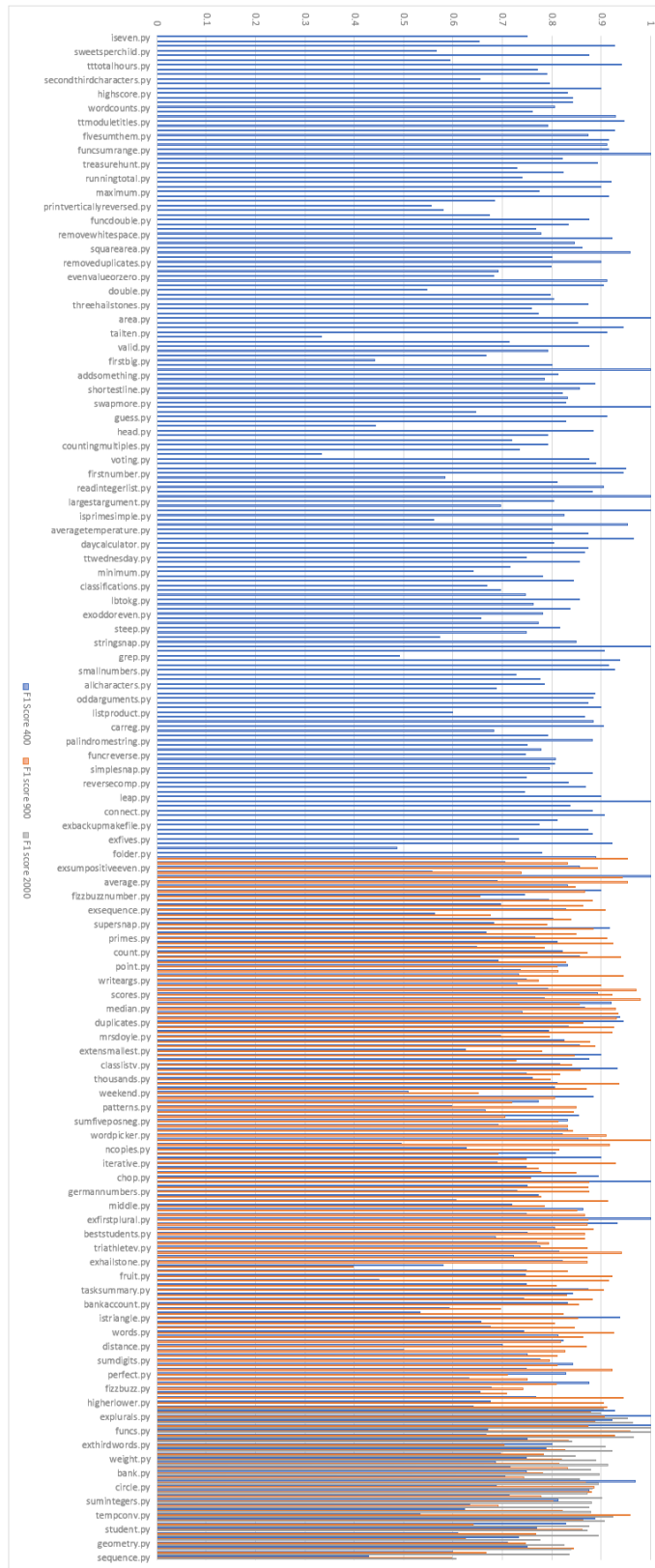


Fig. 4.4 Python Minimum Instances to F1 score



## 4.7 Result Comparison to Related Work

In this section we compare our results to those presented in previous works. We aim to provide a comparison of similar works and results, which excludes those which are completely different or non-comparable.

Our overall precision and recall on all of our algorithms were above the 44% reported in a deep learning work by Piech *et al.* (2015). Liu *et al.* (2019) used 10 programming questions to benchmark their system against, with an average accuracy of 92%. Their approach outperforms our best results with an accuracy of 87% on ByteCode - Random Forest (Table 4.12) and our best performing CNN with 81% accuracy (Table 4.22). It is worth noting that both of our results were looking at 663 different questions instead of 10. The closest comparable results would be matched against the Minimum Instance examination (Section 4.6), where we trained our CNN algorithm on every question individually, in which case we achieved 100% accuracy in several questions.

Most of the other related works investigated the problem in a different way, such as measuring the closeness to a correct answer as explained by Rai *et al.* (2019). Their Support Vector Machine with RBF Kernel achieved an 82% precision, which was very close to several of our CNNs (Tables 4.23 with 79 % precision on our False examples and to our Random Forest (Table 4.12) with 84% precision on the True examples.

# Chapter 5

## Conclusion

Manual grading of student submissions in the Python programming language is time-consuming and resource intensive. Instructors in general use similar questions in tests year after year. The change in the way the questions are worded does not mean a change in the required outcome, therefore automation is possible and highly sought after, as discussed in the Literature Review (section 2).

Within the broader context of aiming for successful automation of fine-grained assessment of code, this research was focused specifically on finding out whether it is possible to measure the level of correctness of code using machine learning and, if so, under what constraints. The research also aimed to understand if the difference between different grades could be predicted consistently, without knowing the question itself only by looking at the source code directly.

To answer these questions, 9 algorithms using 8 different sets of data derived from the same dataset were used. The main dataset was labelled by a unit testing framework, generating pass and fail results. This was further graded manually for further differentiation between levels of correctness.

Throughout the research, we were able to prove that code quality can be modelled using machine learning algorithms and data based on source code tokens; these investigations were mainly conducted on binary-labelled data but some experiments show initial promising results for multi-class labelled data.

The most successful model for the token count set is Random Forest. This success can be attributed to the nonlinearity of the model and to the simple questions processed, which

shows that counting the tokens can be enough to predict a grade at the first and second-year levels.

One of the most suitable algorithm on the token sequence sets is the Convolutional Neural Network, due to its pattern recognition capability. As part of the research we have also investigated how CNN's would assist grading in a real-life scenario. Using CNN's on the full dataset did result in reassuring F1 scores, however, using the same model with a manually created different question train/validation split showed that the learnt weights are not truly generalisable.

An interesting and consistent conclusion relates to ByteCode, which not just requires fewer resources to train, but provides more consistent results. This can be attributed to the fact that the structure is more recognizable for every algorithm apart from CNN.

The investigations have not yielded any models that are general enough to successfully assess random pieces of code. The models that are successful are only capable of assessing code that answers the same question as at least some of the training code instances.

Based on the research, the final conclusion is that measuring the level of correctness of code using machine learning might be possible if the training data contains only one question and has more than 100 instances in each category.

## 5.1 Proposal for Application

To be able to represent the personal views of the instructors, our proposal is to train a CNN for each instructor and for each question. This would create an individual model for each instructor and each subject, overcoming the issue of different labelling and personal bias. As discussed in section 3.3.1 we can estimate that a minimum of 400 submissions with equal distribution of categories for the same questions would be a suitable training set. This, along with the fine-graded results and our pre-trained network can create a suitable model for each instructor and question, which they can use to grade their students.

We propose to use individual algorithms which have to be trained for each question for each individual teacher. This can be set up as an API, and could be readily available for the teachers to use.

## 5.2 Future Work

Hyperparameterization is crucial in any model, and although we have tried to build the most accurate models, it might be possible to create an even better-optimised CNN with a different dataset. We can envision further research in concentrating solely on CNNs for automatic programming assignment grading, experimenting with more in-depth parameter sets and algorithm structure modifications. This would warrant separate research by itself and could lead to a more accurate solution.

Reinforcement Learning is also a viable option, however as our results show, the individual training of teachers and question-based algorithms didn't require extensive resources, therefore it might not be necessary. Reinforcement Learning and Transfer learning would be the most optimal solution if a widely accepted and commonly agreed coding practice, along with a grading practice could be implemented. This is a huge task, which is not part of our remit, however, it would pave the way to a secure and well-implemented practice, which would aim to assist in teaching students the best possible way to code.

Although equally not researched here, we believe the same methodology and algorithms would work on any programming language, not just Python. This is especially true in languages that are compiled to a lower-level language as we discussed when comparing ByteCode to Python.

It could also be plausible to imagine further research that would repeat the steps directly on machine language, which would provide a truly language-independent and generalizable way to grade student coding on the finest level.

# References

- Allamanis, M., Barr, E.T., Ducouso, S. and Gao, Z. (2020) Typilus: Neural type hints in: *Proceedings of the 41st acm sigplan conference on programming language design and implementation* pp. 91–105
- Alon, U., Zilberstein, M., Levy, O. and Yahav, E. (2019) code2vec: Learning distributed representations of code *Proceedings of the ACM on Programming Languages* **3**(POPL), pp. 1–29
- Azcona, D., Arora, P., Hsiao, I.H. and 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
- Azcona, D. and Smeaton, A. (2019) +5 million python & bash programming submissions for 5 courses & grades for computer-based exams over 3 academic years. [https://figshare.com/articles/dataset/\\_5\\_Million\\_Python\\_Bash\\_Programming\\_Submissions\\_for\\_5\\_Courses\\_Grades\\_for\\_Computer-Based\\_Exams\\_over\\_3\\_academic\\_years\\_/12610958](https://figshare.com/articles/dataset/_5_Million_Python_Bash_Programming_Submissions_for_5_Courses_Grades_for_Computer-Based_Exams_over_3_academic_years_/12610958) accessed: 2020-06-15
- Bui, N.D.Q., Yu, Y. and Jiang, L. (2020) Infercode: Self-supervised learning of code representations by predicting subtrees
- Campbell, G.A. and Papapetrou, P.P. (2013) *SonarQube in action* Manning Publications Co.
- Checkstyle (2017) Checkstyle program location <https://checkstyle.sourceforge.io/> accessed: 2022-09-15
- Chen, H.M., Chen, W.H. and Lee, C.C. (2018) An automated assessment system for analysis of coding convention violations in java programming assignments\* *Journal of Information Science and Engineering* **34**, pp. 1203–1221
- Cheng, D., Gong, Y., Zhou, S., Wang, J. and Zheng, N. (2016) Person re-identification by multi-channel parts-based cnn with improved triplet loss function in: *Proceedings of the IEEE conference on computer vision and pattern recognition* pp. 1335–1344
- Codelion (2012) Pathgrind program location <https://github.com/codelion/pathgrind/> accessed: 2022-09-15
- Combéfis, S. (2022) Automated code assessment for education: review, classification and perspectives on techniques and tools *Software* **1**(1), pp. 3–30

- Drummond, A., Lu, Y., Chaudhuri, S., Jermaine, C., Warren, J. and Rixner, S. (2014) Learning to grade student programs in a massive open online course in: *2014 IEEE International Conference on Data Mining* pp. 785–790
- Edwards, S.H. and Perez-Quinones, M.A. (2008) Web-cat: automatically grading programming assignments in: *Proceedings of the 13th annual conference on Innovation and technology in computer science education* pp. 328–328
- Fangohr, H., O’Brien, N., Prabhakar, A. and Kashyap, A. (2015) Teaching python programming with automatic assessment and feedback provision *arXiv preprint arXiv:1509.03556*
- Glassman, E.L., Scott, J., Singh, R., Guo, P.J. and Miller, R.C. (2015) Overcode: Visualizing variation in student solutions to programming problems at scale *ACM Transactions on Computer-Human Interaction (TOCHI)* **22**(2), pp. 1–35
- GraderScope (2020) Graderscope program location <https://gradescope-autograders.readthedocs.io/en/latest/> accessed: 2022-09-15
- Gulli, A. and Pal, S. (2017) *Deep learning with Keras* Packt Publishing Ltd
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I.H. (2009) The weka data mining software: an update *ACM SIGKDD explorations newsletter* **11**(1), pp. 10–18
- Hegarty-Kelly, E. and Mooney, D.A. (2021) Analysis of an automatic grading system within first year computer science programming modules in: *Computing Education Practice 2021 CEP ’21* p. 17–20 Association for Computing Machinery, New York, NY, USA
- Hollingsworth, J. (1960) Automatic graders for programming classes *Communications of the ACM* **3**(10), pp. 528–529
- Huang, J., Piech, C., Nguyen, A. and Guibas, L. (2013) Syntactic and functional variability of a million code submissions in a machine learning mooc in: *AIED 2013 Workshops Proceedings Volume* vol. 25 Citeseer
- Jayapati, V.S. and Venkitaraman, A. (2019) A comparison of information retrieval techniques for detecting source code plagiarism *CoRR* **abs/1902.02407**
- Kapur, R., Sodhi, B., Rao, P.U. and Sharma, S. (2021) Using paragraph vectors to improve our existing code review assisting tool-cruso in: *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)* pp. 1–11
- Kharat, A., Kumbhakarn, S., Kolhe, A., Telang, A. and Naglot, D. (2019) Code review and analysis using deep learning *IJRAR* **Volume 6, Issue 2**
- Lan, A.S., Vats, D., Waters, A.E. and Baraniuk, R.G. (2015) Mathematical language processing: Automatic grading and feedback for open response mathematical questions in: *Proceedings of the second (2015) ACM conference on learning@ scale* pp. 167–176
- Lee, S., Han, H., Cha, S.K. and Son, S. (2020) Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer in: *29th USENIX Security Symposium (USENIX Security 20)* pp. 2613–2630

- Li, H., Shi, S., Thung, F., Huo, X., Xu, B., Li, M. and Lo, D. (2019) Deepreview: automatic code review using deep multi-instance learning. in: *Pacific-Asia Conference on Knowledge Discovery and Data Mining* vol. 1 pp. 318–330
- Liu, X., Wang, S., Wang, P. and Wu, D. (2019) Automatic grading of programming assignments: An approach based on formal semantics in: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* pp. 126–137
- Madera, M. and Tomoń, R. (2017) A case study on machine learning model for code review expert system in software engineering in: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* pp. 1357–1363
- McCabe, T. (1976) A complexity measure *IEEE Transactions on Software Engineering* **SE-2**(4), pp. 308–320
- Meurer, A. (2018) Brown water python documentation <https://www.asmeurer.com/brown-water-python/tokens.html> accessed: 2022-09-15
- Mikolov, T. (2015) word2vec: Tool for computing continuous distributed representations of words *Google Code* <https://code.google.com/archive/p/word2vec>
- Mir, A.M., Latoskinas, E., Proksch, S. and Gousios, G. (2021) Type4py: Deep similarity learning-based type inference for python *CoRR* **abs/2101.04470**
- NCCEDU (2017) Ncc grading <https://www.nccedu.com/wp-content/uploads/2017/01/ITP-Sample-Assignment-Marking-Scheme.pdf> accessed: 2022-09-15
- NCL (2020) Ncl <https://www.staff.ncl.ac.uk/andrey.mokhov/EEE1008/marking-criteria.pdf> accessed: 2022-09-15
- Nguyen, Q. (2019) *Hands-on application development with PyCharm: Accelerate your python applications using practical coding techniques in PyCharm* Packt Publishing Ltd
- Pan, S.J. and Yang, Q. (2009) A survey on transfer learning *IEEE Transactions on knowledge and data engineering* **22**(10), pp. 1345–1359
- Perry, D.M., Kim, D., Samanta, R. and Zhang, X. (2019) Semcluster: clustering of imperative programming assignments based on quantitative semantic features in: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* pp. 860–873
- Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M. and Guibas, L. (2015) Learning program embeddings to propagate feedback on student code in: *International conference on machine Learning* pp. 1093–1102 PMLR
- Pradel, M., Gousios, G., Liu, J. and Chandra, S. (2020) Typewriter: Neural type prediction with search-based validation in: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* pp. 209–220



- PythonSoftwareFoundation (2022) Disassembler for python bytecode <https://docs.python.org/3/library/dis.html> accessed: 2022-09-15
- Ragkhitwetsagul, C., Krinke, J. and Clark, D. (2018) A comparison of code similarity analysers *Empirical Software Engineering*
- Rai, K.K., Gupta, B., Shokeen, P. and Chakraborty, P. (2019) Question independent automated code analysis and grading using bag of words and machine learning in: *2019 International Conference on Computing, Power and Communication Technologies (GUCON)* pp. 93–98
- Setoodeh, Z., Reza Moosavi, M., Fakhrahmad, M. and Bidoki, M. (2021) A proposed model for source code reuse detection in computer programs *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*
- Singh, G., Srikant, S. and Aggarwal, V. (2016) Question independent grading using machine learning: The case of computer program grading in: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* pp. 263–272
- Srikant, S. and Aggarwal, V. (2013) Automatic grading of computer programs: A machine learning approach in: *2013 12th International Conference on Machine Learning and Applications* vol. 1 pp. 85–92 IEEE
- Srikant, S. and Aggarwal, V. (2014) A system to grade computer programming skills using machine learning in: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* pp. 1887–1896
- Stanford (2017) Stanford university grading [https://web.stanford.edu/class/archive/cs/cs107/cs107.1166/advice\\_assigngrade.html](https://web.stanford.edu/class/archive/cs/cs107/cs107.1166/advice_assigngrade.html) accessed: 2022-09-15
- UIUC (2020) University of illinois grading <https://citl.illinois.edu/citl-101/measurement-evaluation/exam-scoring/assigning-course-grades> accessed: 2022-09-15
- UT (2017) University of texas grading <https://www.cs.utexas.edu/~mitra/csSpring2017/cs313/grading.html> accessed: 2022-09-15
- UWO (2018) University of aiowa grading <https://www.csd.uwo.ca/~sbeauche/CS3388/CS3388-Marking-Scheme.html> accessed: 2022-09-15
- Wilcox, C. (2015) The role of automation in undergraduate computer science education in: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education SIGCSE '15* p. 90–95 Association for Computing Machinery, New York, NY, USA
- Ye, T., Zhang, L., Wang, L. and Li, X. (2016) An empirical study on detecting and fixing buffer overflow bugs in: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* pp. 91–101 IEEE

# Appendix A

## Resources Used

### A.1 Resources used

1. Most of the algorithms were able to run with *Google Colab Pro* with *Tensorflow* using the High RAM environment with GPU and TPU where it was necessary. The time limit on the High Ram GPU was 12 hours and 24 hours on High RAM with CPU
2. A local PC running on an *AMD Ryzen 5 3600* CPU with 64 GB memory using an *NVIDIA GeForce RTX 3060 Ti* with 8 GB of memory was used for longer processes. The local machine used the *NVIDIA Cuda* environment with *Anaconda*.
3. Some models required more resources and even longer run times than the ones provided by Google or the local machine, therefore the biggest models were run on *Amazon Web Services tensorflow2 p37* machines.



# Appendix B

## Algorithm Packages

### B.1 Versions used

Numpy Version: 1.21.6

Tensorflow Version: 2.9.2

Keras Version: 2.9.0

Pandas Version: 1.3.5

Seaborn Version: 0.11.2

Sklearn Version: 1.0.2

Matplotlib Version: 3.2.2



# Appendix C

## Confusion Matrix Heatmaps

This appendix contains the confusion matrix heatmaps related to results in Section 4. Please note that the results in Section 4 are the average of a 5-fold cross-validation across 3 runs, while the heatmaps only represent one of these runs.

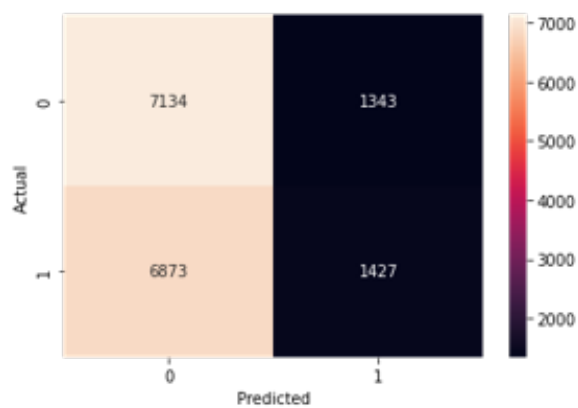


Fig. C.1 Heatmap for Table 4.1 ANN - Code Length Results - Python

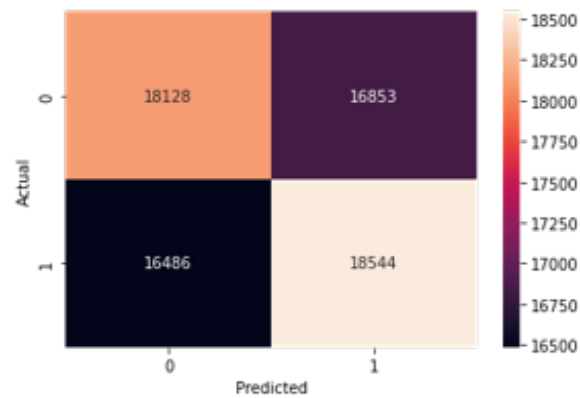


Fig. C.2 Heatmap for Table 4.2 MNB - Token Count Results - Python Token and Keyword Count Set

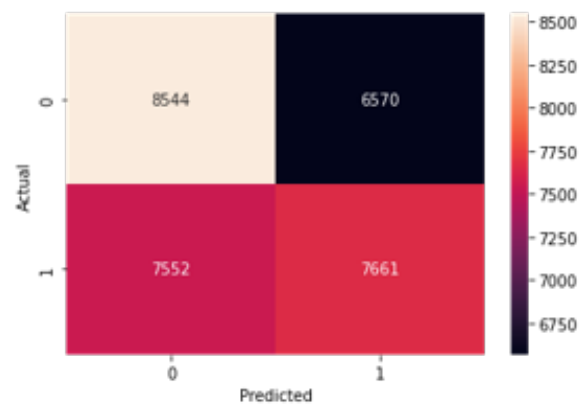


Fig. C.3 Heatmap for Table 4.3 MNB - Token Count Results - ByteCode

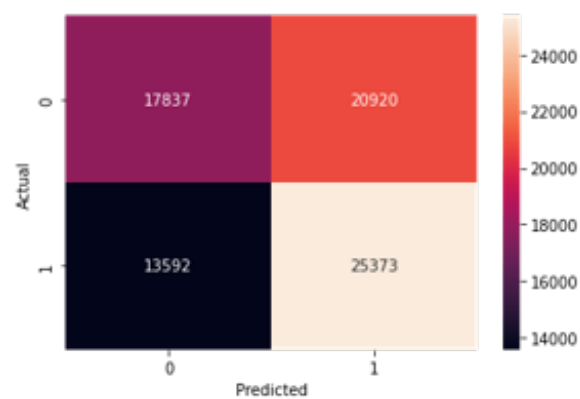


Fig. C.4 Heatmap for Table 4.4 SVM - Token Count Results - Python Token and Keyword Count Set

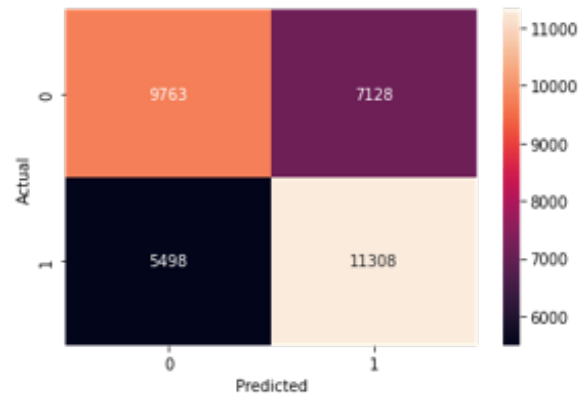


Fig. C.5 Heatmap for Table 4.5 SVM - Token Count Results - ByteCode

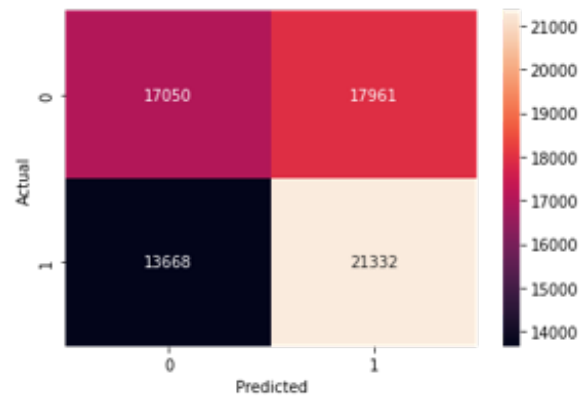


Fig. C.6 Heatmap for Table 4.6 Logistic Regression - Token Count Results - Python Token and Keyword Count Set

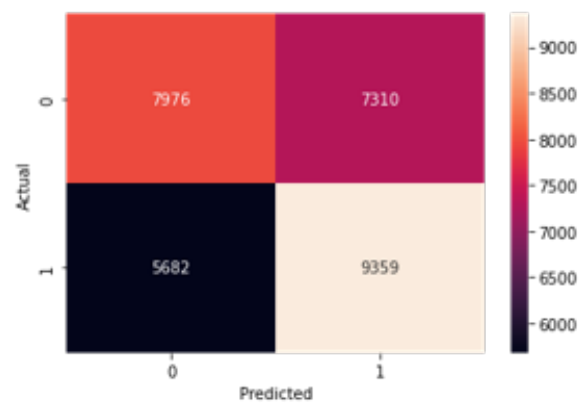


Fig. C.7 Heatmap for Table 4.7 Logistic Regression - Token Count Results - ByteCode



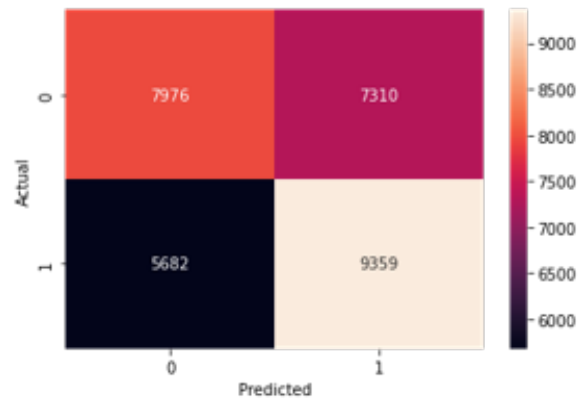


Fig. C.8 Heatmap for Table 4.7 Logistic Regression - Token Count Results - ByteCode

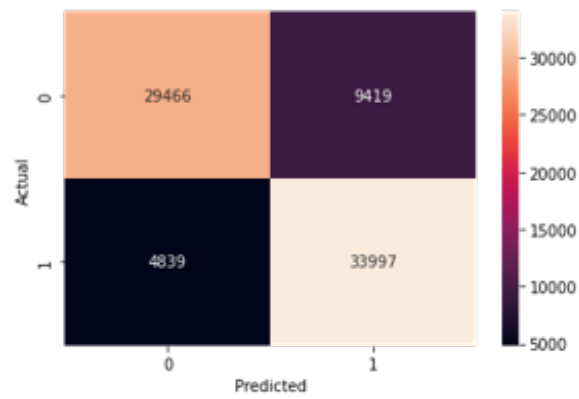


Fig. C.9 Heatmap for Table 4.8 Decision Tree - Token Count Results - Python Token and Keyword Count Set

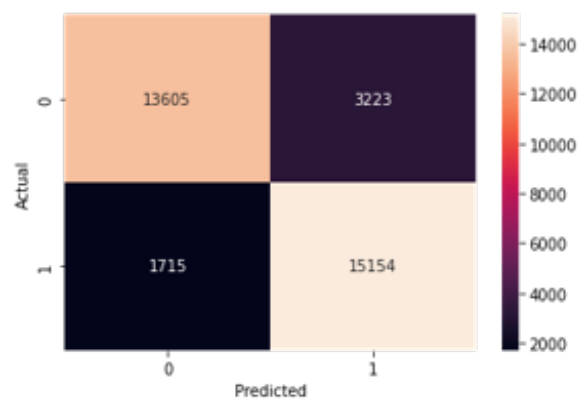


Fig. C.10 Heatmap for Table 4.9 Decision Tree - Token Count Results - ByteCode

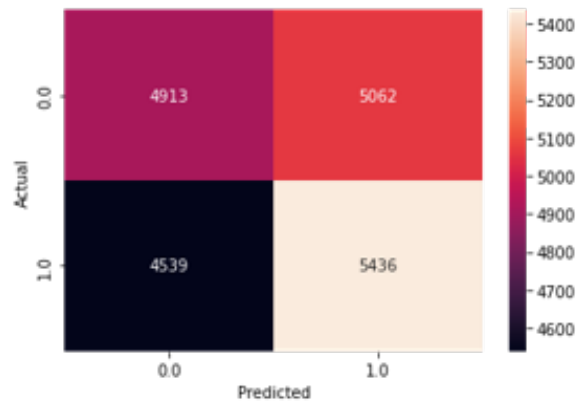


Fig. C.11 Heatmap for Table 4.10 Decision Tree - Keyword Count Results - Python - Question Independent

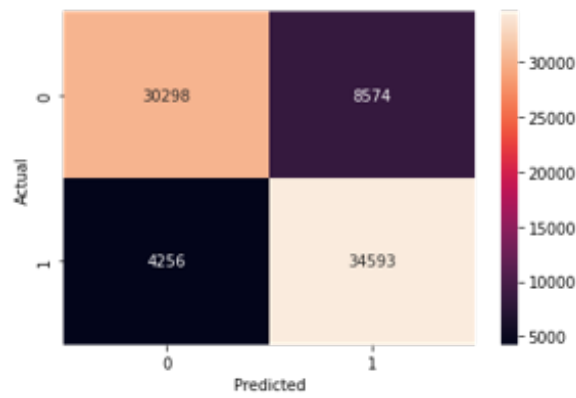


Fig. C.12 Heatmap for Table 4.11 Random Forest - Keyword and Token Count Results - Python

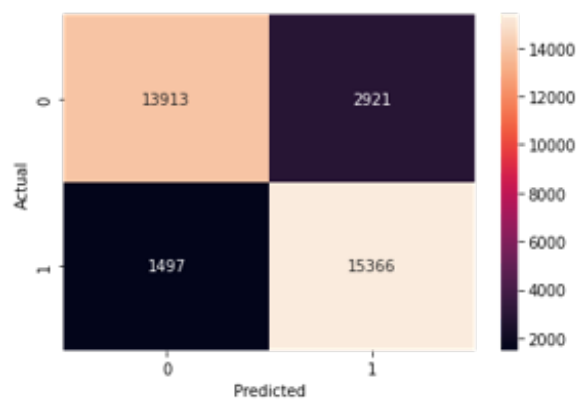


Fig. C.13 Heatmap for Table 4.12 Random Forest - Token Count Results - ByteCode

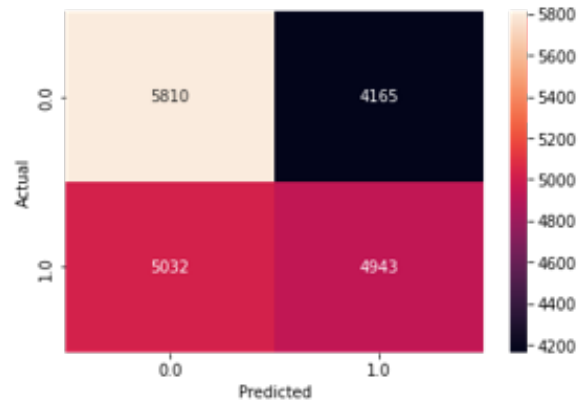


Fig. C.14 Heatmap for Table 4.13 Random Forest - Keyword and Token Count Results - Python - Question Independent

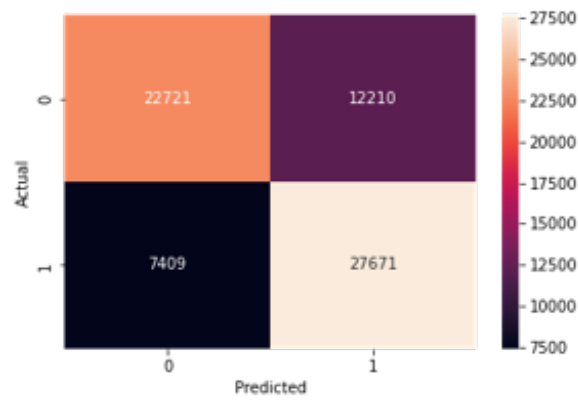


Fig. C.15 Heatmap for Table 4.14 ANN - Token Sequence Results - Python

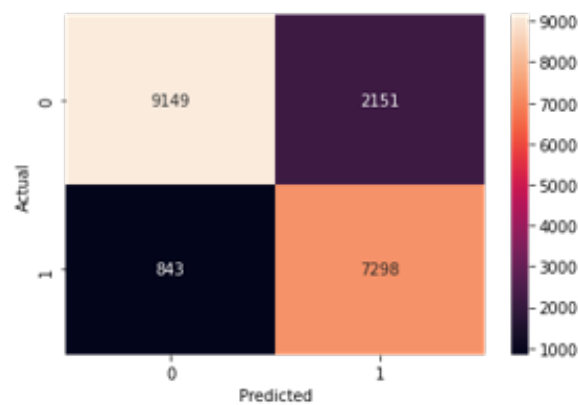


Fig. C.16 Heatmap for Table 4.15 ANN - Token Sequence Results - ByteCode

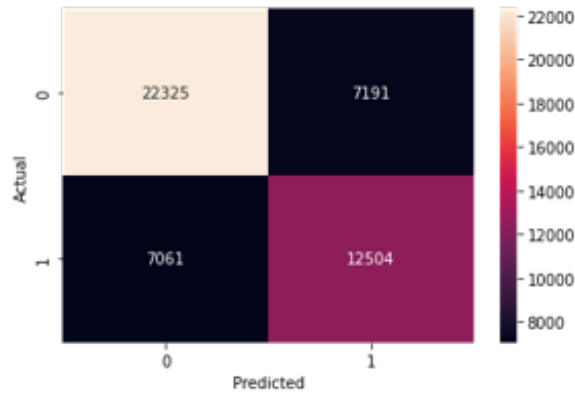


Fig. C.17 Heatmap for Table 4.16 LSTM - Token Sequence Results - Python

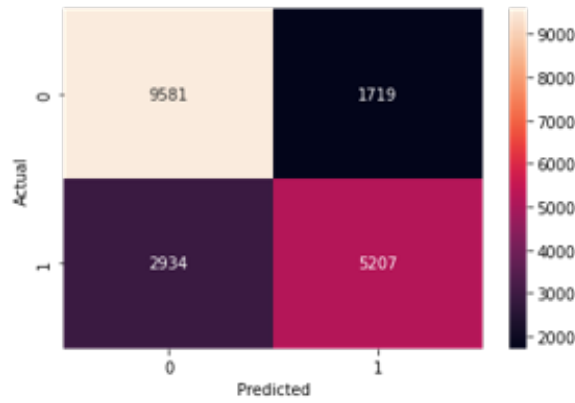


Fig. C.18 Heatmap for Table 4.17 LSTM - Token Sequence Results - ByteCode

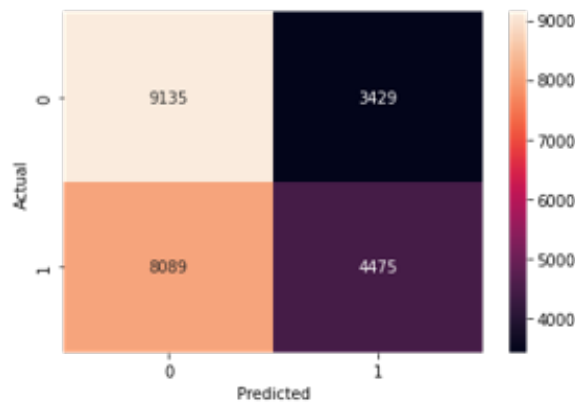


Fig. C.19 Heatmap for Table 4.18 LSTM - Token Sequence Results - Python - Question Independent

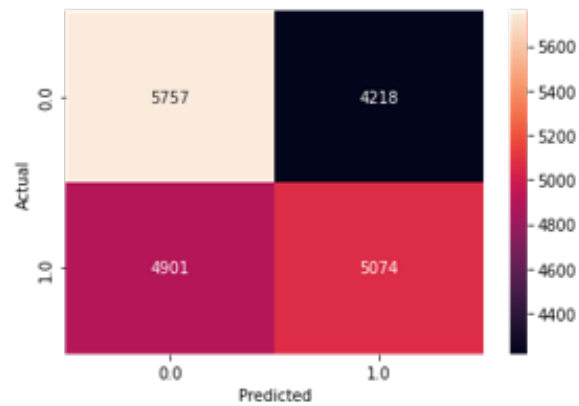


Fig. C.20 Heatmap for Table 4.19 LSTM - Token Sequence Results - ByteCode - Question Independent

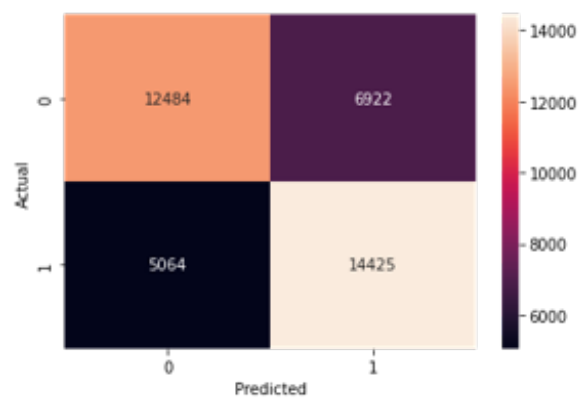


Fig. C.21 Heatmap for Table 4.20 RNN - Token Sequence Results - Python

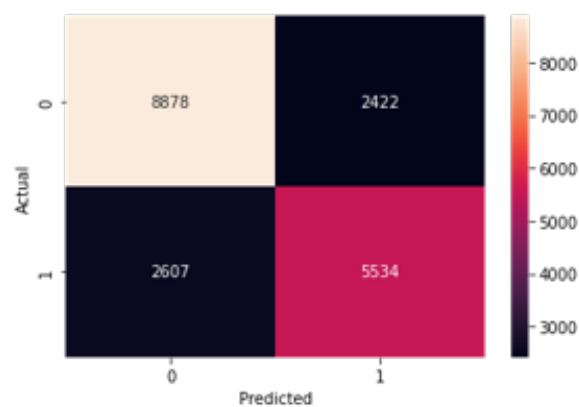


Fig. C.22 Heatmap for Table 4.21 RNN - Token Sequence Results - ByteCode

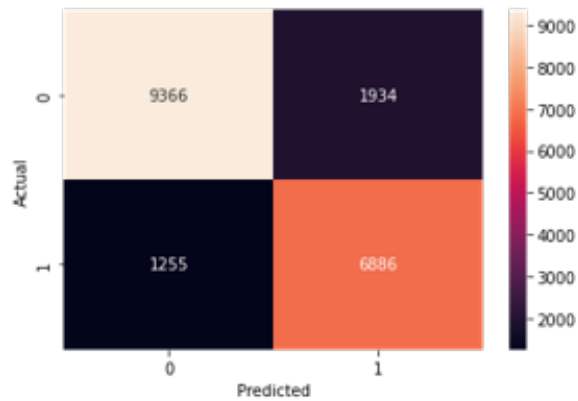


Fig. C.23 Heatmap for Table 4.22 CNN - Token Sequence Results - Python

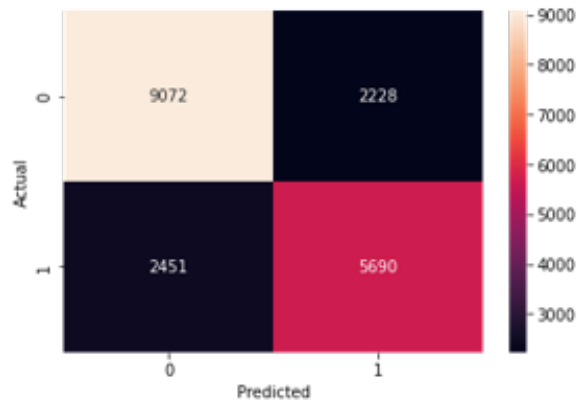


Fig. C.24 Heatmap for Table 4.23 CNN - Token Sequence Results - ByteCode

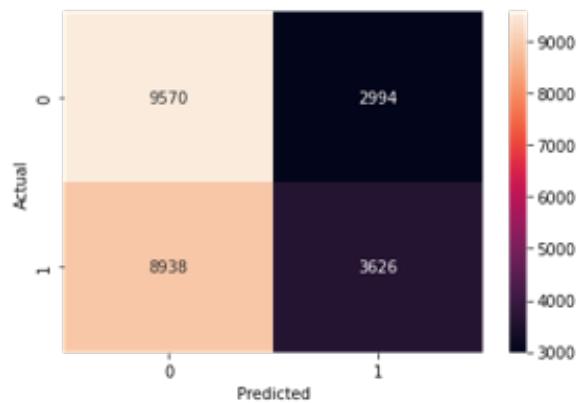


Fig. C.25 Heatmap for Table 4.24 CNN - Token Sequence Results - Python - Question Independent

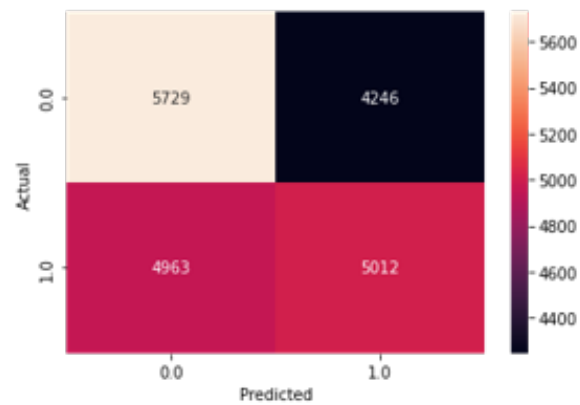


Fig. C.26 Heatmap for Table 4.25 CNN - Token Sequence Results - ByteCode - Question Independent

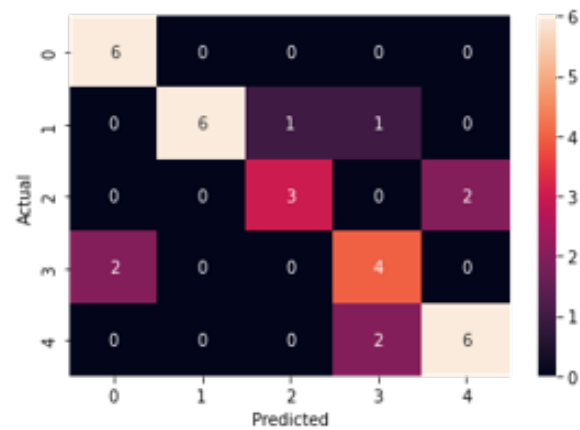


Fig. C.27 Heatmap for Table 4.27 LSTM - Token Sequence Results - Transfer Learning - Python

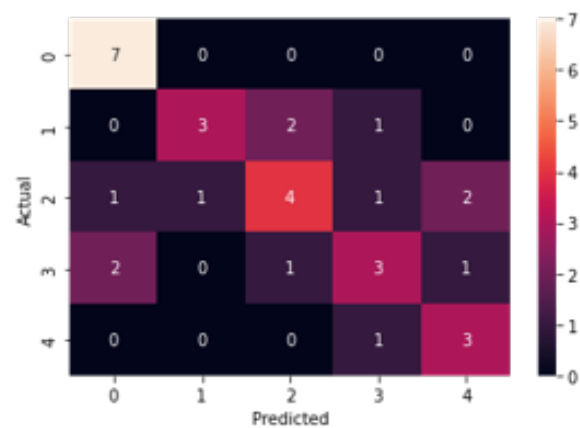


Fig. C.28 Heatmap for Table 4.28 LSTM - Token Sequence Results - Transfer Learning - Python

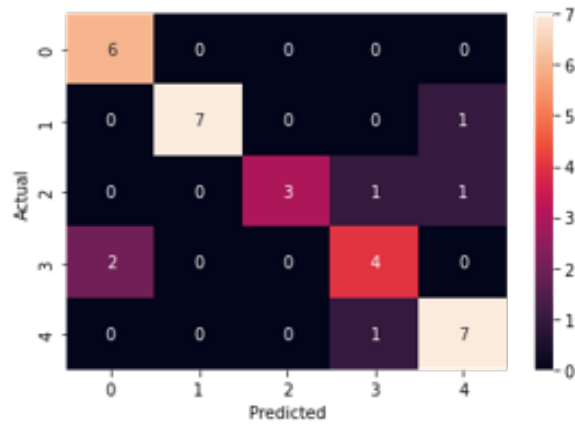


Fig. C.29 Heatmap for Table 4.30 CNN - Token Sequence Results - Transfer Learning - Python

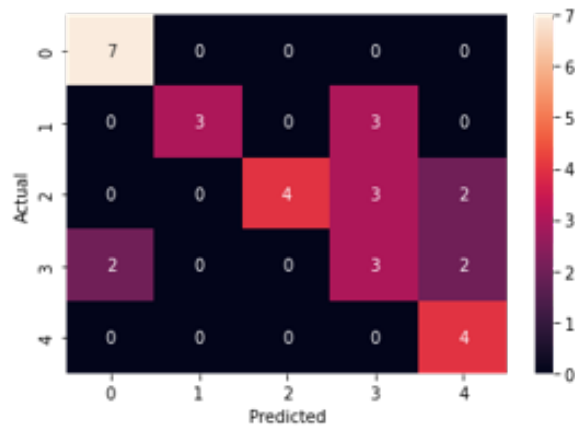


Fig. C.30 Heatmap for Table 4.31 CNN - Token Sequence Results - Transfer Learning - Python





# Appendix D

## Source Code

### D.1 Tokenizer parameters

All tokenizers were using the below configuration, to remove special characters, numbers, line breaks, tabs and new lines. This was necessary to reduce the number of tokens, therefore, reducing noise, training time and resource requirements. The MAX\_NB\_WORDS variable was set to the maximum number of words to be tokenized.

```
1 Tokenizer(num_words=MAX_NB_WORDS, filters=  
2 '!"#$%&()*+,-./:;<=>?@[\\]^`{|}~0123456789\n\t\r',  
3 lower=True)
```

Listing D.1 Tokenizer used

### D.2 Multinomial Naïve Bayes code

Multinomial Naïve Bayes (5 fold 3 times) code used in 3.2.2

```
1 from sklearn.naive_bayes import MultinomialNB  
2 from sklearn.model_selection import KFold  
3 from sklearn.model_selection import cross_val_score  
4 from sklearn.naive_bayes import MultinomialNB  
5 from sklearn.metrics import accuracy_score,  
6 from sklearn.metrics import classification_report  
7 from sklearn.metrics import confusion_matrix  
8 import matplotlib.pyplot as plt  
9 import seaborn as sns  
10  
11 # Define per-fold score containers
```

```
12 acc_per_fold = []
13 loss_per_fold = []
14
15 # Model configuration
16 verbosity = 1
17 num_folds = 5
18
19 inputs = np.concatenate((X_train, X_test), axis=0)
20 targets = np.concatenate((Y_train, Y_test), axis=0)
21
22
23 # set to do the K-fold 3 times
24 for no_full_repeat in range(3):
25
26     # Define the K-fold Cross Validator
27     kfold = KFold(n_splits=num_folds, shuffle=True)
28
29
30     # K-fold Cross Validation model evaluation
31     fold_no = 1
32     for train, test in kfold.split(X_train, Y_train):
33
34         # Generate a print
35         print('-----')
36         print(f'Training for fold {fold_no} ...')
37
38         mnb = MultinomialNB()
39         mnb.fit(inputs[train], targets[train])
40
41         print("score on test: " + str(mnb.score(inputs[test], targets[test]
42         )))
43         print("score on train: " + str(mnb.score(inputs[train], targets[
44         train])))
45
46         #prediction on test data
47         pred=mnb.predict(inputs[test])
48         #plot confusion matrix
49         results = {'y_Actual':targets[test], 'y_Predicted':pred}
50         df_results = pd.DataFrame(results, columns=['y_Actual', '
51         y_Predicted'])
52         confusion_matrix = pd.crosstab(df_results['y_Actual'],
53         df_results['y_Predicted'], rownames=['Actual'], colnames=['
54         Predicted'])
55         sns.heatmap(confusion_matrix, annot=True, fmt='d')
```

```

52     plt.show()
53
54     #print classification report
55     print(classification_report(targets[test], pred))
56
57     # Increase fold number
58     fold_no = fold_no + 1
59
60     print('-----')
61     print('Number of repeats:', no_full_repeat)
62     print('-----')
63
64     print('-----')
65     print('Completed', no_full_repeat)
66     print('-----')

```

Listing D.2 MNB code

## D.3 Support Vector Machine code

Support Vector Machine code snippet used in 3.2.2.

```

1  from sklearn.model_selection import KFold
2  from sklearn.model_selection import cross_val_score
3  from sklearn.svm import SVC
4  from sklearn.metrics import accuracy_score
5  from sklearn.metrics import classification_report
6  from sklearn.metrics import confusion_matrix
7  import matplotlib.pyplot as plt
8  import seaborn as sns
9
10 # Define per-fold score containers
11 acc_per_fold = []
12 loss_per_fold = []
13
14 # Model configuration
15 verbosity = 1
16 num_folds = 5
17
18 inputs = np.concatenate((X_train, X_test), axis=0)
19 targets = np.concatenate((Y_train, Y_test), axis=0)
20
21 # set to do the K-fold 3 times

```

```
22
23 for no_full_repeat in range (3):
24
25     # Define the K-fold Cross Validator
26     kfold = KFold(n_splits=num_folds, shuffle=True)
27
28     # K-fold Cross Validation model evaluation
29     fold_no = 1
30     for train, test in kfold.split(inputs, targets):
31
32         # Generate a print
33         print('-----')
34         print(f'Training for fold {fold_no} ...')
35
36         svc = SVC()
37         svc.fit(inputs[train], targets[train])
38
39         print("score on test: " + str(svc.score(inputs[test], targets[test]
40         )))
41         print("score on train: "+ str(svc.score(inputs[train], targets[
42         train])))
43
44         #prediction on test data
45         pred=svc.predict(inputs[test])
46
47         #plot confusion matrix
48         results = {'y_Actual':targets[test], 'y_Predicted':pred}
49         df_results = pd.DataFrame(results, columns=['y_Actual', '
50         y_Predicted'])
51         confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results
52         ['y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
53         sns.heatmap(confusion_matrix, annot=True, fmt='d')
54         plt.show()
55
56         #print classification report
57         print(classification_report(targets[test], pred))
58
59         # Increase fold number
60         fold_no = fold_no + 1
61
62     print('-----')
63     print('Number of repeats:', no_full_repeat)
64     print('-----')
```

```
62 print('-----')
63 print('Completed', no_full_repeat)
64 print('-----')
```

Listing D.3 SVM code

## D.4 Logistic Regression code

Logistic Regression code used in 3.2.2.

```
1 from sklearn.model_selection import KFold
2 from sklearn.model_selection import cross_val_score
3 from sklearn.linear_model import LogisticRegression
4 from sklearn.metrics import accuracy_score
5 from sklearn.metrics import classification_report
6 from sklearn.metrics import confusion_matrix
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10
11 # Define per-fold score containers
12 acc_per_fold = []
13 loss_per_fold = []
14
15 # Model configuration
16 verbosity = 1
17 num_folds = 5
18
19 inputs = np.concatenate((X_train, X_test), axis=0)
20 targets = np.concatenate((Y_train, Y_test), axis=0)
21
22
23 # set to do the K-fold 3 times
24
25 for no_full_repeat in range (3):
26
27     # Define the K-fold Cross Validator
28     kfold = KFold(n_splits=num_folds, shuffle=True)
29
30
31     # K-fold Cross Validation model evaluation
32     fold_no = 1
33     for train, test in kfold.split(X_train, Y_train):
```

```

34
35 # Generate a print
36 print('-----')
37 print(f'Training for fold {fold_no} ...')
38
39 lr=LogisticRegression(max_iter=20000)
40 lr.fit(inputs[train], targets[train])
41
42 print("score on test: " + str(lr.score(inputs[test], targets[test
43 ])))
44 print("score on train: "+ str(lr.score(inputs[train], targets[
45 train])))
46
47 #prediction on test data
48 pred=lr.predict(inputs[test])
49
50 #plot confusion matrix
51 results = {'y_Actual':targets[test], 'y_Predicted':pred}
52 df_results = pd.DataFrame(results, columns=['y_Actual', '
53 y_Predicted'])
54 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results
55 ['y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
56 sns.heatmap(confusion_matrix, annot=True, fmt='d')
57 plt.show()
58 #print classification report
59 print(classification_report(targets[test], pred))
60
61 # Increase fold number
62 fold_no = fold_no + 1
63
64 print('-----')
65 print('Number of repeats:', no_full_repeat)
66 print('-----')
67
68 print('-----')
69 print('Completed', no_full_repeat)
70 print('-----')

```

Listing D.4 SVM code

## D.5 Decision Tree code

Decision Tree code used in 3.2.2.

```
1 from sklearn.model_selection import KFold
2 from sklearn.model_selection import cross_val_score
3 from sklearn.tree import DecisionTreeClassifier
4 from sklearn.metrics import accuracy_score
5 from sklearn.metrics import classification_report
6 from sklearn.metrics import confusion_matrix
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10
11 # Define per-fold score containers
12 acc_per_fold = []
13 loss_per_fold = []
14
15 # Model configuration
16 verbosity = 1
17 num_folds = 5
18
19 inputs = np.concatenate((X_train, X_test), axis=0)
20 targets = np.concatenate((Y_train, Y_test), axis=0)
21
22
23 # set to do the K-fold 3 times
24
25 for no_full_repeat in range (3):
26
27     # Define the K-fold Cross Validator
28     kfold = KFold(n_splits=num_folds, shuffle=True)
29
30
31     # K-fold Cross Validation model evaluation
32     fold_no = 1
33     for train, test in kfold.split(X_train, Y_train):
34
35         # Generate a print
36         print('-----')
37         print(f'Training for fold {fold_no} ...')
38
39         clf = DecisionTreeClassifier()
40         clf.fit(inputs[train], targets[train])
41
42         print("score on test: " + str(lr.score(inputs[test], targets[test]
43 ])))
```



```

43     print("score on train: "+ str(lr.score(inputs[train],targets[
train])))
44
45     #prediction on test data
46     pred=lr.predict(inputs[test])
47
48     #plot confusion matrix
49     results = {'y_Actual':targets[test], 'y_Predicted':pred}
50     df_results = pd.DataFrame(results, columns=['y_Actual','
y_Predicted'])
51     confusion_matrix = pd.crosstab(df_results['y_Actual'],df_results[
'y_Predicted'], rownames=['Actual'],colnames=['Predicted'])
52     sns.heatmap(confusion_matrix, annot=True, fmt='d')
53     plt.show()
54     #print classification report
55     print(classification_report(targets[test], pred))
56
57     # Increase fold number
58     fold_no = fold_no + 1
59
60     print('-----')
61     print('Number of repeats:', no_full_repeat)
62     print('-----')
63
64     print('-----')
65     print('Completed', no_full_repeat)
66     print('-----')

```

Listing D.5 Decision Tree code

## D.6 Random Forest code

Random Forest code used in 3.2.2.

```

1 from sklearn.model_selection import KFold
2 from sklearn.model_selection import cross_val_score
3 from sklearn.ensemble import RandomForestClassifier
4 from sklearn.metrics import accuracy_score
5 from sklearn.metrics import classification_report
6 from sklearn.metrics import confusion_matrix
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9

```

```
10
11 # Define per-fold score containers
12 acc_per_fold = []
13 loss_per_fold = []
14
15 # Model configuration
16 verbosity = 1
17 num_folds = 5
18
19 inputs = np.concatenate((X_train, X_test), axis=0)
20 targets = np.concatenate((Y_train, Y_test), axis=0)
21
22
23 # set to do the K-fold 3 times
24
25 for no_full_repeat in range (3):
26
27     # Define the K-fold Cross Validator
28     kfold = KFold(n_splits=num_folds, shuffle=True)
29
30
31     # K-fold Cross Validation model evaluation
32     fold_no = 1
33     for train, test in kfold.split(X_train, Y_train):
34
35         # Generate a print
36         print('-----')
37         print(f'Training for fold {fold_no} ...')
38
39         rf = RandomForestClassifier(n_estimators=20, max_depth=40)
40         rf.fit(inputs[train], targets[train])
41
42         print("score on test: " + str(lr.score(inputs[test],
43         targets[test])))
44         print("score on train: " + str(lr.score(inputs[train],
45         targets[train])))
46
47         #prediction on test data
48         pred=lr.predict(inputs[test])
49
50         #plot confusion matrix
51         results = {'y_Actual':targets[test], 'y_Predicted':pred}
52         df_results = pd.DataFrame(results, columns=['y_Actual',
53         'y_Predicted'])
```

```

54     confusion_matrix = pd.crosstab(df_results['y_Actual'],
55     df_results['y_Predicted'], rownames=['Actual'],
56     colnames=['Predicted'])
57     sns.heatmap(confusion_matrix, annot=True, fmt='d')
58     plt.show()
59     #print classification report
60     print(classification_report(targets[test], pred))
61
62     # Increase fold number
63     fold_no = fold_no + 1
64
65     print('-----')
66     print('Number of repeats:', no_full_repeat)
67     print('-----')
68
69     print('-----')
70     print('Completed', no_full_repeat)
71     print('-----')

```

Listing D.6 Random Forest code

## D.7 Deep Learning - Tokenizer code and imports

Deep Learning Tokenizer code used in 3.2.1,3.2.3,3.2.3,3.2.3.

```

1 # Import everything necessary
2 import numpy as np
3 from numpy import genfromtxt
4 from tensorflow.keras.preprocessing import sequence
5 from tensorflow.keras.models import Sequential
6 from tensorflow.keras.layers import Dense, Dropout, Activation,
7     Flatten, Embedding, Conv1D, GlobalMaxPooling1D,
8     GlobalAveragePooling1D, LSTM, SimpleRNN, SpatialDropout1D
9 import tensorflow as tf
10 from tensorflow import keras
11 import pandas as pd
12 from keras.preprocessing.text import Tokenizer
13 from keras.preprocessing.sequence import pad_sequences
14 from sklearn.model_selection import train_test_split
15 from keras.callbacks import EarlyStopping, ModelCheckpoint
16 import matplotlib.pyplot as plt
17 %matplotlib inline

```

```
17 # Read the dataset with Pandas and making sure it is all string first
    for Categorical second for regression
18 df = pd.read_csv('/programming_data_clean_py_nocomment_15K45.csv',
    header=0)
19
20 # Comment this out for K-means clustering
21 df=df.astype(str)
22 df.info()
23
24 # Set the maximum number of words to be used.
25 MAX_NB_WORDS = 50
26
27 # Max number of words in each code
28 MAX_SEQUENCE_LENGTH = 100
29
30 # This is fixed throughout the project
31 EMBEDDING_DIM = 50
32
33 # Tokenize words and filter out characters not wanted, switch to
    lowercase - to avoid too much tokens, remove tabs and new lines
    and numbers
34 tokenizer = Tokenizer(num_words=MAX_NB_WORDS, filters='!"#$%&()
    *+, -./:;<=>?@[\\]^`{|}~0123456789\n\t\r', lower=True)
35 tokenizer.fit_on_texts(df['code'].values)
36 word_index = tokenizer.word_index
37 print('Found %s unique tokens.' % len(word_index))
38 print(word_index)
39
40 # Create sequence from tokens
41 X = tokenizer.texts_to_sequences(df['code'].values)
42 X = pad_sequences(X, maxlen=MAX_SEQUENCE_LENGTH)
43 print('Shape of data tensor:', X.shape)
44 # change to this for NLP based ANN, LSTM
45 Y = pd.get_dummies(df['result']).values
46 # change to this for binary
47 #Y = (df['result']).values
48 print('Shape of label tensor:', Y.shape)
49 print(Y)
50
51
52 # Set training and testing dataset, for X and Y, splitting at 10 %
    for testing - this will be used on all models to be able to
    compare
```

```
53 X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size =
    0.10, random_state = 42)
54 print(X_train.shape,Y_train.shape)
55 print(X_test.shape,Y_test.shape)
56
57 # Explore data
58 class_counts_tr = df.groupby('result').size()
59 print('Training classes:')
60 print(class_counts_tr)
61 print('Training Average text length:')
62 result = pd.DataFrame([[[]]])
63 for col in df:
64     result['code'] = df['code'].apply(len).mean()
65 print(result)
66 print('Training Max text length:')
67 for col in df:
68     result['code'] = df['code'].apply(len).max()
69 print(result)
70 print('Training Min text length:')
71 for col in df:
72     result['code'] = df['code'].apply(len).min()
73 print(result)
74 print('X sample')
75 print(X)
```

Listing D.7 Tokenizer code

## D.8 ANN code

Artificial Neural Network code used in 3.2.1.

```
1 from sklearn.model_selection import KFold
2 from sklearn.model_selection import cross_val_score
3
4
5 # Define per-fold score containers
6 acc_per_fold = []
7 loss_per_fold = []
8
9 # Model configuration
10 batch_size = 64
11 loss_function = 'categorical_crossentropy'
12 no_epochs = 100
```

```
13 verbosity = 1
14 num_folds = 5
15 input_dim = X_train.shape[1]
16
17
18 inputs = np.concatenate((X_train, X_test), axis=0)
19 targets = np.concatenate((Y_train, Y_test), axis=0)
20
21
22 # set to do the K-fold 3 times
23
24 for no_full_repeat in range (2):
25
26     # Define the K-fold Cross Validator
27     kfold = KFold(n_splits=num_folds, shuffle=True)
28
29
30     # K-fold Cross Validation model evaluation
31     fold_no = 1
32     for train, test in kfold.split(X_train, Y_train):
33
34         model = Sequential()
35         model.add(Dense(200, input_dim=input_dim, activation='relu',
36 use_bias=True))
37         model.add(Dropout(0.1))
38         model.add(Dense(2, activation='softmax'))
39
40         # Compile the model
41         model.compile(loss=loss_function, optimizer='adam', metrics=['
42 accuracy'])
43
44         # Generate a print
45         print('-----')
46         print(f'Training for fold {fold_no} ...')
47
48         # Fit data to model
49         history = model.fit(inputs[train], targets[train], epochs=
50 no_epochs, batch_size=batch_size)
51
52         # Generate generalization metrics
53         scores = model.evaluate(inputs[test], targets[test], verbose=0)
54         print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {
55 scores[0]}; {model.metrics_names[1]} of {scores[1]*100}%')
56     # report performance
```

```
53     acc_per_fold.append(scores[1] * 100)
54     loss_per_fold.append(scores[0])
55
56
57     # Evaluate and plot the results
58     accr = model.evaluate(inputs[test], targets[test])
59     print('Test set\n Loss: {:.3f}\n Accuracy: {:.3f}'.format(
accr[0], accr[1]))
60
61     #importing confusion matrix
62     from sklearn.metrics import confusion_matrix
63     #y_pred = model.predict(X_test)
64     y_pred=model.predict(inputs[test])
65     y_pred2=np.argmax(y_pred, axis=1)
66     Y_test2=np.argmax(targets[test], axis=1)
67
68     # Importing the classification report, confusion matrix, and
plotting libraries
69     from sklearn.metrics import accuracy_score, classification_report
, confusion_matrix
70     import seaborn as sns
71     #prediction on test data
72     pred=model.predict(inputs[test])
73     pred2=np.argmax(y_pred, axis=1)
74     test2=np.argmax(targets[test], axis=1)
75
76     #plot confusion matrix
77     results = {'y_Actual':test2, 'y_Predicted':pred2}
78     #print('Results:',results)
79     df_results = pd.DataFrame(results, columns=['y_Actual', '
y_Predicted'])
80     confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results
['y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
81     sns.heatmap(confusion_matrix, annot=True, fmt='d')
82     plt.show()
83     #print classification report
84     print(classification_report(test2, pred2))
85
86     # Increase fold number
87     fold_no = fold_no + 1
88
89
90     # == Provide average scores ==
91     print('-----')
```

```

92 print('Score per fold')
93 for i in range(0, len(acc_per_fold)):
94     print('-----')
95     print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {
acc_per_fold[i]}%')
96     print('-----')
97 print('Average scores for all folds:')
98 print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(
acc_per_fold)})')
99 print(f'> Loss: {np.mean(loss_per_fold)}')
100 print('-----')
101 print('-----')
102 print('Number of repeats:', no_full_repeat)
103 print('-----')
104
105 print('-----')
106 print('Completed', no_full_repeat)
107 print('-----')

```

Listing D.8 ANN code

## D.9 LSTM code

Long-Short Term Memory Network code used in 3.2.3.

```

1 model = Sequential()
2 model.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM, input_length=X_train
    .shape[1]))
3 model.add(SpatialDropout1D(0.1))
4 model.add(LSTM(50, return_sequences=True, dropout=0.1))
5 model.add(LSTM(25, dropout=0.1))
6 model.add(Dense(2, activation='softmax'))
7 model.summary()
8 #model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
9 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[
    'accuracy'])
10 # Setting epoch and batch size - to be the same across the models for
    comparasion
11 epochs = 20
12 batch_size = 128
13
14 # Fitting model while saving history for the plot

```



```
15 print('X:',X_train.shape)
16 print('Y:',Y_train.shape)
17 #history = model.fit(X_train, Y_train, epochs=epochs, batch_size=
    batch_size,validation_split=0.1,callbacks=callbacks_list)
18 history = model.fit(X_train, Y_train, epochs=epochs, batch_size=
    batch_size,validation_split=0.1)
19 #save model
20 model.save('drive/MyDrive/Colab Notebooks/Masters/Data/Models/
    PythonQI_LSTM_ML50_ED50_N50N25_e50 -b128_16092022.h5')
21
22 # Evaluate and plot the results
23 accr = model.evaluate(X_test,Y_test)
24 print('Test set\n  Loss: {:.3f}\n  Accuracy: {:.3f}'.format(accr
    [0],accr[1]))
25 plt.title('Loss')
26 plt.plot(history.history['loss'], label='train')
27 plt.plot(history.history['val_loss'], label='test')
28 plt.legend()
29 plt.show();
30
31 plt.title('Accuracy')
32 plt.plot(history.history['accuracy'], label='train')
33 plt.plot(history.history['val_accuracy'], label='test')
34 plt.legend()
35 plt.show();
36
37 #importing confusion matrix
38 from sklearn.metrics import confusion_matrix
39 #y_pred = model.predict(X_test)
40 y_pred=model.predict(X_test)
41 y_pred2=np.argmax(y_pred, axis=1)
42 Y_test2=np.argmax(Y_test, axis=1)
43 print('Prediction:',y_pred)
44 print('Y Test:',Y_test)
45 confusion = confusion_matrix(Y_test2, y_pred2)
46 print('Confusion Matrix\n')
47 print(confusion)
48
49 #importing accuracy_score, precision_score, recall_score, f1_score
50 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
51 print('\nAccuracy: {:.2f}\n'.format(accuracy_score(Y_test2, y_pred2))
    )
52
```

```
53 print('Micro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='micro')))
54 print('Micro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='micro')))
55 print('Micro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='micro')))
56
57 print('Macro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='macro')))
58 print('Macro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='macro')))
59 print('Macro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='macro')))
60
61 print('Weighted Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='weighted')))
62 print('Weighted Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='weighted')))
63 print('Weighted F1-score: {:.2f}'.format(f1_score(Y_test2, y_pred2,
    average='weighted')))
64
65 # Importing the classification report, confusion matrix, and plotting
    libraries
66 from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
67 import pandas as pd
68 import matplotlib.pyplot as plt
69 import seaborn as sns
70 #prediction on test data
71 pred=model.predict(X_test)
72 pred2=np.argmax(y_pred, axis=1)
73 test2=np.argmax(Y_test, axis=1)
74 print('Predictions:',pred)
75 #plot confusion matrix
76 results = {'y_Actual':test2, 'y_Predicted':pred2}
77 print('Results:',results)
78 df_results = pd.DataFrame(results, columns=['y_Actual','y_Predicted'
    ])
79 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results['
    y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
80 sns.heatmap(confusion_matrix, annot=True, fmt='d')
81 plt.show()
82 #print classification report
```

```
83 print(classification_report(test2, pred2))
```

Listing D.9 LSTM code

## D.10 RNN code

Recurrent Neural Network code used in 3.2.3.

```
1 model = Sequential()
2 model.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM, input_length=X.shape
   [1]))
3 model.add(SimpleRNN(EMBEDDING_DIM))
4 model.add(Dense(116, input_dim= 116, activation="relu"))
5 model.add(Dense(2, activation="softmax"))
6 #model.summary()
7
8 model.compile(loss='categorical_crossentropy', optimizer='adam',
   metrics=['accuracy'])
9
10 epochs = 20
11 batch_size = 128
12
13 import os
14 # Directory where the checkpoints will be saved
15 checkpoint_dir = 'Colab Notebooks/Masters/CheckpointsSRNN'
16 import shutil
17 try:
18     shutil.rmtree(checkpoint_dir)
19 except:
20     print("directory not used yet.")
21
22 # Name of the checkpoint files
23 checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_")
24
25 checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
26     filepath=checkpoint_prefix,
27     monitor='loss',
28     save_weights_only=True,
29     save_best_only=True)
30
31 # Fitting the models with a validation of 20 %
32 history = model.fit(X_train, Y_train, epochs=epochs, batch_size=
   batch_size, callbacks=[checkpoint_callback], validation_split=0.2)
```

```
33
34 # Evaluate and plot the results
35 accr = model.evaluate(X_test,Y_test)
36 print('Test set\n Loss: {:.3f}\n Accuracy: {:.3f}'.format(accr
    [0],accr[1]))
37 plt.title('Loss')
38 plt.plot(history.history['loss'], label='train')
39 plt.plot(history.history['val_loss'], label='test')
40 plt.legend()
41 plt.show();
42
43 plt.title('Accuracy')
44 plt.plot(history.history['accuracy'], label='train')
45 plt.plot(history.history['val_accuracy'], label='test')
46 plt.legend()
47 plt.show();
48
49 #importing confusion matrix
50 from sklearn.metrics import confusion_matrix
51 #y_pred = model.predict(X_test)
52 y_pred=model.predict(X_test)
53 y_pred2=np.argmax(y_pred, axis=1)
54 Y_test2=np.argmax(Y_test, axis=1)
55 print('Prediction:',y_pred)
56 print('Y Test:',Y_test)
57 confusion = confusion_matrix(Y_test2, y_pred2)
58 print('Confusion Matrix\n')
59 print(confusion)
60
61 #importing accuracy_score, precision_score, recall_score, f1_score
62 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
63 print('\nAccuracy: {:.2f}\n'.format(accuracy_score(Y_test2, y_pred2))
    )
64
65 print('Micro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='micro')))
66 print('Micro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='micro')))
67 print('Micro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='micro')))
68
69 print('Macro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='macro')))
```

```

70 print('Macro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='macro')))
71 print('Macro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='macro')))
72
73 print('Weighted Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='weighted')))
74 print('Weighted Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='weighted')))
75 print('Weighted F1-score: {:.2f}'.format(f1_score(Y_test2, y_pred2,
    average='weighted')))
76
77 # Importing the classification report, confusion matrix, and plotting
    libraries
78 from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
79 import pandas as pd
80 import matplotlib.pyplot as plt
81 import seaborn as sns
82 #prediction on test data
83 pred=model.predict(X_test)
84 pred2=np.argmax(y_pred, axis=1)
85 test2=np.argmax(Y_test, axis=1)
86 print('Predictions:',pred)
87 #plot confusion matrix
88 results = {'y_Actual':test2, 'y_Predicted':pred2}
89 print('Results:',results)
90 df_results = pd.DataFrame(results, columns=['y_Actual','y_Predicted'
    ])
91 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results['
    y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
92 sns.heatmap(confusion_matrix, annot=True, fmt='d')
93 plt.show()
94 print(classification_report(test2, pred2))

```

Listing D.10 RNN code

## D.11 CNN code

Convolutional Neural Network code used in 3.2.3.

```

1 model = Sequential()

```

```
2 model.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM, input_length=X_train
   .shape[1]))
3 model.add(Conv1D(50, 3, padding='valid', activation='relu', strides=2))
4 model.add(GlobalMaxPooling1D())
5 model.add(Dense(25, activation='relu'))
6 model.add(Dense(2, activation='sigmoid'))
7 model.summary()
8
9
10 model.compile(loss='categorical_crossentropy', optimizer='adam',
   metrics=['accuracy'])
11
12 # Setting epoch and batch size - to be the same across the models for
   comparison
13 epochs = 100
14 batch_size = 128
15
16
17 # Fitting model while saving history for the plot
18 print('X:', X_train.shape)
19 print('Y:', Y_train.shape)
20 history = model.fit(X_train, Y_train, epochs=epochs, batch_size=
   batch_size, validation_split=0.1)
21
22 #save model
23 model.save('drive/MyDrive/Colab Notebooks/Masters/Data/Models/
   PythonQI_CNN_ML50_W200_ED50_N50_e100 -b128_16092022.h5')
24
25 # Evaluate and plot the results
26 accr = model.evaluate(X_test, Y_test)
27 print('Test set\n Loss: {:.3f}\n Accuracy: {:.3f}'.format(accr
   [0], accr[1]))
28 plt.title('Loss')
29 plt.plot(history.history['loss'], label='train')
30 plt.plot(history.history['val_loss'], label='test')
31 plt.legend()
32 plt.show();
33
34 plt.title('Accuracy')
35 plt.plot(history.history['accuracy'], label='train')
36 plt.plot(history.history['val_accuracy'], label='test')
37 plt.legend()
38 plt.show();
39
```

```
40
41 #importing confusion matrix
42 from sklearn.metrics import confusion_matrix
43 #y_pred = model.predict(X_test)
44 y_pred=model.predict(X_test)
45 y_pred2=np.argmax(y_pred, axis=1)
46 Y_test2=np.argmax(Y_test, axis=1)
47 print('Prediction:',y_pred)
48 print('Y Test:',Y_test)
49 confusion = confusion_matrix(Y_test2, y_pred2)
50 print('Confusion Matrix\n')
51 print(confusion)
52
53 #importing accuracy_score, precision_score, recall_score, f1_score
54 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
55 print('\nAccuracy: {:.2f}\n'.format(accuracy_score(Y_test2, y_pred2))
    )
56
57 print('Micro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='micro'))))
58 print('Micro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='micro'))))
59 print('Micro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='micro'))))
60
61 print('Macro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='macro'))))
62 print('Macro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='macro'))))
63 print('Macro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='macro'))))
64
65 print('Weighted Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='weighted'))))
66 print('Weighted Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='weighted'))))
67 print('Weighted F1-score: {:.2f}'.format(f1_score(Y_test2, y_pred2,
    average='weighted'))))
68
69 # Importing the classification report, confusion matrix, and plotting
    libraries
70 from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
```

```

71 import pandas as pd
72 import matplotlib.pyplot as plt
73 import seaborn as sns
74 #prediction on test data
75 pred=model.predict(X_test)
76 pred2=np.argmax(y_pred, axis=1)
77 test2=np.argmax(Y_test, axis=1)
78 print('Predictions:',pred)
79 #plot confusion matrix
80 results = {'y_Actual':test2, 'y_Predicted':pred2}
81 print('Results:',results)
82 df_results = pd.DataFrame(results, columns=['y_Actual','y_Predicted'
      ])
83 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results['
      y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
84 sns.heatmap(confusion_matrix, annot=True, fmt='d')
85 plt.show()
86 #print classification report
87 print(classification_report(test2, pred2))

```

Listing D.11 CNN code

## D.12 LSTM Transfer Learning code

Long-Short Term Memory Network Transfer Learning code used in 3.2.3.

```

1 #load model weights
2
3 model1=keras.models.load_model('drive/MyDrive/Colab Notebooks/Masters
      /Data/Models/Python_LSTM_ML50_ED50_N50N25_e50 -b64_06032022.h5')
4 #model2.input
5 print(model1.layers)
6 print("Original Model:")
7 print(model1.summary)
8
9 model2 = Sequential()
10 model1=keras.models.load_model('drive/MyDrive/Colab Notebooks/Masters
      /Data/Models/Python_LSTM_ML50_ED50_N50N25_e50 -b64_06032022.h5')
11 for layer in model1.layers[:-2]: # go through until last layer
12     layer.trainable = False
13     model2.add(layer)
14     print(layer)
15     print('1')

```



```
16 model2.add(LSTM(25, dropout=0.1, name='LSTM_TL'))
17 model2.add(Dense(5, activation='softmax', name='dense2'))
18 model2.compile(loss='categorical_crossentropy', optimizer='adam',
19               metrics=['accuracy'])
20 #model.compile(loss='binary_crossentropy', optimizer='adam', metrics
21               =['accuracy'])
22 # Setting epoch and batch size - to be the same across the models for
23   comparasion
24 epochs = 40
25 batch_size = 3
26
27 # Create checkpoints
28 filepath="drive/MyDrive/Colab Notebooks/Masters/Data/Weights/
29           TL_Python_LSTM_ML50_ED50_N50N25_e40 -b3_19092022_4 -10_eq.hdf5"
30 checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=2,
31                             save_best_only=True)
32 callbacks_list = [checkpoint]
33
34 # Fitting model while saving history for the plot
35 print('X:', X_train.shape)
36 print('Y:', Y_train.shape)
37 print("New Model:")
38 print(model2.summary())
39 history = model2.fit(X_train, Y_train, epochs=epochs, batch_size=
40                     batch_size, validation_split=0.1, callbacks=callbacks_list)
41 print(model2.summary())
42 #save model
43 model2.save('drive/MyDrive/Colab Notebooks/Masters/Data/Models/
44            TL_Python_LSTM_ML50_ED50_N50N25_e40 -b3_19092022_4 -10_eq.h5')
45 print(model2.summary())
46 print(model2.layers)
47 # Evaluate and plot the results
48 accr = model2.evaluate(X_test, Y_test)
49 print('Test set\n  Loss: {:.3f}\n  Accuracy: {:.3f}'.format(accr
50                   [0], accr [1]))
51 plt.title('Loss')
52 plt.plot(history.history['loss'], label='train')
53 plt.plot(history.history['val_loss'], label='test')
54 plt.legend()
55 plt.show();
56
57 plt.title('Accuracy')
58 plt.plot(history.history['accuracy'], label='train')
```

```
52 plt.plot(history.history['val_accuracy'], label='test')
53 plt.legend()
54 plt.show();
55
56
57 #importing confusion matrix
58 from sklearn.metrics import confusion_matrix
59 #y_pred = model.predict(X_test)
60 y_pred=model2.predict(X_test)
61 y_pred2=np.argmax(y_pred, axis=1)
62 Y_test2=np.argmax(Y_test, axis=1)
63 print('Prediction:',y_pred)
64 print('Y Test:',Y_test)
65 confusion = confusion_matrix(Y_test2, y_pred2)
66 print('Confusion Matrix\n')
67 print(confusion)
68
69 #importing accuracy_score, precision_score, recall_score, f1_score
70 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
71 print('\nAccuracy: {:.2f}\n'.format(accuracy_score(Y_test2, y_pred2))
    )
72
73 print('Micro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='micro')))
74 print('Micro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='micro')))
75 print('Micro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='micro')))
76
77 print('Macro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='macro')))
78 print('Macro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='macro')))
79 print('Macro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='macro')))
80
81 print('Weighted Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='weighted')))
82 print('Weighted Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='weighted')))
83 print('Weighted F1-score: {:.2f}'.format(f1_score(Y_test2, y_pred2,
    average='weighted')))
84
```

```

85 # Importing the classification report, confusion matrix, and plotting
    libraries
86 from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
87 import pandas as pd
88 import matplotlib.pyplot as plt
89 import seaborn as sns
90 #prediction on test data
91 pred=model2.predict(X_test)
92 pred2=np.argmax(y_pred, axis=1)
93 test2=np.argmax(Y_test, axis=1)
94 print('Predictions:',pred)
95 #plot confusion matrix
96 results = {'y_Actual':test2, 'y_Predicted':pred2}
97 print('Results:',results)
98 df_results = pd.DataFrame(results, columns=['y_Actual','y_Predicted'
    ])
99 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results['
    y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
100 sns.heatmap(confusion_matrix, annot=True, fmt='d')
101 plt.show()
102 #print classification report
103 print(classification_report(test2, pred2))

```

Listing D.12 LSTM Transfer Learning code

## D.13 CNN Transfer Learning code

Convolutional Neural Network Transfer Learning code used in 3.2.3.

```

1 model1=keras.models.load_model('drive/MyDrive/Colab Notebooks/Masters
    /Data/Models/TL_Python_CNN_ML50_W200_ED50_N50_e100 -b64_04092022.h5
    ')
2 #model2.input
3 print(model1.layers)
4 print("Original Model:")
5 print(model1.summary)
6
7 model2 = Sequential()
8 model1=keras.models.load_model('drive/MyDrive/Colab Notebooks/Masters
    /Data/Models/TL_Python_CNN_ML50_W200_ED50_N50_e100 -b64_04092022.h5
    ')
9 for layer in model1.layers[:-3]: # go through until last layer

```

```
10     layer.trainable = False
11     model2.add(layer)
12     print(layer)
13     print('1')
14
15 model2.add(Conv1D(25, 3, padding='valid', activation='relu', strides=2,
16     name='Conv1D2'))
17 model2.add(GlobalMaxPooling1D())
18 model2.add(Dense(5, activation='softmax', name='dense2'))
19 model2.compile(loss='categorical_crossentropy', optimizer='adam',
20     metrics=['accuracy'])
21 #model.compile(loss='binary_crossentropy', optimizer='adam', metrics
22     =['accuracy'])
23 # Setting epoch and batch size - to be the same across the models for
24     comparasion
25 epochs = 40
26 batch_size = 3
27
28 # Create checkpoints
29 filepath="drive/MyDrive/Colab Notebooks/Masters/Data/Weights/
30     TL_Python_CNN_ML50_W200_ED50_N50_e40 -b3_17092022_Graded_eq.hdf5"
31 checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=2,
32     save_best_only=True)
33 callbacks_list = [checkpoint]
34
35 # Fitting model while saving history for the plot
36 print('X:', X_train.shape)
37 print('Y:', Y_train.shape)
38 print("New Model:")
39 print(model2.summary())
40 history = model2.fit(X_train, Y_train, epochs=epochs, batch_size=
41     batch_size, validation_split=0.1, callbacks=callbacks_list)
42 print(model2.summary())
43 #save model
44 model2.save('drive/MyDrive/Colab Notebooks/Masters/Data/Models/
45     TL_Python_CNN_ML50_W200_ED50_N50_e140 -b3_17092022_Graded_eq.h5')
46 print(model2.summary())
47 print(model2.layers)
48 # Evaluate and plot the results
49 accr = model2.evaluate(X_test, Y_test)
50 print('Test set\n Loss: {:.3f}\n Accuracy: {:.3f}'.format(accr
51     [0], accr [1]))
52 plt.title('Loss')
```

```
45 plt.plot(history.history['loss'], label='train')
46 plt.plot(history.history['val_loss'], label='test')
47 plt.legend()
48 plt.show();
49
50 plt.title('Accuracy')
51 plt.plot(history.history['accuracy'], label='train')
52 plt.plot(history.history['val_accuracy'], label='test')
53 plt.legend()
54 plt.show();
55
56
57 #importing confusion matrix
58 from sklearn.metrics import confusion_matrix
59 #y_pred = model.predict(X_test)
60 y_pred=model2.predict(X_test)
61 y_pred2=np.argmax(y_pred, axis=1)
62 Y_test2=np.argmax(Y_test, axis=1)
63 print('Prediction:',y_pred)
64 print('Y Test:',Y_test)
65 confusion = confusion_matrix(Y_test2, y_pred2)
66 print('Confusion Matrix\n')
67 print(confusion)
68
69 #importing accuracy_score, precision_score, recall_score, f1_score
70 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
71 print('\nAccuracy: {:.2f}\n'.format(accuracy_score(Y_test2, y_pred2))
    )
72
73 print('Micro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='micro'))))
74 print('Micro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='micro'))))
75 print('Micro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='micro'))))
76
77 print('Macro Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='macro'))))
78 print('Macro Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='macro'))))
79 print('Macro F1-score: {:.2f}\n'.format(f1_score(Y_test2, y_pred2,
    average='macro'))))
80
```

```
81 print('Weighted Precision: {:.2f}'.format(precision_score(Y_test2,
    y_pred2, average='weighted')))
82 print('Weighted Recall: {:.2f}'.format(recall_score(Y_test2, y_pred2,
    average='weighted')))
83 print('Weighted F1-score: {:.2f}'.format(f1_score(Y_test2, y_pred2,
    average='weighted')))
84
85 # Importing the classification report, confusion matrix, and plotting
    libraries
86 from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
87 import pandas as pd
88 import matplotlib.pyplot as plt
89 import seaborn as sns
90 #prediction on test data
91 pred=model2.predict(X_test)
92 pred2=np.argmax(y_pred, axis=1)
93 test2=np.argmax(Y_test, axis=1)
94 print('Predictions:',pred)
95 #plot confusion matrix
96 results = {'y_Actual':test2, 'y_Predicted':pred2}
97 print('Results:',results)
98 df_results = pd.DataFrame(results, columns=['y_Actual','y_Predicted',
    ])
99 confusion_matrix = pd.crosstab(df_results['y_Actual'], df_results['
    y_Predicted'], rownames=['Actual'], colnames=['Predicted'])
100 sns.heatmap(confusion_matrix, annot=True, fmt='d')
101 plt.show()
102 #print classification report
103 print(classification_report(test2, pred2))
```

Listing D.13 CNN Transfer Learning code