The Use of Machine Learning to Provide an Early Indication of Programming Students in Need of Support

by

Oliver Andrew Paul Kerr

A thesis submitted in partial fulfilment for the requirements for the degree of
Doctor of Philosophy, at the University of Central Lancashire

February 2024

RESEARCH STUDENT DECLARATION FORM

Type of Award **PhD**

School **Psychology and Computer Science**

1. Concurrent registration for two or more academic awards

I declare that while registered as a candidate for the research degree, I have not been a registered candidate or enrolled student for another award of the University or other academic or professional institution.

2. Material submitted for another award

I declare that no material contained in the thesis has been used in any other submission for an academic award and is solely my own work.

3. Collaboration

Where a candidate's research programme is part of a collaborative project, the thesis must indicate in addition clearly the candidate's individual contribution and the extent of the collaboration. Please state below:

N/A

4. Use of a Proof-reader

No proof-reading service was used in the compilation of this thesis.

Signature of Candidate O.Kerr

Print name: Oliver Kerr

Abstract

Programming is a core component of any university-level computer science course. When learning to program, students' efforts can be hampered by a variety of misconceptions pertaining to fundamental programming concepts. These can range from a complete misunderstanding of a concept, to small, yet frequent mistakes that can lead to logical errors within programs. The misconceptions students hold can prevent them from developing appropriate mental models of concepts, which can ultimately create a barrier to students' learning. These misconceptions can cause issues in terms of students' understanding of the content they are being taught and can also have a detrimental impact on students' confidence. As such, it is necessary to identify students who are likely to require support with learning to program at the earliest possible opportunity. The present research, therefore, intends to establish a deeper understanding of the mental models students hold of core programming concepts prior to starting their degrees, and how they develop during the first semester of teaching within an introductory programming module. How students' mental models relate to their prior experiences and their perceived levels of confidence is also explored as part of this work, as well as how these factors link to students' performance within their programming module.

There are two distinct parts to this investigation. The first part focuses on the design and development of an aptitude test, termed the Programming Checkup, which is the main data collection mechanism for this research. The Programming Checkup was subsequently issued to students at two occasions, with the first being at the beginning of their courses and the second being towards the end of the first semester, therefore, allowing for an examination of students' progress throughout the initial stage of their introductory programming module. The second part of the investigation explores the potential for using machine learning and students' responses to the Programming Checkup at the beginning of their courses, as a means to predict students' results in their first introductory programming assessment.

The findings from the analysis conducted during this investigation indicate that there is a clear benefit to students in terms of their likelihood of holding appropriate mental models and their levels of confidence and anxiety surrounding learning to program by having prior programming experience. Likewise, having previously studied computer science also benefits students, although not as substantially as prior programming experience. It is apparent that

previously studying a mathematics-based subject after leaving school, does not benefit students in ways directly represented in their likelihood of holding appropriate mental models, nor in terms of their levels of confidence or anxiety surrounding learning to program, to the same extent as previously studying computer science or having prior programming experience. Furthermore, factors that point to some students being intrinsically motivated, wherein students intend to work in a software engineering role after they graduate or consider themselves to be "self-taught programmers", are seen to relate to higher levels of confidence and for students being more likely to hold appropriate mental models.

One of the main intentions of this investigation was to explore how students' responses to the Programming Checkup at the beginning of their course can be used to help identify students who are likely to require support with learning to program. As such, an exploration of how machine learning can be utilised to predict the results students achieve in their first introductory programming assessment was undertaken, with both classification and regression approaches being considered. The results of this evaluation found that the best performing regression model was the Random Forest Regressor, which achieved an average RMSE of 0.1686 when trained on the full training dataset, and 0.1687 when evaluated on the holdout testing dataset. This, therefore, demonstrates that the training data have not been overfitted, and that the model is capable of making predictions with a level of accuracy that is sufficient to provide an indication of a student's performance, and as such, used as a guide for identifying students who likely benefit from additional support. Similarly, the Random Forest Classifier was found to be the best performing classification model, achieving an average AUC of 0.7400 when trained on the full training dataset. However, an average AUC of only 0.6595 was achieved when evaluated on the holdout test set, thus indicating a substantial amount of overfitting, potentially due to the inherent imbalance within the dataset when a result of 50% is used as a threshold. There is, therefore, a clear need for future work to establish a more appropriate threshold, as well as to explore ways of improving the performance of both the regression and classification models. However, this investigation has demonstrated the potential of this approach, which can be improved and expanded upon within future research stemming from this work.

Acknowledgements

I would like to express my deepest appreciation to my supervisory team for their unwavering support throughout this process.

Linden, as my Director of Studies you have continuously pushed me to be at my best. From guiding my methodological approaches to fine tuning my writing, your support and encouragement has helped to make me not only a better researcher, but also a better lecturer.

Nicky, without you, none of this would have been possible. Thank you for believing in me, all the way from my first year as an undergraduate student, right through to completing my PhD. Although, I think you will always say my writing is too "flowery"!

Gareth, although you retired whilst my PhD was still at a very early stage, thank you for your support and for sharing your wealth of experience of teaching programming with me.

I am extremely grateful to all the students who chose to take part in my investigation. Having struggled with programming myself at college, helping you overcome your difficulties with learning to program has been the driving force behind my work. By taking part in my research, you are helping me to find new ways to help future cohorts of students, for which I thank you.

I must also especially thank my family for their support whilst completing my PhD. In particular, I wish to thank my mum, Allison Kerr, who has always done her upmost to allow me to achieve my ambitions. Thank you.

Finally, I wish to dedicate my work to my grandad, Iain Balmer, who died from Prostate Cancer in March 2023. He was always a huge part of my life, and I know how proud he was of my achievements. I hope I continue to make him proud in all that I do in the future.

Iain David Balmer

6th December 1947 – 11th March 2023

R.I.P.

Table of Contents

Abstract .... *I*

Acknowledgements .... *III*

List of Tables .... *VI*

List of Figures .... *IX*

List of Appendices .... *X*

List of Abbreviations .... *XI*

1. Introduction *1*

1.1 Investigation Rationale 1

1.2 Scope of Research 2

1.3 Research Questions 5

1.4 Thesis Structure 8

1.5 Research Contribution 9

2. The Issue of Learning to Program *11*

2.1 Literature Scope 11

2.2 The Difficulties of Programming 12

2.3 The Mind of a Programmer 24

2.4 Programming Cognition 31

2.5 Students' Interpretations of Programming Concepts 44

2.6 Summary 60

3. Investigation Methodology *61*

3.1 Investigation Scope 61

3.2 Potential Factors for Inclusion in the Aptitude Test 62

3.2.1 Aptitude Test Rationale 62

3.2.2 Students' Previous Experience 64

3.2.3 Students' Mental Characteristics 65

3.2.3 Working Memory Capacity and Spatial Ability 72

3.3 Predictive Model Considerations 76

3.4 Aptitude Test Design 85
3.4.1 Section Outline 85
3.4.2 Initial Aptitude Test Design 85
3.4.3 Subsequent Modifications 98
3.5 Overview of Machine Learning Algorithms 106
3.6 Summary and Methodology Reflection 123
4. Predictive Model Development........ 128
4.1 Model Objectives 128
4.2 Data Pre-Processing 131
4.3 Model Evaluation and Testing 152
4.4 Summary 159
5. Programming Checkup Analysis........ 160
5.1 Analysis Scope 160
5.2 T1 and T2 Comparison 160
5.2.1 Analysis of Students' Understandings of Core Programming Concepts 160
5.2.2 Influence of Prior Experiences on Likelihood of Holding Appropriate Mental Models 168
5.2.3 Analysis of Students' Levels of Confidence 175
5.3 Examination of Relationships with Assessment 1 Results 197
5.4 Comparison with Assessment 2 Results 230
5.5 Summary 235
6. General Discussion and Reflections of Research Outcomes and Future Work........ 236
6.1 Scope of Discussion 236
6.2 Responses to Research Questions 236
6.3 Limitations of this Investigation 251
6.4 Future Work 254
6.5 Self-Reflection and Concluding Remarks 256
References 259
Appendices 288

List of Tables

Table 2.1 *Example Schema of a House* *32*

Table 3.1 *Comparison between Learning Analytics and Educational Data Mining (derived from Siemens & Baker, 2012)* *77*

Table 4.1 *Associated Misconceptions of Each Mental Model* *133*

Table 4.2 *Chi-Squared Test Between Binarized Assessment 1 Results and Dichotomous Background Factors* *142*

Table 4.3 *Mann Whitney U Tests Between Assessment 1 Results and Dichotomous Background Factors* *143*

Table 4.4 *Mann Whitney U Tests Between Binarized Assessment 1 Results and Confidence Factors* *147*

Table 4.5 *Spearman's Rank Correlation Tests Between Assessment 1 Results and Confidence Factors* *148*

Table 4.6 *Mann Whitney U Tests between Binarized Assessment 1 Results and Mental Model Estimates Established Using Bayesian Knowledge Tracing* *150*

Table 4.7 *Spearman's Rank Tests Between Assessment 1 Results Mental Model Estimates Established Using Bayesian Knowledge Tracing* *151*

Table 4.8 *10-Fold Cross Validation Scores of Regression Models (RMSE)* *154*

Table 4.9 *10-Fold Cross Validation Scores of Classification Models (AUC)* *155*

Table 5.1 *Wilcoxon Signed Rank Comparison of Misconception Occurrences at T1 and T2* *162*

Table 5.2 *Wilcoxon Signed Rank Comparison of Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2* *166*

Table 5.3 *Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2* *170*

Table 5.4 *Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2* *171*

Table 5.5 *Comparison of Prior Programming Experience, Previously Studying Computer Science and Average Estimates of Having an Appropriate Mental Model at T1 and T2* *172*

Table 5.6 *Spearman's Rank Correlation Test Between Students' Agreement in Considering Themselves "Self-Taught Programmers" at the Start of Their Course and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2* *174*

Table 5.7 *Mann Whitney U Tests Between Previously Studying a Mathematics-Based Subject (Yes/No) and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2 ....175*
Table 5.8 *Wilcoxon Signed Rank Comparison of Confidence Factors at T1 and T2...........176*
Table 5.9 *Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Confidence Factors at T1 and T2 ....179*
Table 5.10 *Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Confidence Factors at T1 and T2 ....182*
Table 5.11 *Spearman's Rank Correlation Test Between Students' Agreement in Considering Themselves "Self-Taught Programmers" at the Start of Their Course and Confidence Factors at T1 and T2..186*
Table 5.12 *Mann Whitney U Tests Between Previously Studying a Mathematics-Based Subject (Yes/No) And Confidence Factors at T1 and T2 ....187*
Table 5.13 *Spearman's Rank Correlation Tests Between Self-Efficacy Factors and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2................189*
Table 5.14 *Spearman's Rank Correlation Tests Between Average Confidence in Answers for All Programming Questions and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2 ....190*
Table 5.15 *Mann Whitney U Tests Between Students' Intentions to Pursue a Career in Software Engineering (Yes or Undecided/No) and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2...194*
Table 5.16 *Mann Whitney U Tests Between Students' Intentions to Pursue a Career in Software Engineering (Yes or Undecided/No) And Confidence Factors at T1 and T2.........196*
Table 5.17 *Mann Whitney U Test Between Assessment 1 Results and Dichotomous Background Factors, Conducted on All Available Data at T1...198*
Table 5.18 *Mann Whitney U Tests Between Prior Programming Experience (Yes/No) And Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1...200*
Table 5.19 *Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Confidence Factors, Conducted on All Available Data at T1 ....201*
Table 5.20 *Moderation Analysis Between Prior Programming Experience (Yes/No) and Mental Model Estimates When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1...203*

Table 5.21 *Moderation Analysis Between Prior Programming Experience (Yes/No) and Confidence Factors When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1* *205*

Table 5.22 *Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) And Mental Model Estimates, Conducted on All Available Data at T1* *208*

Table 5.23 *Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Confidence Factors Conducted on All Available Data at T1* *209*

Table 5.24 *Moderation Analysis Between Previously Studying Computer Science (Yes/No) and Mental Model Estimates When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1* *210*

Table 5.25 *Moderation Analysis Between Previously Studying Computer Science (Yes/No) and Confidence Factors When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1* *212*

Table 5.26 *Spearman's Rank Correlation Tests Between Assessment 1 Results and Confidence Features, Conducted on All Available Data at T1* *215*

Table 5.27 *Mediator Analysis Conducted on Confidence Factors When Predicting Assessment 1 Results* *219*

Table 5.28 *Spearman's Rank Correlation Tests Between Assessment 1 Grades and Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1* *221*

Table 5.29 *Mediator Analysis Conducted on Mental Model Estimates When Predicting Assessment 1 Results* *222*

Table 5.30 *Mediator Analysis Conducted on Confidence Factors and Mental Model Estimates When Predicting Assessment 1 Results* *226*

Table 5.31 *Spearman's Rank Correlation Tests Between Assessment Grades and Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments* *232*

Table 5.32 *Spearman's Rank Correlation Tests Between Assessment Results and Confidence Factors, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments* *233*

Table 5.33 *Mann Whitney U Tests Between Assessment Grades and Dichotomous Background Features, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments* *234*

List of Figures

Figure 2.1 *Representation of Cognitive Load* 36
Figure 3.1 *Bayesian Knowledge Tracing Hidden Markov Model* 82
Figure 3.2 *Example of Objects Used in Mental Rotation Test* 89
Figure 3.3 *Corsi Block Test Implementation Screenshot* 91
Figure 3.4 *Example of Single Assignment Operation Question* 93
Figure 3.5 *Example of Multiple Assignment Operations Question* 94
Figure 3.6 *Example of "If" Statement Question* 95
Figure 3.7 *Example of "For" Loop Question* 96
Figure 3.8 *Example of "While" Loop Question* 96
Figure 3.9 *Example of Recursion Question* 98
Figure 3.10 *Example of Variable Assignment with Inferred Meaning Variables Question* . 100
Figure 3.11 *Example of Boolean Operator Question* 102
Figure 3.12 *Example of Addition of "Braces" to Questions* 103
Figure 4.1 *Assessment 1 Grade Distribution within the Training Dataset* 138
Figure 4.2 *Distributions of Results Relating to Students' Background Factors* 140
Figure 4.3 *Distributions of Results Relating to Students' Confidence Factors* 140
Figure 4.4 *Distributions of Results Relating to Students' Mental Model Estimates Established Using Bayesian Knowledge Tracing* 141
Figure 4.5 *Year of Birth Distribution within the Training Dataset* 144
Figure 4.6 *Random Forest Regressor Feature Importance Plots* 158
Figure 4.7 *Random Forest Classifier Feature Importance Plots* 158
Figure 5.1 *Distribution of Misconceptions and the Frequency of Occurrences Per Student (Who Completed Both Programming Checkups) at T1 and T2* 161
Figure 5.2 *Estimates of Whether Students Hold Appropriate Mental Models at T1 and T2, Established Using Bayesian Knowledge Tracing with a Threshold of 0.5* 166
Figure 5.3 *Conceptual Diagram of a Simple Mediation Model* 217
Figure 5.4 *Estimates of Whether Students Hold Appropriate Mental Models at T1 (Using All Available Data), Established Using Bayesian Knowledge Tracing with a Threshold of 0.5* 223

List of Appendices

Appendix A Final Programming Checkup Questions..*288*

Appendix B Misconceptions Examined by the Programming Diagnostic Questions Within the Programming Checkup..*326*

List of Abbreviations

AUC	Area Under the ROC Curve
BKT	Bayesian Knowledge Tracing
CART	Classification And Regression Tree
CPU	Central Processing Unit
EDM	Educational Data Mining
ENR	Elastic Net Regression
EPE	Expected Value of Prediction Error
GCSE	General Certificate of Secondary Education
IDE	Integrated Development Environment
KNN	K-Nearest Neighbor
LA	Learning Analytics
LASSO	Least Absolute Shrinkage and Selection Operator
ML	Machine Learning
MOOC	Massive Open Online Course
OLS	Ordinary Least Squares
PRIMM	Predict, Run, Investigate, Modify and Make
RBF	Radial Basis Function
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristic Curve
RQ	Research Question
RSS	Residual Sum of Squares
SMOTE	Synthetic Minority Over-Sampling Technique
SVC	Support Vector Machine Classifier
SVM	Support Vector Machine
SVR	Support Vector Machine Regressor
ZPD	Zone of Proximal Development

1. Introduction

1.1 Investigation Rationale

This research investigation is inspired by my own experiences of learning to program, which began whilst studying A-Level computer science. I remember copying lines of Visual Basic 6 code from the whiteboard and running them without really understanding what they meant or how to modify them. It took a lot of independent work for me to really develop an understanding of core programming concepts, although it wasn't until I started learning C++ at university that everything really began to "click together".

I remember looking around my first-year lab classes and seeing others struggling to get to grips with programming, in a similar way to how I had struggled during my classes at college. Consequently, when undertaking my first-year research project, I was drawn toward developing an understanding of the difficulties of learning to program as a key research theme, which continued through into my undergraduate project work and now into this research.

As a computer science lecturer, I now have the opportunity to see the process of learning to program from an educator's perspective. Now that I am teaching the first-year undergraduate introductory programming module at the University of Central Lancashire, I am required to accommodate large variances in programming abilities in students, ranging from those who have no programming experience at all, to others who have become quite advanced. Nevertheless, I often observe a significant proportion of students who struggle to comprehend core programming concepts and require additional support in order to overcome misconceptions they are holding. These experiences reinforce my rationale for investigating the difficulties students face whilst learning to program, with the eventual aim of providing a more individualised learning environment for students, in order to ensure they receive the support they need.

1.2 Scope of Research

When students are attempting to learn to program at university level, their efforts may be hampered by a variety of misconceptions pertaining to fundamental programming concepts. These misconceptions can take a wide range of forms, from a complete misunderstanding of a concept, to simple, yet frequent mistakes that will result in logical errors within their programs (i.e., the program will compile but the output may not be what the student expects). Although many of these misconceptions may appear to be trivial to experienced programmers they can be difficult for students to overcome (Sorva, 2013), creating a barrier to their learning.

The misconceptions students develop are potentially unknown to the student and are not likely to be addressed without specific intervention from teaching staff. However, as Bergin and Reilly (2005b) explain, it is common for there to be a very high student-to-lecturer ratio in university courses. Lecturers often do not know how students are performing until their first assessment, which can take place six to eight weeks into the course. By this time students' misconceptions will have become embedded and will be harder to overcome. It is, therefore, necessary to identify students who are most in need of support early on in the course to allow appropriate and timely support to be provided to tackle their misconceptions directly. This need to identify students who are likely to require support, at the earliest possible opportunity, is therefore the primary focus of this investigation.

Subsequently, there are two distinct parts to this investigation. The first part revolves around the design and development of an aptitude test, termed the Programming Checkup, which is used to collect data on a variety of different factors about each student, including their backgrounds and levels of confidence, as well as assessing their levels of understanding of core programming concepts. This is achieved by evaluating students' capacity to read, comprehend and logically deduce appropriate answers from a series of simple programming statements, which can subsequently be used to highlight any misconceptions that are preventing them from developing an accurate mental model of each concept.

The Programming Checkup is issued to students twice, once at the start of their course, prior to any teaching taking place (T1), and once at the end of the first semester; approximately 12

weeks later (T2), therefore, allowing for a comprehensive view of students' progress between the two timepoints.

For context, this investigation is being carried out within the Computing Department at the University of Central Lancashire, with first-year students who are studying one of a number of computer science related courses at undergraduate level, which include Computer Science, Software Engineering, Computer Games Development, Cyber Security and Computer Networks & Security. An additional degree named Computing is also offered, which provides students with a level of optionality in what they study during their second and third years.

All courses involve the completion of a common first-year, which includes the introductory C++ programming module at the centre of this investigation, the learning outcomes of which are:

- Appropriately apply the principles of programming to produce working programs
- Design an appropriate solution for a given problem
- Implement a readable and maintainable software solution based on their own design
- Evaluate the quality of their developed software

Students are not required to have any prior experience of programming in order to study for their degrees, therefore the first semester of teaching focuses primarily on introducing fundamental programming concepts, up to and including functions, which are reflected in their first assessment undertaken towards the end of the first semester. These concepts are then built upon during the second semester with topics up to and including object-oriented programming (OOP) being covered. The module's second assessment provides the opportunity for a wider range of programming skills to be evaluated, although both assignments do assess all four learning outcomes. It should be noted that prior to the commencement of their introductory programming module, all students undertake an intensive four week long module that exposes them to the various topics that they will encounter during their degrees (see Mitchell et al., 2013), including programming in the form of Appinventor.

The main data collection for this investigation was carried out across three academic years, commencing in September 2019. As such, the second and third years of data collection were impacted by the Covid-19 pandemic, which necessitated that teaching, and therefore, data collection with the Programming Checkup, needed to be carried out solely online during the second year of the investigation (academic year 2020-2021). Subsequently, teaching and data collection during the third year of the investigation (academic year 2021-2022) was carried out in a hybrid setting, which included both in-person and online sessions. Teaching in the latter part of the 2019-2020 academic year was also impacted by the pandemic, although this was constrained to the final weeks of the introductory programming course and students' submissions of their second assessment. Therefore, data collection with the Programming Checkup and students' first assessment were not affected during the first year of the investigation. Nevertheless, efforts were made to ensure that both teaching and data collection were as consistent as possible across all three years of the investigation.

The second part of this investigation centres around demonstrating the potential for using machine learning and students' responses to the Programming Checkup at T1, as a means of predicting the results they are likely to achieve in their first introductory programming assessment, in order to provide an indication as to whether they require additional support. Students' responses at T1 are utilised as this allows for support mechanisms to be implemented as students' progress through the course, rather than attempting to correct potentially engrained misconceptions at a later point, for example, after completing their first assessment.

This work should generally be viewed as an initial explorative investigation that examines whether it is possible to make predictions on students' performance before they have engaged with any teaching, with the aim of identifying those who are likely to require support, and upon which future studies can be based on.

Ethical approval for this research was obtained through the University of Central Lancashire's Psychology and Social Sciences (PSYSOC) ethics committee (Reference number: PSYSOC 454).

1.3 Research Questions

At the heart of this investigation, is my motivation as a practitioner-researcher to help struggling students overcome their difficulties with learning to program. As such, the investigation is guided by three research questions:

RQ 1 How do students' mental models of core programming concepts develop during a university introductory programming module?

Mental models can be defined as a mental representation of the properties and behaviours of a given concept that is based upon an individual's prior knowledge and experiences (Norman, 1983; Sorva, 2013). As such, the concept of mental models is critical within this investigation as the process of learning to program can be viewed in terms of students' development of coherent mental models that represent the actions that fundamental programming concepts perform when processed by a computer (Ben-Ari, 1998; VanDeGrift et al., 2010). Students' mental models are significantly influenced by their prior experiences and the knowledge they believe to be relevant (Ben-Ari, 2001; Sorva, 2013). However, misconceptions can be introduced if mental models are constructed upon knowledge which is in fact irrelevant or inaccurate, resulting in a mental model that is therefore inaccurate, and can create a barrier to students' learning (Sirkiä & Sorva, 2012).

In order to establish a deeper understanding of the issues students face when learning to program, it is important to examine the mental models that they possess upon commencement of their course, and subsequently, how they develop as students progress with their learning. As such, in this research, estimates are produced of how likely students are to be holding appropriate mental models of core programming concepts through students' responses to the Programming Checkup and a technique known as Bayesian Knowledge Tracing (Baker et al., 2008; Corbett & Anderson, 1994). Originally developed for use with Intelligent Tutoring Systems, Bayesian Knowledge Tracing attempts to estimate the probability of a student knowing a skill based on whether students answer questions correctly or not, while also taking into account the potential for them to slip and make a mistake or guess the answer correctly (Baker, 2020; Baker et al., 2008; Corbett & Anderson, 1994). Therefore, within the context of this investigation, Bayesian Knowledge Tracing estimates the likelihood of students holding an appropriate mental model for each concept, with answers being examined

to identify whether students have demonstrated use of an appropriate model, or whether potential misconceptions have been shown within their answers. Such an approach allows for a detailed analysis of the development of students' mental models between T1 and T2, as well as their performance within the introductory programming module.

RQ 2 Is students' perception of confidence and their previous experience positively related to their mental model development as well as their performance within their first introductory programming assessment?

As previously discussed, students' prior experiences exert a significant influence on the misconceptions that they hold, and as such, the accuracy of their mental models (Bonar & Soloway, 1985). Given that this investigation is taking place within higher education, students are likely to come from a variety of backgrounds, which could potentially be of a help or a hindrance when learning to program. Therefore, it is important to establish what relevant previous experience a student has, in order to examine the impact it has on their programming abilities. This naturally includes whether they have prior experience of programming and/or studying computer science, but also their experiences in studying mathematics-based subjects, given that experience in mathematics has been shown to aid students when learning to program (Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Wilson & Shrock, 2001). Furthermore, it is also beneficial to examine how strongly students consider themselves to be "self-taught programmers", and whether they intend to pursue a career in software engineering, as both of these factors provide an indication as to whether students are intrinsically motivated, which has previously been shown to be linked to higher levels of performance within an introductory programming module when compared to students who are extrinsically motivated (Bergin & Reilly, 2005a).

Learning to program is often a slow, complex and daunting process for students (Cheah, 2020; Guzdial, 2010; Perkins et al., 1986; Robins, 2019; Rogalski & Samurçay, 1990), particularly for those who do not have any prior experience associated with programming. Indeed, students' programming abilities can also be influenced by their perceived levels of confidence, with Rogerson and Scott (2010) claiming students' level of fear towards programming can form almost physical barriers, resulting in a loss of confidence that ultimately hampers their learning. The Programming Checkup contains a number of factors

that assess different aspects of students' confidence levels, ranging from a modified version of Ramalingam and Wiedenbeck's (1998) "Computer Programming Self-Efficacy Scale", to measures of how confident students are in their answers being correct for each question within the Programming Diagnostic portion of the Programming Checkup. Reviewing how students' previous experiences and levels of confidence relate to their programming abilities, both in terms of their mental model estimates and their results within the introductory programming module, allows for the establishment of a deeper understanding of the factors that influence students' programming abilities.

RQ 3 Can students' initial responses to the Programming Checkup be used to make predictions of students' introductory programming assessment results?

The design of the Programming Checkup was guided by the previous two research questions. The overall intention of the Programming Checkup was always the identification of students who would likely require support with learning to program. Early identification of these students enables support mechanisms to be put in place in order to allow for their misconceptions to be tackled directly before they have become engrained, thus aiding in students' mental model development and allowing them to progress through their course (Romero & Ventura, 2019).

As the T1 data collection for the Programming Checkup takes place at the commencement of students' courses and as such, prior to any teaching taking place, the results can be utilised as input for a predictive model that attempts to predict the result each student will achieve within their first introductory programming assessment, given that it assesses students' understandings of core programming concepts. This, therefore, requires an exploration of a variety of machine learning algorithms, with both regression and classification techniques being considered, which can then be built upon in future work that explores integrating the predictions into formal support mechanisms within the introductory programming module that students study.

At this stage, the aim of this research is to examine whether it is at all possible to use the results from the Programming Checkup at T1 to make predictions on students Assessment 1 results given the inherent difficulties associated with making predictions at such an early stage, due to the variety of factors that can influence students' performance (López-

Zambrano et al., 2021). Therefore, a degree of error can be tolerated when considering the performance of regression models, with the intention being for the models to be refined and improved upon in future, larger studies.

1.4 Thesis Structure

This thesis consists of six chapters, including this introduction, which has defined the scope of the investigation and provided context for the three research questions that have influenced this work. The overall thesis structure, including a summary of the subsequent chapters is provided below:

Chapter 2 – The Issue of Learning to Program provides a detailed account of the literature that has influenced the focus of this investigation. This includes discussion of topics relating to the difficulties students face when learning to program, including the misconceptions that they can develop, as well as on the cognitive impacts of learning to program.

Chapter 3 – Investigation Methodology is primarily focused on the first part of this investigation whereby the aptitude test, which becomes the Programming Checkup, is developed. This includes discussion of literature associated with the different factors that were considered for inclusion, along with the pilot studies that were used to refine the aptitude test design. A brief explanation of each of machine learning algorithms being considered for use within the predictive models, as detailed in Chapter 4, is also presented within this chapter.

Chapter 4 – Predictive Model Development describes the second part of this investigation, which is the development and testing of the models that use students' responses to the Programming Checkup to predict their first introductory programming assessment results. As such, this chapter includes a full description of how the Programming Checkup data were processed and subsequently used to train and evaluate the different classification and regression models.

Chapter 5 – Programming Checkup Analysis presents an in-depth statistical analysis of students' Programming Checkup results, which includes reviewing how students' responses change between T1 and T2, as well as how they relate to both introductory programming assessments. It should be noted that the analysis within this chapter was conducted after the model development described in Chapter 4. This was done in order to prevent the data, which had been originally isolated in order to be used as a test dataset, from influencing any of the decisions during model development.

Chapter 6 – General Discussion of Research Outcomes and Future Directions details the conclusions stemming from this work, including directly addressing the three research questions that were established at the start of this thesis. Additionally, the limitations which constrain the conclusions from this work and the future work stemming from it are also discussed. Finally, the work presented in this thesis is reflected upon within the concluding remarks and self-reflection.

1.5 Research Contribution

This investigation is firmly situated within the field of computer science education. The findings of this work can directly contribute to supporting both researchers and educators alike with understanding some of the difficulties students face whilst learning to program. Specifically, this investigation represents an original contribution to knowledge as it is the first time that students' mental models of core programming concepts, their levels of confidence and their prior experiences have been evaluated concurrently, subsequently allowing for a deepening of the understanding of how these factors relate to each other and also how students' performance within their introductory programming module develops. Furthermore, the use of Bayesian Knowledge Tracing to estimate the likelihood of students holding appropriate mental models for each of the concepts being examined within the Programming Checkup represents a unique approach to assessing students' mental models that could be adopted for use within other fields.

The intention behind this investigation has always been to identify students who are likely to require support with learning to program at the earliest possible opportunity. This investigation also seeks to demonstrate the potential for using students' responses to the Programming Checkup at T1, which takes place prior to any teaching, to predict the results they are likely to achieve in their first introductory programming assessment, therefore,

giving an indication as to whether they are likely to require support. The subsequent exploration of how this can be achieved using machine learning provides a foundation for future work to examine methods of improving the performance of the predictive models, particularly in the case of the classification model where a significant body of work could be conducted into determining an appropriate threshold within students' assessment results from both a machine learning and a pedagogic perspective. Furthermore, this investigation should be viewed as a starting point that enables future research into the issues students face when learning to program and the interventions that can be put in place to support them.

The contributions from this investigation align with the intention to publish within the field of computer science education. As such, publications will be developed with specific foci on students' mental models of core programming concepts, the impact of students' confidence levels on their progression within their introductory programming module and predicting students' assessment results using data from the Programming Checkup. Additionally, publications focusing on pedagogic interventions will form a core part of future work stemming from this investigation.

2. The Issue of Learning to Program

2.1 Literature Scope

This chapter intends to provide an account of the literature that has influenced the focus of this investigation through the three previously stated research questions. The reasons as to why programming is seen to be such a difficult topic for students to learn will be discussed as a means of providing context for further consideration regarding the cognitive impacts of learning to program and the issues students face when attempting to comprehend fundamental programming concepts.

Specifically, Section 2.2 provides context to this investigation by exploring why learning to program is generally a difficult task for students. Subsequently, Section 2.3 focuses on exploring the concept of computational thinking and approaches to teaching and learning programming, with discussions surrounding the cognitive factors that can impact on students' capacity to comprehend the material being taught being presented within Section 2.4. Finally, Section 2.5 introduces the concept of "mental models" and provides discussion pertaining to how students can misinterpret fundamental programming concepts. As such, this chapter aims to contextualise the motivations of this research by providing a detailed understanding of the process of learning to program and subsequently highlight the difficulties that students face.

The literature discussed in this chapter comprises a wide range of sources collected using the University of Central Lancashire Library Search, which is powered by the ExLibris database, and supplemented with the use of Google Scholar and the ACM Digital Library. The preference was to include literature that has been published more recently. However, it is important to acknowledge that a significant amount of work surrounding the difficulties of learning to program was conducted pre-2000 and is still heavily cited. This work has also been included within this section as although technologies have developed significantly over the years, the core principles of teaching and learning programming have remained the same.

2.2 The Difficulties of Programming

Exploring the difficulties attached to learning to program is crucial to the construction of this work and has been a widely deliberated topic in computer science for decades. Stemming back as far as the 1970s and 80s, researchers have been attempting to develop an understanding of the difficulties faced by anyone wanting to learn to program (Capstick et al., 1975). The broad nature of programming has encouraged a variety of research approaches, from investigating relationships between programming abilities and cognitive styles (the methodology a student uses to solve problems, i.e. analytical or heuristic based approaches; Cheney, 1980), to examining how students' understanding of specific programming concepts (such as iteration or recursion), can influence their capacity to understand other related concepts. For example, Wiedenbeck (1989) examined the relationships between the understanding of iteration and recursion. Subsequently, as computers have evolved over time, so have the research methods, with newer investigations utilising various data mining and artificial intelligence techniques (Al-Radaideh et al., 2006; Blikstein et al., 2014; Watson et al., 2013) in an attempt to gain insight into the factors that influence programming abilities.

An initial analysis by Konecki and Petrlic (2014) of literature relating to the problems faced by novice programmers revealed that the general consensus amongst both students and teachers is that programming is a difficult topic to learn. To that end, a large, multi-institutional study conducted by the Innovation and Technology in Computer Science Education (ITiCSE) 2001 working group (McCracken et al., 2001) revealed that many students are still unable to program after the conclusion of their introductory course. They go on to discuss that many of the programming solutions that students provided would not compile due to syntax errors, suggesting that students have not even acquired the skills needed to make a program during their course. This observation aligns with original motivation for conducting this research.

Although the study conducted by McCracken et al. (2001) highlights a range of issues facing students who are learning to program, their study does not attempt to identify any causes of these difficulties. Subsequently, a later ITiCSE working group (Lister et al., 2004) explored the issues identified by McCracken et al. (2001) through an additional multi-institutional study, in an attempt to identify an explanation of why students struggle to program.

The premise of Lister et al.'s (2004) study focuses around providing an alternative to the belief that a lack of problem-solving abilities is responsible for students' difficulties. Lister et al. (2004) believe this to be a popular explanation, although no empirical evidence is provided to support this claim, My own personal experiences of teaching programming at university-level does, however, points towards this as a cause.

Like the study conducted by McCracken et al. (2001), Lister et al. (2004) undertook a multi-institutional investigation in which students were tasked with completing a series of multiple-choice questions. These questions tested students' ability to predict the outcome of executing a short piece of code, as well as their capacity to complete a short piece of near-complete code by selecting from a number of possibilities. Lister et al. (2004) also conducted interviews with the students and analysed their "doodles" on scratch paper to gain further insight into how they arrived at their answers.

It was determined by Lister et al. (2004) that many of the students who took part in their study had manifested a fragile ability to systematically analyse a short piece of code and as such, lacked the capacity to read and comprehend code. These claims were supported by interviews conducted with students who had scored poorly in the test as they showed the students had significant difficulties in correctly evaluating the code, with some students even admitting to guessing.

Lister et al. (2004) noted that some students who scored well on their test but struggled to write code of a similar complexity are likely suffering from a weakness in problem-solving ability, as the ability to accurately read and comprehend code is a precursor to developing the ability to devise appropriate solutions to problems. They therefore suggest that any future studies wishing to investigate the problem-solving abilities of novice programmers should include a mechanism to identify students who face difficulties with reading and comprehending code.

Lister et al. (2004) also acknowledge the fact that as this is a multi-institutional study, it is inevitable that there will be some differences amongst the participating students. For example, it was noted that students' programming abilities varied by institutions – potentially because of differing entry requirements. It was also acknowledged that the questions had to be translated from Java to C++ and indention styles changed to account for what students had

been taught in their classes. Furthermore, although the majority of students completed the test towards the end of the first semester, it was found that some students completed the test at an earlier point in the semester or as late as the third semester, thus having differing levels of experience. Differences in how the test was carried out were also identified, with one researcher creating multiple versions of the test to prevent copying and another issuing the test on a computer whilst the rest of the tests were issued on paper. Additionally, students' motivations for taking part in the test also differed, with some students taking part voluntarily, whilst others were required to complete the test as it contributed to their course results.

To improve the generalisability of future studies, greater experimental control of factors would be desirable, for example, by using pseudocode to make the test language-independent and making the participant recruitment and testing process as identical as possible to support any claims arising from the study. It is also worth noting that over half of Lister et al.'s (2004) data was contributed by a single university. Lister et al. (2004) acknowledge this fact and state that after conducting a statistical analysis it was determined that the uneven sampling did influence the results of the investigation but did not dominate their findings. However, further studies should be carried out to support their claims.

The initial claims that a lack of problem-solving skill can hamper programming ability were also examined by Gomes et al. (2006) who conducted an investigation into how students' programming abilities are influenced by mathematical and problem-solving abilities. Gomes et al.'s (2006) propose that a lack of problem-solving abilities, specifically those that involve mathematical and logical knowledge, significantly contribute to the difficulties students face whilst learning to program. To investigate this belief, their investigation focused on around 33 students who had failed their first programming course and were showing severe difficulties in getting to grips with basic programming concepts. As part of the investigation students were enrolled in a course designed to improve their mathematical and logical knowledge. During the course, students attended sessions where they were required to complete different mathematics and logic-based exercises, as well as a number of problem-solving tasks in a variety of contexts, with a particular focus on tasks relating to programming. Students were also supported by mathematics and computer science teachers who were able to address problems and introduce mathematical concepts where necessary after each task-based session.

Although Gomes et al. (2004) did not present results for how students' programming abilities had progressed by the time they had finished the course, they did identify that students exhibited a significant lack of mathematical skills, which was reflected in limited problem-solving abilities and, therefore, poor programming abilities. This is an important consideration that must be kept in mind within the design of the study for this research. It will be interesting to determine whether previously studying mathematics has a significant relationship with students' success within an introductory programming module.

McCracken et al. (2001) defined problem solving as a five-step process:

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose
5. Evaluate and iterate

Despite the limited sample size of Gomes et al.'s (2004) study, a number of factors, which they propose contributed to students' lack of problem-solving abilities, were identified, some of which appear to correlate with McCracken et al.'s (2001) definition of the problem-solving process. These factors were mostly related to students' abstraction abilities, which involve developing an understanding of a problem, breaking it down into smaller chunks and then making logical deductions to develop an appropriate solution.

It is apparent from these three studies that more work is needed to validate the relationship between programming and problem-solving abilities. More research is also required into the factors that contribute to problem-solving abilities, including the ability to accurately read and comprehend code, and engage in abstraction (a key component of computational thinking - which is discussed in more detail in the next section).

An interesting line of research has also been posed by Lowe (2019) who re-examined the work conducted by Lister et al. (2004) using Dual Process Theory. According to Lowe (2019), the minds of novice programmers can often seem forgetful, irrational, and sometimes paradoxical, with students appearing to be on the right path and then making an irrational mistake. Dual Process Theory offers a potential explanation for how decisions are made by separating mental processing in to two mechanisms, System 1 and System 2 (sometimes

referred to as Implicit and Explicit). System 1 can be described as universal cognition and includes instinctive behaviours and complex, yet mundane tasks such as reading and interpreting text. Whereas System 2 handles more logical, focused thinking (Evans, 2003; Lowe, 2019). Although Lowe's (2019) initial study only examines data from two of Lister et al.'s questions, they have potentially identified a new method of examining the thought processes of novice programmers, which may prove useful in the design of future research into programming pedagogy.

Although a lack of problem-solving ability has been suggested as one potential cause of difficulties in learning to program, the broad nature of programming provides several different areas that could be potentially troublesome for students. Du Boulay (1986) suggests that there are five key areas of difficulty within programming:

1. General Orientation – students must develop an understanding of what programming is for, the kinds of problems that can be tackled by it and the advantages of learning it.
2. Notional Machine – students can struggle to realise how a computer executes the instructions in a program through a lack of understanding of the *notional machine*. A notional machine represents the general properties of the machine the student is learning to control and as such, is a characterisation of the computer in its role as an executor of programs in a particular language (Sorva, 2013).
3. Notation – students may experience problems with learning the syntax and semantics of a particular language. The semantics of a language can be viewed as an elaboration of the properties and behaviour of the notional machine.
4. Structures – students may face difficulties in applying the notation of a language when attempting to apply or adapt known schemas and plans to suit the requirements of a program, for example, adapting a loop to compute a numerical sum.
5. Pragmatics – students must learn to apply their knowledge of programming to specify, develop, test and debug a program. This not only requires an understanding of how to write a program, but also how to identify and solve problems effectively.

These five areas cannot be fully separated from each other and as such, students are often overwhelmed during their first encounters with programming as they attempt to try and comprehend all of the different issues at once (Du Boulay, 1986). In this research, this feeling of overwhelmingness will be referred to as "programming shock". Despite the fact that Du

Boulay's (1986) analysis of novice programmers was conducted over 30 years ago, during which time technology has advanced significantly, the principles that Du Boulay discussed are still highly relevant to modern-day programming and, result in Du Boulay's (1986) work still being heavily cited within the Computing Education research field.

Robins (2019) describes programming languages as "complex artificial constructs", which, like natural language, "consist of a relatively small number of elements that can be combined in infinitely many productive ways (p. 327). Consequently, the process of learning to program is often referred to as being slow and complex (Guzdial, 2010; Robins, 2019; Rogalski & Samurçay, 1990). Rogalski and Samurçay (1990) suggested that the complexities of learning to program stem from the fact it relies on a variety of cognitive activities, with students being required to develop accurate and reliable mental representations of the processes carried out during the development of a program, as well of basic programming concepts such as variables, loops and conditional statements that are, in effect, the building blocks students utilise to solve problems. Even the most basic of programming concepts is abstract in nature, with no real-world counterpart (Guzdial, 2010; Khalife, 2006), which consequently makes understanding and applying them appropriately an area of difficulty for students (Cheah, 2020; Guzdial, 2010; Lahtinen et al., 2005; Luxton-Reilly, 2016; Luxton-Reilly et al., 2018; Robins, 2019). Indeed, students' knowledge is often limited to a surface level, 'line by line' view of programs, resulting in students often struggling to identify where it is appropriate to use a particular concept, even if they have a general understanding of how it works (Lahtinen et al., 2005; Perkins et al., 1986).

In an attempt to gain a more detailed understanding into why students find even the most basic programming concepts difficult to master, Berglund and Lister (2010) utilised Kansanen and Meri's didactic triangle (1999) to analyse the interactions between students, teachers and content in an introductory programming scenario. A didactic triangle is a technique used to illustrate the interactions between components within the teaching-studying-learning process (Kansanen & Meri, 1999). This was applied by Berglund and Lister (2010) to gain an improved awareness of the issues that they believe are often taken for granted or left implicit when teaching and learning programming. During their investigation Berglund and Lister (2010) revealed that there is a strong disconnect between teachers and students, with teachers tending to base their lessons on their own understanding of a particular concept rather than on how it should be taught effectively.

Berglund and Lister (2010) demonstrate this latter point with an example from Bruce (2005), who revealed that teachers who are familiar with teaching procedural programming, as opposed to object-oriented programming, tend to teach an object-oriented based course much the same as they would teach a procedural programming course, and only include object-oriented topics when required too. Although Bruce's comments are derived from an analysis of the Special Interest Group on Computer Science Education (SIGCSE) mailing list discussion and have little empirical evidence to support them, they do demonstrate teachers' viewpoints and support Berglund and Lister's view that teachers base lessons on their own needs instead of that of the students.

Berglund and Lister (2010) go on to add that programming is a difficult topic to both teach and learn and that educators know very little about their students' world and motivations. To make the teaching more accessible to students, educators must adapt their methods to meet students' viewpoints. Berglund and Lister's (2010) work highlights the need for more comprehensive research into the different factors that influence a student's view of basic programming concepts, and programming as a whole, as well as into how teaching methods can be modified to meet students' needs. Furthermore, it is the motivating factor behind the research questions that form the focus of this investigation, as enabling educators to identify students who are likely to require support, as well as developing an understanding as to how students are attempting to understand programming concepts, will enable them to provide more direct support to students.

Additionally, Lister (2011) examines the processes through which students learn to comprehend and reason about code from a neo-Piagetian perspective, which is centred around the belief that regardless of age, people progress towards increasingly abstract forms of reasoning as they gain more domain experience (Lister, 2011; Teague & Lister, 2014a). This is opposed to the traditional Piagetian perspective which focuses on the development of children, whereby increasingly forms of abstract reasoning becomes possible as their brains develop (Lister, 2011; Teague & Lister, 2014a). The four stages of cognitive development within novice programmers; which progress from least mature to most, are defined as (Lister, 2011; Teague et al., 2013; Teague & Lister, 2014a, 2014b, 2014c):

1. **Sensorimotor** – Students in this stage cannot reliably trace code in order to establish the final values of variables due to the misconceptions that they hold. Their domain knowledge is limited and fragile, and the focus of the student is on learning the syntax of the language, thus making tracing code a task which requires significant cognitive effort. Students in this stage also often work through trial and error.
2. **Preoperational** – By this stage, students are able to trace code reliably but struggle to reason about it. Misconceptions may still be present, with students being generally unable to see how different pieces of code fit together to produce a solution as a whole. Students at this stage may eventually produce a correct solution, but significant effort would be required in order to do so.
3. **Concrete Operational** – Students at the Concrete Operational stage are starting to reason at a more abstract level. They can understand short pieces of code simply by reading them, without the need to manually trace through their operations, and can now, for the first time, comprehend both the whole solution, and the individual parts at the same time.
4. **Formal Operational** – At the Formal Operation stage, students are able to reliably and efficiently produce solutions to solve problems by carrying out each of the previously discussed 'problem solving steps' (McCracken et al., 2001).

A series of think aloud studies were conducted by Teague et al. (Teague et al., 2013; Teague & Lister, 2014a, 2014b, 2014c) which provide support for the validity of these four developmental categories. Similar to Berglund and Lister's (2010) recommendation to modifying teaching methods to meet students' needs, there is also a clear need to explicitly consider students' reasoning abilities within the design of introductory programming modules (Lister, 2011; Luxton-Reilly et al., 2018; Teague et al., 2013; Teague & Lister, 2014a, 2014b, 2014c). However, it would be beneficial for future studies to be conducted with larger numbers of participants in order to allow for the exploration of the rate at which students progress between each of the categories while learning to program, and what issues can prevent them from making progress (Luxton-Reilly et al., 2018).

As has been mentioned previously, learning to program is a slow and complex task (Guzdial, 2010; Rogalski & Samurçay, 1990) with Winslow (1996) suggesting it takes approximately 10 years to turn a novice programmer into an expert, a view supported by my own personal experiences in learning and later teaching programming. Consequently, a three-year

undergraduate course can only provide a platform for students to develop their programming abilities from.

Naturally, the conversion from novice to expert has several intermediate steps. Winslow (1996) cites a commonly referenced scale from Dreyfus and Dreyfus (1986), a republished version of which (Dreyfus, 2004) has been cited over 700 times, that breaks down the novice/expert continuum into five stages:

1. **Novice**: Learns objective facts, features and rules for determining actions – everything they do is context free.
2. **Advanced Beginner**: Starts to recognise and handle situations not covered by given facts, features and rules (context sensitive) without quite understanding what they are doing.
3. **Competence**: After considering the whole situation, consciously chooses an organised plan for achieving the goal.
4. **Proficiency**: No longer has to consciously reason through all the steps to determine a plan.
5. **Expert**: Generally knows what to do based upon mature and practiced understanding.

It is hoped that by the end of an undergraduate degree students should be ranked between competent and proficient (Winslow, 1996). However, with large portions of students being unable to produce working programs at the end of their introductory programming modules (Konecki & Petrlic, 2014), the aim of having the majority of graduates being classed as at least competent programmers seems optimistic at best. Bruce et al. (2004) investigated in great detail the processes that students go through whilst learning to program. Through interviews conducted with students, Bruce et al.'s (2004) study revealed that students can go about learning to program in any of five different ways:

1. **Following:** Students who are classed as "following" are generally only interested in keeping up with set assignments. Their interests are only focused on where there are marks to be gained, and they often exhibit frustration if the course material does not match their expectations. Their interests are limited to what is needed to pass the module and they do not reflect on programming in a broader context.

2. **Coding:** Students view the act of learning to program as learning to code specifically. Such students focus on learning the syntax of a language as being central to learning to program. They are driven by the belief that they must learn to code in order to program. Due to the amount of syntax needing to be learned students often get frustrated and see taking time to explore concepts and discovering their own solutions as being a waste.
3. **Understanding and Integrating:** In this category students view the act of learning to program as learning to understand and integrate the concepts involved. Students who go about learning to program in this way are seeking to develop an understanding of the “bigger picture” of programming. Typing in the code and seeing if it works is not enough for them; these students seek to understand what they have done in order to affect the particular outcome. In some cases, students view learning to program as building on prior experience, with concepts being learned sequentially. Each concept is viewed as a “building block” which must be mastered before moving on to the next one; a student may spend a significant amount of time on a single concept they are struggling to understand and will only move on once they have mastered it, or when introduced to an additional concept they feel is more important. It is possible that students may adopt a trial and error approach to writing their programs, making experimentation an important part of their learning process. Students in this category focus less on the code itself, but more on using code as a means to achieve an understanding of concepts. They are motivated by their desire for insight and are consciously aware of the ‘bigger picture’ of programming outside of their module assignments.
4. **Problem Solving:** Students in this category experience learning to program as learning what it takes to solve problems. As in the previous category, students are conscious of the ‘bigger picture’ of how the programming skills they are learning relate to the problems they are attempting to solve. For students in this category the problem is always the starting point and although coding is an important part of the learning process, it is not the main focus. Students in this category do tend to want to ‘jump in’ and start coding in an attempt to solve the problem and can also be inspired to solve problems that have not been set as part of the assignment.
5. **Participating:** Students in this category are learning what it means to be part of a programming community. Students are no longer focusing on learning the syntax and semantics; instead, students are learning how to think like a professional programmer,

> as well as investigating the different aspects of working in the software engineering industry. Students also become much more aware of programming culture – for example, evaluating the readability of a program as being as important as the program's features.

Bruce et al. (2004) go on to state that students who do not move past Categories 1 (following) or 2 (coding) are less likely to achieve the learning outcomes of their programming course. These students have adapted a surface orientation and are, therefore, not seeking meaning in what they are doing and are merely learning the answers to questions or strings of code needed to complete tasks. As a result of their surface orientation students may struggle to apply the concepts they have 'learned' in later programming tasks.

In contrast, students who experience learning to program as in Categories 3 to 5 are adopting a deeper orientation to the subject – they are seeing the meaning in what they are doing and can place it within the 'bigger picture', therefore providing themselves with much firmer foundations for learning more advanced concepts in later programming classes.

Bruce et al. (2004) note that the range of categories highlights a distinction between students who focus on 'parts' as opposed to those who focus on 'wholes' in their learning experience and as such, teaching and assessment strategies must be adapted to ensure students of all categories remained engaged with the course. For example, students from Categories 1 or 2 could be classed as focusing on 'parts' of the subject, as Category 1 students tend to have a desire for information to be presented to them in small amounts and make little or no attempt to place learning into the boarder context of programming. Similarly, students in Category 2 focus purely on the syntax of the language they are trying to learn, often to the detriment of their understanding of the underlying programming concepts. Teachers may be able to encourage students to begin to see the bigger picture by prompting them to refocus by explicitly emphasising the broader context of what the students are learning and programming generally. In terms of assessment, Bruce et al. (2004) suggest that smaller, more frequent assignments may help students who see learning to program as a means of getting through the course as it increases the opportunities for the students to receive feedback and for teachers to ensure they are on the right track.

Alternatively, students who focus on 'wholes' whilst learning to program, as in Categories 3 and 4, expect staff to provide context and ways to help them develop a sense of understanding of how the topics they are learning relate to the wider programming world. Consequently, students in these categories tend to prefer assignments which build on previous ones, as students' motivation may increase if they are given problems they perceive as having relevance to them (Bruce et al, 2004).

If it is the goal of the class to produce students who approach programming in the way that Category 3 describes, then this must be explicitly incorporated into the teaching strategy. Students cannot be expected to embrace new ways of viewing the programming world if they are not being introduced to it by their teachers. These comments echo those of Berglund and Lister (2010), who noted that teachers must adapt their material to meet the views of the student. Although the studies by Bruce et al. (2004) and Berglund and Lister (2010) are both well cited, there still remains a significant amount of work to determine how well these modifications to teaching strategies actually benefit students who are struggling to learn to program. Bruce et al. (2004) also acknowledge that whilst their categories are discrete, students may adopt different ways of experiencing learning to program at various points during their introductory programming module, therefore, a more in-depth study should be carried out across a larger sample of students to identify how they progress through an introductory programming module and what factors influence their approaches to learning. Nevertheless, the categories described by Bruce et al. (2004) speak to the motivational levels of students and as such, may influence students' performance within their introductory programming module.

2.3 The Mind of a Programmer

There is a common misconception that computer science is predominately about programming (Lu & Fletcher, 2009) however, an important factor in both learning to program and developing an understanding of computer science as a whole is the concept of *computational thinking*.

A popular definition of computational thinking comes from a paper by Jeannette Wing (2006), which has been cited over 10,000 times (Google Scholar), and describes computational thinking as "taking an approach to solving problems, designing systems and understanding human behaviour that draws on concepts fundamental to computing". Further to Wing's definition, Lue and Fletcher (2009) provide an overview of the main components of computational thinking:

- It is a method of solving problems and designing systems that draws on fundamental computer science concepts.
- Different levels of abstraction are used to understand and solve problems more efficiently.
- It involves thinking algorithmically and applying mathematical conceptions to develop more efficient, fair and secure solutions.
- An understanding of the consequence of scale must be developed, not only in terms of efficiency, but also for economic and social reasons.

It is important to note that computational thinking is not about getting people to think like computers, rather it is about encouraging students to develop a full set of mental tools necessary to effectively use computers to solve complex human problems (Lu & Fletcher, 2009; Reges, 2008; Wing, 2006).

Rogalski and Samurçay (1990) highlighted the fact that acquiring and developing programming knowledge is a highly complex process, as students must not only learn to code (i.e., by understanding and applying the syntax and semantics of a language appropriately), but they must also develop the skills and thought processes needed to establish the requirements of a program and to devise appropriate solutions. In essence, students must develop their computational thinking abilities concurrently with learning to code.

Students' exposure to topics such as problem solving through iteration (Brennan & Resnick, 2012) and abstraction (Wing, 2008) whilst learning to code help to form the building blocks of their computational thinking abilities, which in turn enable students to improve their problem solving and develop more succinct solutions (Wing, 2006; Wing, 2008). However, the applications of computational thinking are not limited to the realms of computer science and programming alone. Wing (2008) believes that computational thinking is for everyone, everywhere, as it encourages new ways of tackling problems which would not be possible without a computer. As such, computational thinking has found increasing relevance in subjects outside of computer science including mathematics, science and engineering (Hambrusch, Hoffmann, Korb, Haugan, & Hosking, 2009; Weintrop et al., 2016; Wing, 2008).

The importance of computational thinking within the computer science syllabus itself has also been acknowledged by a number of exam boards (AQA, 2020; OCR, 2020; Pearson, 2020) whose GCSE specifications explicitly examine students' Computational Thinking abilities. Although the exact content of introductory programming classes at undergraduate level varies by university, they must all nevertheless address some of the key components of computational thinking whilst also introducing students to basic programming concepts such as variables, selection and iteration.

The range of content needing to be addressed within introductory programming classes, combined with the fact that students with varying levels of experience must be catered for, makes designing effective content difficult for educators. This is reflected in the relatively high failure rates in introductory programming modules which were revealed in a study conducted by Bennedsen and Caspersen (2007) who estimated that on average 33% of students fail their introductory programming modules. Although the initial study by Bennedsen and Caspersen (2007) had a limited sample size, their initial results were later confirmed by Watson and Li (2014) and then later revised in a significantly larger study to 28% of students failing (Bennedsen & Caspersen, 2019). These figures are substantial enough to indicate that there is still a clear requirement for improvement within introductory programming modules and provides support for the importance of adapting materials to meet the needs and experiences of students, as Berglund and Lister (2010) suggest.

One potential method for improving student performance in their introductory programming modules could be the adoption of the constructivism teaching methodology (Ben-Ari, 2001; Jones & Brader-Araje, 2002; Piaget, 1973; Vygotsky, 1962). Constructivism differs from traditional teaching approaches in which knowledge is transferred to students in a continuous process in the form of lectures, textbooks, etc. Rather, constructivism-based teaching is centred around students actively constructing their own understanding of a concept. In doing so, students create a *cognitive* model, which is a combination of their pre-existing *domain knowledge* and the knowledge that they have gained through applying the concept being learned (Ben-Ari, 2001; Gonzalez, 2004; Jones & Brader-Araje, 2002; Piaget, 1973; Vygotsky, 1962; Yadin, 2012).

There are said to be two dominant constructivism philosophies: Piaget's personal (individual) constructivism and Vygotsky's social constructivism (Phillips 2000, as cited in Amineh & Asl, 2015). Piaget's philosophy is based around active participation, whereby students pass through successive stages that allow for an increasingly accurate understanding of reality (Piaget, 1973) to be established. As such, there may be stages where students accept an idea, but then may come to change it or reject it entirely at a later stage (Amineh & Asl, 2015; Piaget, 1973), therefore, students develop their understanding of a topic through active participation, meaning learning cannot occur passively (Amineh & Asl, 2015).

Vygotsky's social constructivism philosophy, on the other hand, is centred around the belief that cognitive growth occurs first at a social level and then later at an individual level (Amineh & Asl, 2015; Vygotsky, 1978). Learning is still viewed as an active process but is done so in coordination with other people (Amineh & Asl, 2015). Leeds-Hurwitz (2009) suggests that the two most important elements in social constructivism are the assumption that humans rationalise their experiences by creating a model of the social world and the way it functions, as well as the belief that language is the essential system through which humans construct reality. Learning, according to Vygotsky (1978) is a process of continual movement from a student's current intellectual level, to a higher one which is closer to their potential. Subsequently, this movement occurs within the Zone of Proximal Development (ZPD) (Amineh & Asl, 2015; Shabani et al., 2010; Vygotsky, 1978) which is defined by Vygotsky as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" (Vygotsky, 1978,

p. 86). Therefore, the ZPD represents a key component of social constructivism as it describes a student's current level development, as well as the next level that the student can achieve when appropriate support, in the form of the presence of a more skilled, or knowledgeable, person (Amineh & Asl, 2015; Shabani et al., 2010; Vygotsky, 1978). This highlights the key belief behind social constructivism that interactions, whether it be with a teacher or a fellow student, supports learning (Shabani et al., 2010).

Despite the clear differences between the philosophies presented by Piaget and Vygotsky, a commonality amongst them both, as well as amongst the many other variations of constructivism, is that students take an active role in their learning (Amineh & Asl, 2015; Biggs, 2014; Shephard, 2019). Within the setting of higher education, a key constructivism-based approach which has widespread use is *constructive alignment* (Biggs, 2014). This is a outcome-based approach to teaching whereby the intended learning outcomes are defined prior to any teaching taking place. The intended learning outcomes are then used to design appropriate teaching and learning strategies to enable students to achieve the outcomes and assess how well they have been met (Biggs, 1996, 2014; Shephard, 2019). Biggs (1999) notes that two variables that influence learning are the level of engagement by students, and the extent to which the teacher requires students to be actively involved in the teaching process. As such, Biggs' work provides further grounds to support and encourage students taking an active role in their learning.

Ben-Ari's (2001) widely cited paper 'Constructivism in Computer Science Education' provides one of the earliest discussions surround constructivism within computer science education. The paper presents a comprehensive comparison between traditional and constructivist teaching methodologies, which is centred around the following definition of the four key components of an education paradigm from Ernest (1995, as cited in Ben-Ari, 2001):

- An *ontology,* which is a theory of existence.
- An *epistemology,* which is a theory of knowledge, referring to both the specific knowledge of an individual and to shared human knowledge.
- A *methodology* for acquiring and validating knowledge.
- A *pedagogy,* which is a theory of teaching.

Ben-Ari uses these four components to describe the traditional educational paradigm culminating in the following points:

- An ontological reality *does* exist. Scientific theories of relativity, quantum mechanics and the Newtonian model of absolute space and time are generally used to represent reality.
- Epistemology is *foundational*. The truth is out there to be uncovered. Necessary truths, such as 2 + 2 = 4, are accepted and are combined with valid forms of logical deduction to expand the extent of true knowledge.
- Minds are a *clean slate*, which can be filled with knowledge. Once enough facts and rules of inference are understood, new knowledge can be created through logical deduction.
- The primary means of *knowledge transmission* is listening to lectures and reading books. Repetition ensures that knowledge is retained.

Ben-Ari goes on to describe the constructivist paradigm which he believes to be "dramatically different" to that of the traditional educational paradigm.

- The ontological reality is either completely rejected or at least considered irrelevant as you can never truly 'know' something, therefore, ontologies do not influence the constructivist paradigm.
- Constructivism's epistemology is *nonfoundational* and *fallible.* Absolute truth is unattainable and therefore there are no foundations of truth to build upon. Knowledge is constructed by each individual and is therefore fallible.
- Knowledge is acquired *recursively*, with sensory data being combined with existing knowledge to create new cognitive structures, which then form the foundation for further construction. Knowledge can also be created through reflection on existing knowledge.
- Passive learning is likely to fail as each student brings a different cognitive frame to the classroom and as such, will construct knowledge differently. Learning must therefore be active under guidance from the teacher and with feedback from fellow students. As Winslow (1996) stated, learning concepts and techniques of a new language requires writing programs in that language. The constructivism paradigm is

centred on the belief that effective learning requires not only the discovery of facts, but also the construction of viable mental models (discussed in detail in the subsequent section).

Ben-Ari (2001) also notes that as constructivism builds recursively on knowledge already held by the student, the result is an idiosyncratic version of knowledge that may differ from "standard scientific knowledge". In such cases the student is said to have developed a *misconception* (programming misconceptions are discussed in Section 2.4). It is Ben-Ari's belief that teaching techniques derived from constructivism may be more successful than traditional techniques as they explicitly address the process of knowledge construction, thus placing greater emphasis on addressing the misconceptions students develop. Indeed, I believe Ben-Ari's (2001) suggestions of explicitly teaching models of underlying constructs is particularly important in supporting students' understandings of concepts when learning to program.

Learning how to appropriately use and apply a programming language can require a student to revaluate their understanding of the language they are attempting to learn, and also their understanding of the computer system as a whole (Pea & Kurland, 1984). Discussions surrounding constructivism have highlighted the need for students to play an active role within their learning, as the teaching of even the simplest of concepts, such as variables, can be surprisingly complex (Hill & Guzdial, 2019). As Ben-Ari (2001) suggests, there is a need for clear and direct instruction of models of fundamental programming concepts (Ben-Ari, 2001), which is supported by Hill & Guzdial's (2019) belief that direct teaching should focus on plans (groups of statements as opposed to single lines of code), with worked examples being used to aid comprehension and understanding. However, Pea and Kurland (1984) believe that students are unlikely to experience the complex cognitive changes required to develop understandings of programming conceptions either through spontaneous exploring or explicit instruction alone, as they must be engaged with a task in order to interpret new concepts.

Whilst the specific ways in which introductory programming modules are taught are not the focus of this investigation, they are relevant to how the outcomes of this work are intended to be implemented. The improved understanding of students' mental models which RQ 1 aims to establish, would directly support a teaching strategy informed by Lui et al.'s (2004)

guidelines, which encourage the clear instruction of mental models of core concepts. Furthermore, students deemed likely to struggle within their introductory programming course, as per RQ 3, could be supported to take an active role within their learning, through constructivism-based techniques. Possible techniques include physical computing (Brehm et al., 2019; Przybylla & Romeike, 2014), program visualisation (Bakar et al., 2019; Moons & De Backer, 2013), or "cognitive apprenticeships", such as that described by Boyer et al. (2008), which is centred around live demonstrations of how to solve particular problems. Boyer et al. (2008) also utilised peer learning activities and online discussion-based activities to help promote active engagement amongst students in order to challenge students in their zone of proximal development. These techniques could naturally form the basis of the main teaching within an introductory programming module, or as part of dedicated, targeted interventions, which could be developed as part of an extension to RQ 3 in future work.

2.4 Programming Cognition

Programming is a complex, abstract process that can be difficult to learn (Guzdial, 2010; Khalife, 2006). Therefore, in order to conduct meaningful research, and to develop effective tools to support the teaching of introductory programming classes, an understanding of the cognitive processes involved in programming must be established. There is a strong bi-lateral relationship between programming and the field of cognitive psychology, as programming offers cognitive psychologists the ideal opportunity to examine cognitive processes in a real-world domain whilst participants are carrying out clearly defined tasks. Similarly, cognitive psychology offers methods for examining the processes that underlie performance in computing tasks (Ormerod, 2014).

Ormerod (2014) compares the cognitive processes being carried in the brain to those of a computer, stating that much of cognitive psychology is based on a “computational metaphor” in which the mind is viewed as a type of information processor. Ormerod goes on to discuss the computational metaphor in more detail, stating that the brain carries out processes such as memory storage and retrieval, language production and comprehension, attention, perception and problem solving that are also carried out by a computer’s Central Processing Unit (CPU).

A computer system can be broken down into three key levels: the software; the implementation of programs on the hardware (e.g., memory allocation, CPU speed, etc.); and the hardware itself. Ormerod (2014) adopts a literal interpretation of the computational metaphor when describing the human cognitive system: the cognitive software is made up of mental procedures and representations of knowledge used in performing cognitive tasks. Cognitive implementations of software relate to the mechanisms for carrying out mental procedures and knowledge representation, such as storage, retrieval or symbol manipulation, where limitations in attention and memory can hamper problem solving. Finally, the cognitive hardware are the physiological structures in which cognitive processes are carried out, specifically, the human brain.

An important construct relating to knowledge representation within the human cognitive system is that of the “schema”. A schema is a type of cognitive software that in essence, is a data structure used to represent generic concepts being stored in a person’s memory (Ormerod, 2014; Rumelhart & Ortony, 2017). Schemas are used to hold generalised versions

of objects, situations, events and sequences of actions or events, essentially representing a stereotype of a given concept. Anderson (2015) states that schemas represent categorical information using a slot structure with each slot representing a particular category.

The slots within a schema can hold multiple *default values* or specific instances of the slot's attributes, as shown in Table 2.1 (Anderson, 2015).

Table 2.1

Example Schema of a House

Category	Attributes
Isa	Building
Parts	Rooms
Materials	Wood, Brick, Stone
Function	Human Dwelling
Shape	Rectilinear, Triangular
Size	100 – 10,000 square feet

The attributes listed in Table 2.1 are the *default values* for the category; what you would expect to see. However, other values that have not been included are also acceptable as Anderson explains, "the fact that houses are usually built of materials such as wood, brick, and stone does not mean that something built of cardboard could not be a house" (Anderson, 2015, p.113). Additionally, schemas can include an '*isa'* slot, which unless contradicted, allows a concept to inherit the features of a higher concept through a generalisation hierarchy, for example, a schema for a house inherits from a building schema, thus negating the need to explicitly represent the building's features within the house schema (Anderson, 2015).

Interestingly, the way schemas represent information is similar to that of a "Class" within object-oriented programming, as the schema itself represents a class definition, slots represent data members, attributes represent the values assigned to the data members and the real-world object that the schema describes is represented by an instance of a class. Additionally, classes can also inherit from one another in the same way that schemas do using an *Isa* slot.

It is worth noting that the content of a schema is purely an abstract representation of a particular concept, independent from any particular instance, and is based solely on an individual's prior knowledge (Ormerod, 2014). Drawing on the work of Bartlett (1933), Sweller (1994) provides a useful analogy for explaining how the mind processes schemas, by examining schemas designed for handling trees. Sweller states that no two trees are identical, but all share common features such as branches, height, colour, etc. When asking a person to describe a particular tree from memory their description will be heavily influenced by their tree schema rather than the exact features of the tree they were asked to describe. Therefore a person can easily process potentially infinite varieties of trees by incorporating them into the existing tree schema (Ormerod, 2014).

According to Sweller (1994), intellectual skills are gradually learnt through incremental schema acquisition, which when first acquired, will be severely constrained until a person becomes proficient in the specific skill. Similarly, Perkins et al. (1986) state that experienced programmers rely on a repertoire of well-practiced schemas that are developed over time, which as Winslow (1996) suggests, may take up to ten years before an individual has acquired sufficient knowledge to be classed as an "expert" programmer.

Skills such as programming are gradually learnt through incremental schema acquisition, which when first acquired, will be severely constrained until sufficient experience has been gained in applying the schemas to allow the individual to become proficient in the skill (Sweller, 1994). Subsequently, as a skill is learned and reinforced, the way it is processed in the brain changes. A highly cited psychological model developed by Schneider and Shiffrin, which was presented across two separate publications (Schneider & Shiffrin 1977; Shiffrin & Schneider, 1977), implies that when a skill is first learned it requires the explicit attention of the individual in order to carry it out. As such, this can often be time consuming and require conscious effort to move from one step to another (Paas & Van Merriënboer, 1994; Sweller, 1994). Schneider and Shiffrin (1977) therefore, termed this type of information processing *Controlled Processing*. In contrast to controlled processing, Schneider and Shiffrin (1977; Shiffrin & Schneider, 1977) also defined *Automatic Processing*, which occurs with no conscious attention from the individual. Automatic processing of a skill is triggered by a corresponding stimulus (input) and is then processed automatically without any explicit intervention from the individual, thus allowing for faster processing, which appears effortless

to the individual (Paas & Van Merriënboer, 1994; Schneider & Shiffrin, 1977; Sweller, 1994).

Paas and Van Merriënboer (1994) describe the process of a skill developing from purely controlled to purely automatic processing as *Rule Automation*, which occurs through continuous practice allowing for the development of "rules" that control problem solving behaviour, often over a prolonged period of time. For example, a young child who is learning to read must make a conscious effort to read and understand a simple sentence; however, an adult who has been reading successfully for a significant number of years would not need to actively devote attention to deciphering the meaning of individual letters or words (Sweller, 1994). Therefore, the child is seen to be using controlled processing, whereas the adult is using automatic processing.

Paas and Van Merriënboer (1994) go on to state that novel and inconsistent processing tasks typically necessitate controlled processing, whereas automatic processing typically occurs in well-practiced consistent tasks. However, complex cognitive tasks require a combination of both controlled and automated processing due to the fact that these kinds of tasks can contain aspects that cannot be easily automated.

Although schema construction and automation play an important part in the acquisition and development of an intellectual skill, they also play an important part in reducing the amount of working memory required to perform the skill. Working memory is essentially an area of memory. which is dedicated to storing information needed to perform a task that is currently being carried out (Anderson, 2015; Baddeley, 1992). Information is stored in *slots* within working memory, with the total number of slots (i.e., a person's working memory capacity), being of a fixed amount. Miller (1956) theorised that an individual's working memory has 7 (+/-2) slots available, thus imposing a limit on the amount of information that can be held at any one time in working memory.

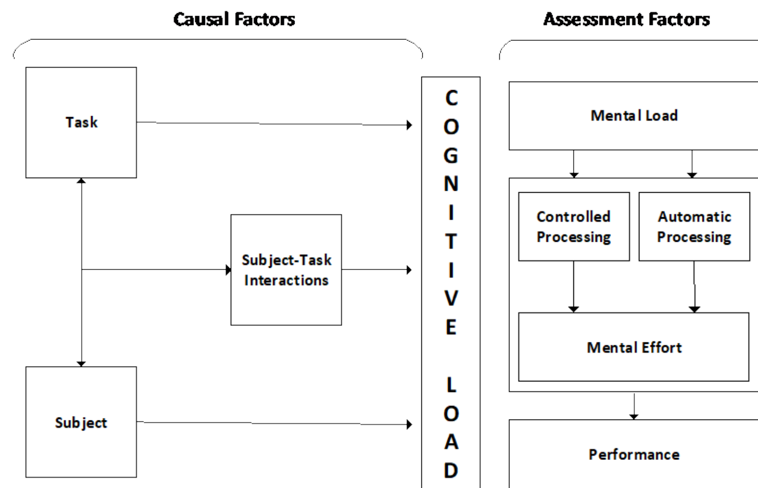
Owing to the limits of working memory it can become a bottleneck when learning new intellectual skills (Duran et al., 2018) as more complex tasks require a greater amount of working memory, which in turn degrades the learner's performance once their working memory capacity has become overloaded (Paas et al., 2003). However, the use of schemas decreases the overall demands placed on working memory by increasing the amount of

information that can be stored within a single slot (Sweller, 1994). For example, a schema's representation of a car can be stored within a single slot without the need to also recall the individual elements (engine, doors, wheels, etc.) into working memory. Similarly, once a schema has been fully automated there is no need to recall it into working memory, therefore bypassing it entirely as processing occurs automatically and allows for other functions to utilise available working memory capacity (Paas et al., 2003; Sweller, 1994).

When an intellectual skill is first encountered, a significant amount of cognitive resources must be utilised in order to process it until a sufficient number of schemas have been constructed and automated, thus freeing resources for other activities (Sweller, 1994). The amount of load that performing a particular task poses on a person's cognitive systems is represented by Sweller's (1988) *Cognitive Load Theory*. Paas and Van Merriënboer (1994) provided an in-depth examination into the causes and effects of cognitive load on skill development, stating that the construct of cognitive load is comprised of causal and assessment factors; the factors that affect cognitive load and the factors that are affected by cognitive load, respectively. Figure 2.1 depicts Paas and Van Merriënboer's (1994) diagrammatic overview of the concept of cognitive load, with casual factors on the left, and assessment factors on the right.

Figure 2.1

Representation of Cognitive Load



Note. From "Instructional Control of Cognitive Load in the Training of Complex Cognitive Tasks" by F. G. Paas and J. J. Van Merriënboer, Educational Psychology Review.

The casual factors of cognitive load relate to the characteristics of the task (or the environment in which it is being performed), the characteristics of the individual performing the task, or interactions between the two (Paas et al., 1994; Paas & Van Merriënboer, 1994). For example, Paas and Van Merriënboer (1994) suggest that task characteristics such as task novelty, time available to complete the task and the possible rewards from the task can all influence levels of cognitive load. Additionally, an individual's characteristics can affect levels of cognitive load, such as task-relevant prior experiences, cognitive style, preferences and the like. These characteristics tend to be relatively stable, meaning they are unlikely to suddenly change when performing a task (Paas et al., 1994; Paas & Van Merriënboer, 1994). However, factors such as motivation, state of arousal and internal criteria of optimal performance, are dependent upon interactions between the individual and the task, and are, therefore, unstable, meaning that they may not remain fixed throughout the duration of the task (Paas et al., 1994; Paas & Van Merriënboer, 1994).

There are three key factors that can be used to assess an individual's cognitive load: *mental load, mental effort* and *performance* (Paas & Van Merriënboer, 1994). Mental load is determined by task or environemnt demands and as such, is independent from an individuals

characteristics (Paas et al., 1994; Paas & Van Merriënboer, 1994). Paas and Van Merriënboer gave the following example to demonstrate mental load:

"[S]uppose that there are two maze tasks A and B, and that maze A is more complex than maze B. Then, for all subjects solving the task, the mental load releated to task A is higher than the mental load related to task B."
(Paas & Van Merriënboer, 1994, p. 354)

From this example it is evident that the mental load of a task is a consistent factor across all personel involved in completing that task. Subsequently, mental effort refers to the amount of cognitive resources required to complete a particular task (Paas & Van Merriënboer, 1994) and is therefore a differentiating factor amongst a set of individuals completiing a task. The amount of mental effort required to perform a task also reflects the amount of controlled processing being carried out (Paas et al., 1994), and, as such, tasks that require more controlled processing will incur a higher cognitive load compared to those that can be processed automatically (Sweller, 1994). Paas and Van Merriënboer (1994) again refer to their maze example to demonstrate mental effort, stating that as task A is more complex, and commands a higher mental effort, it will therefore often show a higher level of mental load. However, as Paas and Van Merriënboer go on to explain, this assumption is not always accurate, as it is possible that an individual may "brush aside" task A due to its complexity and put a lot of effort into completing task B, thus resulting in a higher level of mental effort for task B than task A. Additionally, prior expierence can also affect mental effort as a group of students who have no previous knowledge of either of the mazes will show higher levels of mental effort for task A, whereas a group of students who have had expierence with task A but none with task B will show a higher level of mental effort with task A (Paas and Merriënboer, 1994).

Due to the subjective nature of mental effort, it is also important to consider an individual's performance on the task that is being examined, in order to accuratly determine the level of cognitive load being experienced by an individual, as all three causal factors are reflected within task performance (Paas et al., 1994; Paas & Van Merriënboer, 1994). Refering again to Paas and Van Merriënboer's (1994) maze example, performance is likely to be higher on the simpler of the two tasks (task B) with the maze being completed quicker and with fewer

errors. However, it may possible to obtain a similar level of performance on the more complex task through an increase in a mental effort (Paas & Van Merriënboer, 1994), and therefore an increase in overall cognitive load. Differences in prior knowledge and expierence are also likely to influence performance levels when completing these tasks.

The level of cognitive load experienced by a student may be a significant factor in determining whether a particular topic or concept is successfully learnt by the student (Paas, et al., 2003). The overall level of cognitive load a person experiences is in fact a culmination of three specific types of cognitive load: *intrinsic load, extraneous load* and *germane load* (Paas, et al., 2003). Paas et al. (2003) provide the following definitions for each type of cognitive load:

- Intrinsic Load – is the interaction between what is being learned and the expertise and experiences of the learner.
- Extraneous Load – is the extra load on top of intrinsic load, which is mainly produced through poorly designed instructions.
- Germane Load – is the amount of load required for the creation and automation of schemas.

Whilst Paas et al. (2003) acknowledge that each of the types of cognitive load combine to create a *total cognitive load,* more research is required to study how each of these types of cognitive load can be measured individually and to investigate any potential relationships with performance. The causes of a high overall level of cognitive load have been discussed in length by Paas and Van Merriënboer (1994). They suggest that there are a number of characteristics of complex tasks that invoke a high level of mental load, which often results in high levels of mental effort and consequently, a high level of cogntive load. Paas and Merriënboer (1994) identified two specifc factors that can contribute to high levels of mental load: the number and nature of *component skills* involvelved in completing the task, and the complexity of the *goal hirearchies*.

Paas and Van Merriënboer (1994) define component skills as “subskills that form part of the to-be-learned skill”. Consequently, Paas and Van Merriënboer suggest that skills with greater numbers of component skills induce a higher level of mental load than those with fewer

component skills. Paas and Van Merriënboer (1994) go on to state that the component skills can either be *recurrent components*, which dictate a consistent level of performance across problem situations, or *nonrecurrent components*, which having varying levels of performance between tasks, with tasks requiring a greater number of nonrecurrent components imposing a greater demand on an individual's cognitive system.

To demonstrate their definition of component skills, Paas and Van Merriënboer (1994) use examples from the context of programming. Tasks such as the use of an Integrated Development Environment (IDE), selecting basic commands and applying syntax rules can all be classed as recurrent components, whereas tasks including problem decomposition, or the identification and resolution of errors can be classed as nonrecurrent components. In addition to component skills, Paas and Van Merriënboer (1994) identified that the complexity of the sub-goals which must be accomplished in order to achieve a specific overall goal; the *goal hierarchy*, can also provoke higher levels of cognitive load through more complex goal hierarchies. It is evident from the work conducted by Paas and Van Merriënboer (1994) that as the complexity of the task increases, so does the overall cognitive load experienced by the individual.

Sanders and Thomas (2007) conducted an investigation into the misconceptions shown by 16 novice programmers across five separate programming assignments. Their aim was to support instructors of an object-oriented programming course, by developing a number of checklists to be used to assess students' understanding of programming concepts, and also diagnose common problems in their programs. Sanders and Thomas (2007) developed their checklists by manually reviewing "several hundred pages of code" which they admit was a time consuming task. A similar study in the future could potentially use a selection of the Educational Data Mining techniques, discussed in Section 3.3, to conduct a more efficient study, with a reduced potential for bias. However, during the course of their investigation, Sanders and Thomas identified that as the complexity of programs increased, so did the number of "elementary" syntactic mistakes. Although this could be, as Sanders and Thomas (2007) suggest, due to a "lack of time rather than lack of understanding", their finding could be a potential indicator of the increased cognitive load placed on the students, which is inducing a greater number of simplistic mistakes, thus supporting Paas and Van Merriënboer's (1994) claims.

In addition to Sanders and Thomas' (2007) evidence in support of the effects of cognitive load, a study conducted by Anderson and Jeffries (1985) appears to also support Paas and Van Merriënboer's (1994) view. Anderson and Jeffries (1985) present a series of experiments that studied the errors made whilst learning to program in LISP, the first of which examined how the complexity of the tasks students were performing affected error rates. Like Sanders and Thomas (2007), they determined that as the programs students were writing became more complex, the more errors were made. Anderson and Jeffries (1985) go on to state that they believe the increase in error rate is due to the excessive demands placed on the students' working memory and may in fact be responsible for the majority of errors made by the students rather than their own misunderstandings.

It is reasonable to assume, based on the evidence discussed above, that students experience higher levels of cognitive load when completing more complex programming tasks. This higher level of cognitive load has the potential to have a negative effect on the students' learning of new material (Paas & Van Merriënboer, 1994), whilst also increasing the likelihood of mistakes being made on previously learnt content (Anderson & Jeffries, 1985; Sanders & Thomas, 2007). As such, attempting to learn new tasks whilst contending with high levels of cognitive load will be difficult for students (Sweller, 1994), thus making the level of cognitive load a student experiences a limiting factor when attempting to convey tasks of substantial complexity (Paas et al., 1994).

Whilst the design of instructional materials is outside the scope of this investigation, it is important to highlight the impact students' levels of cognitive load can have on their abilities to fully engage with the learning process (Berssanette & De Francisco, 2022). Given the complexities associated with learning to program, it has been suggested that cognitive overload is one of the primary problems within introductory programming due to the increased demand that is placed on students' working memory (Yousoof et al., 2007). Students' levels of cognitive load are, therefore, likely to be a factor in students' mental model development and their levels of confidence, and as such, may influence the results for both RQ 1 and RQ 2.

It should be noted, however, that whilst the level of intrinsic load placed on students is fixed, the level of extraneous load can be reduced (Sweller, 1994), for example, by reducing the number of new topics introduced in a single lesson, ultimately reducing the overall demands

on students' cognitive systems. If lessons are not suitably designed to reduce cognitive load the possibility of students experiencing cognitive overload is increased. This is when the demands of the task being completed by a student exceed their cognitive capacity and consequently, increase the likelihood that a student will fail to learn the topic being taught (Paas et al., 1994). A student experiencing "programming shock" is a good example of how presenting numerous complex, unfamiliar concepts simultaneously can result in cognitive overload and hamper further progress. If a student frequently experiences cognitive overload, Paas et al. (1994) believe that it may result in the student losing motivation within the subject, and ultimately failure, consequently making it especially important that students' cognitive load levels be considered when designing course materials.

As discussed previously, mental effort is a key component that can be used to assess an individual's level of cognitive load. An interesting experiment conducted by Mason and Cooper (2012) examines the views of academic staff from 28 Australian universities on the characteristics of the "bottom 10%" of students on their introductory programming modules. Mason and Cooper (2012) postulate that many low performing students lack relevant pre-existing schemas, which are necessary for understanding programming concepts, and experience high levels of intrinsic and extraneous cognitive load when faced with an instructional presentation. This, in turn, blocks the capacity for germane load, and therefore prevents learning. Mason and Cooper go on to explain that the three types of cognitive load broadly align to what must be processed by students when learning to program, which are as follows:

- Intrinsic load, associated with the concepts and interpretation of the problem statements;
- Extraneous load, determined by the language and environment, along with associated constraints such as syntax; and
- Germane load, associated with the cognitive processing to acquire and automate new schemas.

  (Mason & Cooper, 2012, p. 191)

Subsequently, participants within Mason and Cooper's (2012) study were asked to provide mental effort ratings on three areas of cognition (understanding the problem statement, using

the development environment and reinforcing previous concepts) for themselves (instructors), an average student and a student in the bottom 10% of their class. Perhaps unsurprisingly, the participants in Mason and Cooper's (2012) study rated their own mental effort for each of the three components as low, with an average student needing to exert "above average" levels of mental effort and students in the bottom 10% needing to exert much higher levels of mental effort than an average student. Mason and Cooper state that these results are consistent with their argument that "students in the bottom 10% of performance are perceived by the introductory programming instructors to be effectively swamped in mental effort on each of the three measures" (Mason & Cooper, 2012, p.5). Mason and Cooper (2012) go on to identify three main "profiles" of students in the bottom 10% of their introductory programming modules based on the interviews conducted with academic staff. They are as follows:

- 'Strivers' – less capable students who are actually trying.
- 'Idlers' – those students who attend class but do not try.
- 'Ghosts' – students who do not attend class/are not seen by instructors.
  (Mason & Cooper, 2012, p. 192)

Mason and Cooper's (2012) identification of the Idler and Ghost profiles raises an important point in relation to student success within a course, as in order to succeed a student must be willing to put in a reasonable level of effort. Students who never attend classes (Ghosts) or attend but do not apply themselves (Idlers) are much more likely to exhibit misunderstandings of fundamental programming concepts as they have not put in the effort to learn them. Paas et al. (1994) also touched on this point when stating that students failing to learn complex tasks can be attributed to the task demands exceeding their cognitive capacity, an inadequate allocation of attention from the student, or both.

On the contrary, students who are identified as Strivers are, according to Mason and Cooper (2012), putting in very high to extreme levels of mental effort and yet are failing to learn and cannot be expected to put in any more effort than they are already doing. The students in Mason and Cooper's (2012) investigation showed very clear distinctions between those who are failing due to a lack of effort, and those who are failing despite a significant amount of effort being exerted. It would therefore be beneficial to investigate what factors lead to a

student becoming a Striver, Idler or Ghost and what can be done to lift them out of the bottom 10% of the class, such as the deployment of methods for getting students re-engaged with their course (Ghosts and Idlers) or the use of alternative teaching techniques to reduce the overall cognitive load on students and help them develop the schemas needed to progress within the course.

Additionally, Mason and Cooper's (2012) investigation could be expanded to include mental effort measurements directly from students, thus allowing for a more comprehensive analysis on students' mental effort levels to be undertaken, which is not solely reliant on the views of academic staff. The literature presented clearly indicates that learning to program requires significant mental effort on the part of the student, particularly in students who appear to be struggling as they are likely to become "swamped" by the mental effort that they are required to exert, according to Mason and Cooper (2012), which may limit their progress in the course. Furthermore, students who may have little or no prior experience with programming may be more likely to experience "programming shock" and, as such, be at risk of cognitive overload. This, therefore, supports the inclusion of RQ 2, which allows for the identification of any significant relationships between students' prior experiences, as well as perceptions of confidence, and their programming abilities. It should be noted, however, that newer formulations of Cognitive Load Theory now consider Germane Load to be a component of working memory used to handle information associated with intrinsic cognitive load (Berssanette & De Francisco, 2022; Duran et al., 2022), so whilst Mason and Cooper (2012) refer to what can be considered 'Old Cognitive Load', the implications of their findings remain relevant.

2.5 Students' Interpretations of Programming Concepts

The process of learning to program is a complex, and often daunting one that can require a student to re-evaluate their current, albeit naive, understanding of not only the programming language itself, but also the computer system as a whole (Cheah, 2020; Guzdial, 2010; Pea & Kurland, 1984; Perkins et al., 1986; Rogalski & Samurçay, 1990). The complex cognitive changes that are required to develop an understanding of basic programming concepts are, according to Pea and Kurland (1984) "unlikely to occur through either spontaneous exploration or explicit instruction alone" (p.140), thus requiring a student to be fully engaged in both the task and the programming course in general to support their emerging understanding of programming concepts.

Linn (1985) provides a chain of potential "cognitive accomplishments" that a student must achieve in order to successfully learn to program:

- Learning the language features – the fundamental, non-decomposable concepts of the programming language, such as variables, if statements, etc.
- Learning to design programs to solve problems – by developing a repertoire of templates; stereotypical patterns of code used to perform specific tasks or parts of tasks, and developing the procedural skills required to combine templates and/or features of the language to perform a specific task. Additionally, procedural skills are developed to ensure programs accomplish the stated objectives or to redevelop the solution until the objectives are met.
- Learn problem-solving skills, which can be applied to other formal systems – for example, being able to solve problems in a different programming language. Additionally generalised procedural skills for carrying out planning, testing and reformulating problems in a variety of formal systems will also be developed.

Despite Linn's work being carried out in the mid 1980's, the cognitive accomplishments she describes are still very relevant to today's introductory programming classes and in fact, reflect many of the factors that cause students initially to become overwhelmed, as described by Du Boulay (1986). One factor that may create a barrier to students achieving the cognitive accomplishments set out by Linn (1985) was identified by Sorva (2013) as the fact that in some disciplines, there may be concepts that are not fixed and are open to interpretation by

the student. This is not the case when it comes to programming, as concepts such as variable assignment, if statements and for loops have all been designed to operate in a particular way. However, as Sorva (2013) states, novice programmers may misinterpret these concepts and may make mistakes whilst using them, and whilst these mistakes may appear trivial to experts, misunderstandings such as these can be widespread amongst novice programmers and difficult to overcome.

As noted earlier, students' interpretations of fundamental programming concepts (correct or otherwise) can be described as their *mental models*, which can be broadly defined as a mental representation of the properties and behaviours of a given concept that is based upon an individual's prior knowledge and experiences (Norman, 1983; Sorva, 2013). Johnson-Laird, who is cited to be one of the pioneers of Mental Model Theory (Sasse, 1997), suggests that humans view and interprets the world in accordance with their pre-existing mental models (Johnson-Laird, 1983, 2010). An additional significant contribution to Mental Model Theory was a series of observations conducted by Norman (1983), whose study involved observing a wide variety of subjects carrying out a diverse selection of tasks ranging from the use of everyday technologies. including computers, text editors, calculators, cameras, and digital watches, to highly specialised tasks such as piloting aircraft. Norman's (1983) observations resulted in the development of a set of general characteristics of mental models:

1. Mental models are incomplete and simplified due to limited knowledge or experience.
2. People's abilities to "run" their models are severely limited.
3. Mental models are unstable – that is, details of the system can be forgotten if it has not been used in a while.
4. Mental models do not have firm boundaries, leading them to be confused with models of other similar systems.
5. Mental models are "unscientific" as people maintain superstitious behaviour patterns even when they are known to be unnecessary. Norman demonstrates this point with his observation of calculator usage, as a major pattern amongst participants was the belief that the CLEAR function needed to be carried out several times, or even before any calculations had been entered on calculators. which did not require the user to do so (Norman, 1983, p.10).
6. Mental models are parsimonious, as people often perform unnecessary actions that could be avoided by mental planning. People would rather carry out additional

physical actions instead of increasing mental complexity, especially when this allows a single simplified rule to be applied to multiple devices, which consequently, reduces the chances for confusion.

Norman's (1983) seminal characteristic set highlights the fact that a person's mental model typically only represents a limited portion of a particular concept (or system) and that people's ability to run through the model is extremely limited, which in turn, limits their ability to fully comprehend and utilise the concept to its fullest extent. In terms of learning to program, the lack of appropriate mental models makes learning to program especially difficult, particularly for students who have not studied programming or computer science before, as mental models are constructed from what the student believes to be related prior knowledge (Ben-Ari, 2001; Sorva, 2013).

Although students may be able to develop an understanding of the syntax of the language, they are likely to lack appropriate models of programming concepts at the beginning of their course, which are of critical importance for solving programming problems (Ben-Ari, 2001; George, 2000). This initial lack of appropriate models is likely to make the process of learning to program more difficult for students, as when faced with unfamiliar scenarios, students will attempt to construct models based on superficially similar tasks, which may or may not be appropriate (Ben-Ari, 2001; Sorva, 2013). Sorva (2013) gives the example of how when faced with an unfamiliar Graphical User Interface (GUI), a person creates an initial model based on GUIs they have encountered previously in an attempt to understand how to navigate the new system. Similarly, when a student is faced with a programming concept that resembles something they are familiar with, confusion can arise. For example, students may attempt to use the assignment operator; which is denoted by an equals sign (=), in the same way it is used in mathematics, which is to signify equality, whereas in many programming languages, equality is in fact signified with a double equals sign (==). These inaccurate mental models have been constructed based on inaccurate or irrelevant prior knowledge and will likely lead to mistakes being made by the student, which if made repeatedly, will create a barrier to learning (Sirkiä & Sorva, 2012).

The process of learning to program can be viewed as students' development of coherent mental models, that represent the actions fundamental programming concepts perform when processed by a computer (Ben-Ari, 1998; VanDeGrift et al., 2010). It is therefore vital that

teaching staff understand and account for any preconceptions students hold at the beginning of a course by explicitly teaching the mental model that students need to develop (Ben-Ari, 1998), that is, by presenting walkthroughs of how a particular concept operates and addressing any commonly held misconceptions directly (Sudol & Jaspan, 2010). If students' mental models are not taken into account within the teaching, it is likely that students will construct models that do not fully represent the concepts being taught, which will result in misconceptions being developed (Winslow, 1996). As such, RQ 1 focuses on examining the mental models that students initially hold at the start of their course, and how they progress during the first semester, which ultimately supports RQ 3 through the use of students' mental models when attempting to predict their assessment results.

By the end of their introductory programming module students should have begun to develop models that encompass how fundamental concepts operate within a program, typical ways of solving common problems, as well as general knowledge about the syntax of the language they have learnt (Canes et al, 1994). However, Sorva (2013) cites studies by Tew (2010) and Kunkle (2010) as having revealed that the difficulties of introductory programming modules to adequately teach students fundamental programming concepts is not limited to a single institution or programming language.

Kunkle and Allen (2016) republished Kunkle's original study from 2010, which revolved around the development and validation of an instrument designed to assess students' understanding of both fundamental and object-orientated concepts. In order to develop the instrument, Kunkle and Allen (2016) examined differences in teaching approach and the language being learnt by students, by conducting two data collection sessions – one at the start of the term and one at the end. The sessions consisted of a demographic survey, an attitude survey to gauge students' attitudes towards computer science as a subject and a computer concepts survey, which assessed students' programming knowledge through a series of 24 multiple choice questions.

The teaching approaches Kunkle and Allen (2016) examined included *objects-first,* which introduces students to the more advanced topic of object-orientation at the beginning of their studies, *imperative-first,* which introduces students to procedural programming, that is, functions, program logic, etc. first, leaving object-orientated concepts until last, and also a *project-based* approach to learning to program. Additionally, the languages students were

taught differed between each approach, with students studying the object-first approach learning Java, with students studying the imperative-first approach learning C++ and with project-based students learning Visual Basic.

Kunkle and Allen's (2016) analysis revealed differences in student performance using their assessment instrument with students who studied the objects-first approach remaining consistent in their performance and imperative-first students improving between the two tests. However, students who studied the project-based approach were found to be performing worse in the second test than the first. Although Kunkle and Allen (2016) attempt to explain the drop in performance due to students "forgetting" concepts, it is difficult to ascertain whether students' performance is being affected by the teaching approach or the programming language being studied, owing to each teaching approach utilising a different programming language. Kunkle and Allen (2016) do admit that they cannot fully explain their findings, as well as acknowledging that any claims they make may be debatable due to the lack of consistency in introductory programming syllabuses. However, their study does identify the fact that students' performances can differ between introductory programming modules, whether this is because of a difference in programming language or a difference in teaching approach. As such, they identify the fact that instructors must take care when designing their courses in order to extract the most from students.

Tew (2010) also took the approach of performing a language-independent study through a series of six experiments that were used to inform and validate the design of a programming assessment instrument. Within these studies it was revealed that students exhibited a significant number of misconceptions. However, Tew (2010) states that being able to determine whether students' errors are caused by conceptual misunderstandings instead of syntactical mistakes was not possible from her study and is unsure whether any further studies would be able to reasonably investigate this phenomenon owing to how the syntax and semantics of a language are deeply intertwined.

The literature discussed in this section has so far demonstrated the importance of appropriate mental model construction for students learning to program. However, during the development of these models, students can inadvertently develop misconceptions which affect their understanding of the concepts they are trying to learn. Various definitions of what constitutes a programming misconception exist, with broad definitions being provided by

Sorva (2013) who defined them as “understandings that are deficient or inadequate for many practical programming contexts” (p. 4), and Qian and Lehman (2017) who view programming misconceptions as conceptual misunderstandings. A more direct approach is taken by Chiodini et al. (2021) who state that a “programming language misconceptions is a statement that can disproved by reasoning entirely based on the syntax and/or semantics of a programming language” (p. 381). Chiodini et al.’s. (2021) definition is tightly focused on misconceptions associated with a particular programming language, although they do acknowledge that misconceptions can exist across multiple languages.

However, Evans et al. (2023) take the view that misconceptions are the properties of the learner, not programs or languages, with Sorva (2012) stating that generic misconceptions may lie behind more specific ones. He subsequently provides a comprehensive list of what he considers to be generic misconceptions that demonstrate inaccuracies in students' understandings of the execution of programs. Given the language-independent nature of this investigation, the misconceptions being examined fall in line with the broader definitions of programming misconceptions (Qian & Lehman, 2017; Sorva, 2012), with a view being taken that the misconceptions students demonstrate are “symptoms” of inaccurate mental models of a particular concept.

A significant factor that can contribute to students developing misconceptions is the existing knowledge they bring to their programming classes – their “preprogramming knowledge” as termed by Bonar and Soloway (1985), hence the need to examine students’ previous experiences as part of RQ 2. Despite programming being a drastically different subject from what students have studied previously, it is not appropriate to treat it as being isolated from the wider world as students are able to intuitively comprehend various programming concepts by drawing on content learnt in other subjects, as well as their interactions with the physical world (Pea, 1986; Qian & Lehman, 2016, 2017; Robins, 2010, 2019; Smith et al., 1994).

Both Pea (1986) and Bonar and Soloway (1985) suggest that students draw on natural-language concepts when constructing mental models of programming concepts such as looping, decision making and specifying and following instructions in a set order. However, in some cases, the analogy of human-like conversations can lead students astray with their mental model construction. Bonar and Soloway (1983) demonstrate this point with an example of how students can misinterpret “while” loops by assuming that the code contained

within the loop is continuously being evaluated for the break condition becoming true, as opposed to the actual way while loops operate, which is by evaluating the break condition once per iteration. Bonar and Soloway (1983) equate this to how the word "while" is commonly used within natural English to describe a continuous condition – that is, "while the highway is in two lanes, continue north". Similarly, Clancy (2004) refers to the mismatch between some programming key words, such a "while", and natural English as *linguistic transfer* when identifying it as a potential source of confusion for students. Simple misunderstandings such as these form a barrier to the development of accurate mental models and as such, become confounded into stronger misconceptions that can then interfere with students' learning (Cheah, 2020; Smith et al., 1994). These difficulties can be further compounded if English is not a student's first language, as students are required to translate concepts into their native language, subsequently increasing extraneous cognitive load and creating an additional barrier to learning (Guo, 2018; Qian & Lehman, 2016).

Students' difficulties in predicting program outputs (mentally tracing through the program and processing each instruction) were suggested to still be present after more than a year of instruction by Pea (1986). Pea (1986) goes on to state that a number of misconceptions students possess arise from a "*superbug*", in which they believe that there is a "*hidden mind* somewhere in the programming language that has intelligent, interpretative powers" (p. 5). This conceptual superbug is a culmination of three individual classes of bugs (misconceptions) which can lead students astray (Pea, 1986), the first of which is the *parallelism* bug. Pea (1986) explains that while it can appear in a variety of contexts, fundamentally, the parallelism bug refers to when a student assumes different lines of a program can be concurrently active, for example, a student may mistakenly believe that a program will continually evaluate an "if statement" such as the following example:

```
int a = 4;
if (a > 10)
{
	std::cout << "Hello World";
}
```

As the variable “a” has a value of 4, the if statement evaluates to false and as such, the code contained within the if statement is not processed. However, if later in the program the value of “a” is increased to a value greater than 10 students believe that the program would output “Hello World”, thus demonstrating a misunderstanding of the flow of control within the program.

The second class of bug identified by Pea (1986) is the *intentionality bug***,** which is where students demonstrate the belief that a program has the ability of foresight and can go beyond the information explicitly given in the code, in essence, making the program self-aware. Pea (1986) goes on to define an additional bug termed as the *egocentrism* bug, which as Pea states, is the opposite of the intentionality bug and represents students' mistaken belief that there is more meaning to the code that has been written than there actually is, consequently meaning that students believe that the program can do more than what is has been explicitly told to do. This type of inappropriate mental model could prove particularly frustrating for students as they struggle to get to grips with the basic syntax and semantics of the language they are trying to learn.

Despite Pea's (1986) work being carried out over thirty years ago, “little has changed” according to Simon (2011) when highlighting that students still struggle with issues relating to Pea's (1986) parallelism bug. Furthermore, Kwon (2017) provides a real-world example of the egocentrism bug, which was uncovered whilst analysing solutions provided by a group of undergraduate students. Kwon (2017) begins by describing a separate misconception amongst students where they would declare multiple variables in accordance with the number of expected values – that is, students would define variables such as “m” and “f” (male and female) instead of declaring a single gender variable which could hold “m” or “f” as a value. Kwon (2017) goes on to describe how students demonstrate the egocentrism bug by “assuming the computer would be able to tell the gender if they specified the gender in form of “if m” or “if f”. This, therefore, shows a misunderstanding of how a conditional statement must be used to evaluate an if statement while also showing an assumption that the program understands what is being implied by “if male”, which would be acceptable in natural-language, but not within a computer program.

Pea's (1986) description of the inappropriate mental models that students use to comprehend how the code they write translates into actions being performed by the computer, reflects Du

Boulay, O'Shea and Monk's (1999) view that teaching students how to control the machine they are using is one of the more difficult aspects of learning to program. As mentioned earlier, Du Boulay et al. (1999) suggest that the learning process can be made easier for students by introducing them to programming through the use of a *notional machine* – an abstracted model of the computer that is used to understand what happens when a program executes, which is influenced by the programming language being used rather than the specific computer hardware (Du Boulay et al., 1999; Sorva, 2013). Sorva (2012) highlights that many of the misconceptions that students exhibit are as a result of a lack of an appropriate mental model of the notional machine, whereby students do not have a clear model of how the program is executed. However, Sorva (2012) goes on to state that in addition to understanding what actually happens when a program is run, students must also recognise what a notional machine (and therefore, a computer) does *not* do unless explicitly instructed to do so by a programmer. Students who fail to recognise what explicitly needs to be defined within a program could be seen to be demonstrating elements of Pea's (1986) Superbug.

A potential implementation of a notional machine has been developed by Berry and Kölling (2013), although more research is required to evaluate if this approach is truly beneficial to students' learning as there is limited literature available that directly measures the effectiveness of teaching introductory programming using a notional machine (Fincher et al., 2020). Support for the use of notional machines within introductory programming modules is, however, provided by Johnson et al.'s (2020) view, that teaching introductory Python without use of a notional machine to support students' comprehension of the underlying concepts, can result in students developing misconceptions and subsequently, inadequate mental models (Dickson et al., 2020; Johnson et al., 2020).

Whilst Pea's (1986) work takes a more generalised view of students' mental models of program execution, an alternative research approach has been to examine students' models of individual programming concepts. One rather controversial study that took this approach was conducted as part of Saeed Dehnadi's PhD research in which he explores how mental models of variable assignment can be used to predict success within an introductory programming module. Dehnadi's work is presented across a set of papers (Bornat et al., 2008; Dehnadi, 2006; Dehnadi et al., 2009; Dehnadi & Bornat, 2006) in which students mental models of variable assignment were assessed through a series of multiple choice questions. Dehnadi's

(2006) questions asked students to predict the values of each variable after the assignment operation(s) have been carried out and ranged from simple one line assignment operations such as:

```
int a = 10;
int b = 20;

a = b;
```

to more complex multi-line operations, such as:

```
int a = 5;
int b = 3;
int c = 7;

a = c;
b = a;
c = b;
```

By using multiple choice questions, Dehnadi (2006) was able to map each potential answer to a specific mental model. Originally, Dehnadi had predicted eight different mental models but three more were uncovered throughout the course of the experiment. The preliminary test was administered twice to first year undergraduate students: at the beginning of the course and after the students had been taught about variable assignment and sequences. Dehnadi (2006) revealed that after the first test, three groups of student responses were identified:

- Consistent – students used a single mental model to answer all (or most) of the questions, with 44% of students being classed as consistent.
- Inconsistent – students used several models to answer questions, with 39% of students being classed as inconsistent.
- Blank – students refused to answer the majority of the questions, with 8% of students being classed as blank.

Dehnadi (2006) goes on to state that the majority of students became consistent in their model usage after the second test. However, he does not provide any exact figures and focuses subsequent analysis on the results from the first test. As only 60 students were included in Dehnadi's (2006) preliminary study it is difficult to draw any statistically reliable conclusions. However, Dehnadi indicates that a clear "separation of populations" (Dehnadi, 2006, p. 29) can be observed when correlating the first test results against the official course results. Although a visual analysis of Dehnadi's (2006) results does indeed reveal a separation between the consistent and inconsistent/blank groups, no statistical tests are provided to corroborates it. Dehnadi (2006) also presents a correlation of the first test results against the official in-course exam but does not conduct any further analysis, stating that the more "complex picture" should not be analysed at this stage due to the small sample size (p. 29).

Despite the distinct lack of any in-depth statistical analysis, Dehnadi (2006) documents his testing and marking process thoroughly, allowing his experiment to be replicated relatively easily. Nevertheless, Dehnadi concludes by saying that he has developed a categorisation method that is "more likely to be used as a reasonable predictor of success in introductory programming" (Dehnadi, 2006, p. 35), claims which without further statistical analysis, appear premature at best. It should be noted that additional claims made by Dehnadi and Bornat (2006) regarding their aptitude test's ability to accurately predict students who are likely to fail their introductory programming module were later retracted by Bornat (2014). Subsequently, Dehnadi's (2006) original study has been replicated several times by different researchers with mixed results. Bornat et al. (2008) applied the test to 500 students across six institutions but their results indicated that the aptitude test failed to live up to the original expectations from the promising preliminary study. Bornat et al. (2008) stated that they failed to separate the "programming goats from the non-programming sheep" (p. 8) within their expanded investigation, although they believed that their results indicate further research into the consistency of mental models is warranted.

Additionally, a study by Caspersen et al. (2007) applied Dehnadi's test to approximately 300 students and was unable to find a correlation similar to that originally presented by Dehnadi (2006) and questions the viability of Dehnadi and Bornat's interpretation of their results, stating that their test instrument "does not measure what it is supposed to" (Caspersen et al., 2007, p.210). However, a study by Strnad et al. (2009) adds support for Dehnadi's aptitude

test being used as a predictor of success with students who have had no prior programming experience.

Dehnadi (2009) later revised his aptitude test design in response to the criticism from PPIG members by expanding the total number of models being examined from eight to eleven, as well as making the judgment for consistency more explicit and repeatable. Dehnadi et al. (2009) also conducted a meta-analysis using the refined test in an attempt to confirm the initial findings. The results appear to support claims of a relationship between consistent mental model usage and student performance but does not suggest any explanation for it.

Ultimately, Dehnadi's (2006) aptitude test design presents an intriguing method of examining the mental models of students, which has had some mixed success in predicting the abilities of students with no prior programming experience. Dehnadi's methods warrant further research, and subsequently inform the research questions at the heart of this study. However, the context in which Dehnadi's methods will be integrated into this investigation will be in a far less draconian context than an attempt to separate "programming goats from the non-programming sheep".

In addition to Dehnadi's work in analysing students' mental models of variable assignment, there have been a number of studies that demonstrate students' difficulties with developing appropriate mental models of core programming concepts through an examination of students' misconceptions. Although not all studies explicitly refer to mental models, the misconceptions students demonstrate can be a useful indication that the mental model a student holds of a given concept is either incomplete or inaccurate. For example, students' difficulties with understanding that a variable can only hold a single value that is not affected by the name of the variable have been uncovered (Grover & Basu, 2017; Kaczmarczyk et al., 2010; Sirkiä & Sorva, 2012). Subsequently, misconceptions relating to variable assignment, that is, believing that 5 = A and A = 5, have also been prevalent in a number of studies (Du Boulay, 1986; Ma et al., 2008; Qian et al., 2020; Qian & Lehman, 2017; Simon, 2011; Sirkiä & Sorva, 2012; Žanko et al., 2019, 2022), thus supporting Dehnadi's (2006) methodology. Furthermore, the fundamental nature of variables to programming means that students who struggle to develop an appropriate understanding will face greater difficulties when attempting to comprehend more complex topics such as iteration (Corney et al., 2011; Simon, 2011).

Pea and Kurland (1984) state that handling conditional statements (if statements) are a major part of programming and as such, it is reasonable to assume that a student who has sufficient understanding of conditional logic is more likely to succeed than a student who does not. To this end, a number of misconceptions have been identified which indicate students' difficulties with grasping conditional logic. These misconceptions include continuously monitoring the if statement throughout the program – identified by Pea (1986) as the parallelism bug. Or alternatively, believing that if the if statement condition evaluates to false, then the program will stop, or that if the condition evaluates to true, that both the if and the else blocks are executed (Sleeman et al., 1986, as cited in Qian & Lehman, 2017; Swidan et al., 2018).

Work by Grover and Basu (2017) identified students' difficulties with grasping Boolean operators such as AND and OR. Although most students answered questions involving the AND operator correctly, only half of students answered questions about the OR operator correctly, some exhibited a misconception where when both conditions are true the statement is evaluated to false. Grover and Basu (2017) explain this is an embodiment of the natural-language use of "or", as students believe only one of the conditions can be true, which is also equivalent to the XOR (exclusive or) condition. Consequently, Grover and Basu's (2017) XOR misconception is an embodiment of Clancy's (2004) linguistic transfer.

Misconceptions that demonstrate potentially inadequate mental models have also been identified for two related programming concepts, namely, iteration and recursion (Kessler & Anderson, 1986). Iteration, the simpler of the two concepts, involves repeating a block of code until a condition is met. However, students have been known to have difficulties identifying which lines of code are being repeated, as well as how many times the loop is repeated (Caceffo et al., 2019; Sleeman et al., 1986, as cited in Qian & Lehman, 2017) . In some cases students fail to recognise how the iterative loop affects the execution of the code (Eckert et al., 2022; Grover & Basu, 2017).

For example, the following “while loop” in C++ should produce an output of “0,1,2,3,4,5”:

```
int num = 0;
while (num <= 5)
{
      std::cout << num << ",";
      num++;
}
```

However, students may predict that the code will repeatedly produce the same output of “0,0,0,0,0,” as suggested by Grover and Basu (2017) or may simply not perform any iteration at all and produce an output of “0,”. Whilst it is possible that students have misunderstood how the variable “num” is being incremented in this example, it is reasonable to assume that their difficulties stem from an inadequate mental model of iteration, leaving them unable to comprehend that the code inside the while loop is being repeated, and that the value of “num” is being increased by one during each loop.

Recursion on the other hand, is a more complex concept, which Kahney describes as a “process that is capable of triggering new instantiations of itself, with control passing forward to successive instances and back from terminated ones” (Kahney, 1983, p. 235). A non-programming example of recursion is performing a factorial calculation. One of the most profound misconceptions amongst students relating to recursion is that they view a recursive function in the same way as they view an iterative loop (Götschi et al., 2003; Kurland & Pea, 1985). However, the complexities associated with recursion lend itself to varying interpretations by students.

Various studies have attempted to gain an insight into students’ interpretations of recursion, both in terms of identifying potential mental models to explain students’ understandings (Götschi et al., 2003; Kahney, 1983) and also examining how differing teaching methods can impact on students’ learning of the concept (Kessler & Anderson, 1986; Kurland & Pea, 1985; Wiedenbeck, 1989). Interestingly, Kessler and Anderson (1986) and Wiedenbeck’s (1989) studies suggest that by allowing students to develop an appropriate and reliable understanding of iteration prior to teaching them recursion eases their construction of

appropriate recursive mental models and reduces the likelihood of the student becoming overwhelmed.

In conducting a study into children's mental models of recursion, Kurland and Pea (1985) identified a number of "general bugs" that were causing students difficulty. These bugs included:

- Decontextualized interpretation of commands – Children carried out "surface reading" of programs, meaning they attempted to develop an understanding of each individual line of the program, thus ignoring the context provided by the previous lines. This is similar to the mental model identified by Dehnadi (2006) where students do not carry the changes made by assignment operations on to subsequent lines.
- Assignment of intentionality to program code – An embodiment of Pea's (1986) intentionality bug, whereby children did not differentiate the meaning of a command from the meaning of lines of commands they were expected to follow.
- Overgeneralization of natural language semantics – Children interpreted keywords within the LOGO programming language as having their natural language meanings, that is, STOP or END would completely stop the program from running rather than ending a statement.
- Overexaggerating of mathematical operators – Kurland and Pea (1985) describe how children expressed confusion when using numbers as inputs and when performing simple calculations, as well as how numbers were often seen as a source of discrepancies between the children's predicted execution of the program and the actual result. Kurland and Pea's (1986) identification of students' difficulties when mathematics is introduced into a program raises an interesting question about the relationship between mathematics and programming skill. Whilst some argue that students with a mathematical background are more likely to succeed within a programming course (Bergin & Reilly, 2005b; Gomes et al., 2006), others state that students' prior experience of mathematics, especially algebra, can lead to additional misconceptions such as assuming that a variable is only a representation of an unknown number, or the difference between assignment and equality operations (Grover and Basu, 2017).

- Mental model of embedded recursion as looping – As discussed previously, the children in Kurland and Pea's (1986) experiment had a fundamental misunderstanding of how the concept of recursion works, resulting in them viewing it in the same way as they view iteration.

Although Kurland and Pea's (1985) study was primarily focused around children's mental models of recursion, it does provide an interesting demonstration of a number of general mental models of programming that are likely to be detrimental to students' learning. There does, however, appear to be a distinct lack of common approach for categorisation and analysis of misconceptions and the subsequent inappropriate and inadequate mental models that are constructed across the entirety of the introductory programming syllabus. This can be seen in the different ways that mental models and misconceptions are identified and presented across many of the studies presented in this section, which in turn, makes comparisons between concepts more difficult.

There has been some effort to create a *concept inventory* for introductory programming by Kaczmarczyk et al. (2010) and Caceffo et al. (2016). However, more work is required to create and validate a more comprehensive list of misconceptions developed from a larger and more varied range of participants.

2.6 Summary

By delving into the literature surrounding the difficulties students face when learning to program, this chapter has provided support for the research questions at the heart of the investigation. There are clear indications from the literature presented throughout this chapter that the teaching and learning of programming is a complex process, where there is a significant potential for simple misunderstandings to have a profound impact on students' progression within their introductory programming module. Whilst designing specific pedagogic interventions is outside of the scope of this investigation, the issues raised within the literature, particularly relating to students' levels of cognitive load and the misconceptions that they can develop, highlight the need for direct support in order to address misconceptions by directly teaching the models students need to establish. As such, it is hoped that the outcomes of this investigation will be able to guide future work into the development and implementation of appropriate early interventions in order to support students with their mental model development and, as such, allow them to progress within their introductory programming course.

3. Investigation Methodology

3.1 Investigation Scope

As mentioned previously, there are two distinct parts to this investigation, with the first focusing on the development of the aptitude test, and the second being the development of the predictive model. The aptitude test acts as the main data collection mechanism for the investigation. The purpose of this chapter is to describe the design process underpinning the development of the aptitude test, including how pilot studies were used to further refine the test, as well as exploring how previous studies that have attempted to predict students' programming abilities have influenced this investigation. The second part of the investigation focuses on the development of the predictive model and is the focus of the next chapter.

Given the use of the aptitude test as a means of collecting data to answer the research questions at the heart of this work, as well as to support the development of the predictive model, this investigation is firmly seated within the realm of quantitative research. Therefore, an appropriate research paradigm must be identified in order to support the investigation design. Like the educational paradigms discussed in Section 2.2, it is necessary to specify the ontological and epistemological orientation of the research. As the investigation is quantitative in nature, Bahari (2012) states that Positivism and Objectivism are the appropriate epistemological and ontological orientations respectively.

The epistemological orientation of Positivism is centred around the empirical testing of hypotheses in a manner that is as independent and unbiased as possible (Bahari, 2012). Within positivist research, knowledge is gained through observations of reality, which allows for relationships to be established and integrated into theoretical models that can be used to make predictions (Bahari, 2012; Flowers, 2009). Furthermore, Objectivism is based around the premise that a reality can be established through the examination of relationships and although the *true* depiction of reality may never be established, researchers have the capacity to move closer to it through their investigations (Bahari, 2012). Consequently, Positivism and Objectivism are clearly appropriate for an investigation of this nature, given that the main focus is identifying factors that have significant relationships with students' programming abilities, which can later aid in the development of the predictive model.

As discussed in detail throughout this chapter, the aptitude test is designed to collect data on a number of factors that could potentially influence a student's success within the introductory programming module. As the aptitude test is issued to students twice, once at the start of their course, prior to any teaching taking place (T1), and once at the end of the first semester; approximately 12 weeks later (T2), it allows for a holistic review of students at the two timepoints. The aptitude test is, therefore, an embodiment of the Positivist research philosophy as it enables empirical evidence to be collated, which can be used to help in answering the three research questions which guide this work. Subsequently, the analysis of the data collected using the aptitude test aligns with the Objectivism philosophy, as I am attempting to establish how the factors examined within the test relate to students' performance within their introductory programming module, and by extension, attempting to develop a predictive model that can aid in the identification of students who are likely to require support.

3.2 Potential Factors for Inclusion in the Aptitude Test

3.2.1 Aptitude Test Rationale

There have been numerous studies that have attempted to predict students' programming abilities. A search conducted within the ACM Digital Library using the phrase "predicting student programming abilities", revealed over 500,000 results dating back to the 1970s, with a variety of different approaches being taken. For example, Simon et al. (2006) attempted to predict the programming abilities of students studying on an introductory programming module by using a series of cognitive tasks, including a paper folding test to evaluate their spatial visualisation and reasoning, map sketching to assess their design skills as well as their ability to make decisions based on these maps and searching a phonebook to assess their ability to form searching strategies. In addition, Simon et al. (2006) also used a questionnaire to explore the students' approaches to learning and studying. Similarly, Bergin and Reilly (2005a, 2005b) examined students' motivational and "comfort" levels using a questionnaire, which they believed would be able to predict students' performance within an introductory programming module. Alternatively, researchers such as Blikstein et al. (2014) and Watwin, Li and Goodwin (2013) took a more automated approach to predicting student performance by utilising various machine learning (ML) techniques to analyse data collected from students during their introductory programming modules.

Although the use of automated methods that track students' progress to produce predictions may be useful for ongoing assessment, this method of performance prediction would not be appropriate for attempting to identify students in need of support at the beginning of the course due to the time required to collect the necessary data. Similarly, conducting exams, whether these involve cognitive tasks or traditional closed-book style tests, needs a significant amount of time for the results to be processed (Bergin & Reilly, 2006), making this approach impractical. As such, it seems essential to develop a method of prediction that is easily automatable and that can readily be used to produce performance predictions at the beginning of an introductory programming module.

Dehnadi's (2006) notion of using an aptitude test, despite the criticisms discussed in Section 2.5, was seen as an appropriate starting point for this investigation, as an online aptitude test would allow for easy distribution to students as well as relatively quick analysis once the appropriate software is developed. It was felt that an online aptitude test, combined with a statistical model capable of predicting students' performance, would be a powerful tool for Computing educators as it enables students who are likely to need additional support to be identified early on the course without the need for time-consuming manual analysis.

The main focus of the aptitude test draws on Dehnadi's (2006) original methodology of exploring students' misconceptions of fundamental programming concepts. The initial design of the test included an adaptation of some of Dehnadi's (2006) variable swapping questions, as well as questions which assess students' understanding of other fundamental concepts such as conditional logic and Boolean operators (AND, OR and NOT), iteration and recursion. In addition to these concepts, students' comprehension of program output statements and flow of compilation was also assessed. For each question, students were required to trace a simple program that assessed one or more of the concepts listed above and answer a question about its output.

As the aptitude test was designed to be used online and at the beginning of an introductory programming module, no assumptions could be made about students' prior knowledge. Therefore, students were only required to read and comprehend the code within the questions, and not write any code for themselves. This approach to testing is supported by Lister et al.'s (2004) belief that the ability to write code relies upon the ability to read code and, as such, is appropriate for use at the beginning of the academic year, where it cannot be assumed that

students have any relevant prior programming experience. Indeed, such students may become overwhelmed when tasked with writing a program, in a similar way to when students experience “programming shock” when they first attempt to write programs for real. Additionally, limiting the aptitude test to questions that only relate to reading code helps to improve the scope for automating the testing process, as expected answers (both correct and incorrect) can be predetermined. Allowing students to write their own code would require a significant amount of analysis, and whilst it might be possible to develop an algorithm capable of analysing students’ code, this was deemed to be beyond the scope of the present research programme. Furthermore, all question code was written using pseudocode based on the OCR GCSE computer science guide (OCR, 2015), which minimises the amount of potentially unfamiliar syntax that is employed, and allows students to logically deduce the answers to questions, even if they have never programmed before. The online format of the aptitude test also allowed for a number of factors to be explored in addition to students’ understanding of fundamental programming concepts. Following on from the literature presented in the previous chapter, the following factors were considered for inclusion within the aptitude test.

3.2.2 Students’ Previous Experience

The mental models that students construct are influenced by what they believe to be relevant prior knowledge (Ben-Ari, 2001; Sorva, 2013). It is therefore important to establish an understanding of students’ previous experiences, as students at university level are likely to come from a wide range of backgrounds and have a variety of prior knowledge, which could potentially be a help or a hindrance when learning to program. An obvious starting point for examining students’ previous experiences is to determine whether they have had any prior programming experience, including whether they have studied computer science before and also whether they consider themselves to be self-taught. Identifying students who have had prior experience with programming is important because if it is to be believed that it takes 10 years for a novice programmer to become an expert (Winslow, 1996), students who have been exposed to programming will have begun to construct their own mental models of fundamental programming concepts.

However, it is also important to understand the context in which students have previously been learning to program. For example, if a student has been learning to program without the

support of a teacher, then it is possible that they have constructed mental models that include unrecognised misconceptions. Moreover, although these misconceptions may not currently be hampering the student, they may cause issues for them during their subsequent studies. Furthermore, Dehnadi's (2006) test was able to work reasonably well on students who had no prior programming experience but did not work at all for students with previous experience. Therefore, it is important to identify and explore the differences between students with or without prior programming experience, as this may need to be accounted for by the predictive model.

In addition to prior programming experience, students' mathematical abilities, as explored throughout Chapter 2, have been cited as a potential influencing factor in students' programming capabilities. Ideally, a mathematical aptitude test could be run alongside the programming aptitude test to provide an independent evaluation of students' mathematical abilities; however, the development of such a test was considered to be beyond the scope of the current research programme. Nevertheless, it is important to identify students who have studied mathematics, or other math-based subjects such as Physics or Engineering, as they potentially have stronger mathematical skills than students who have not studied such subjects.

Another important consideration that must be taken into account when evaluating students' misconceptions is the fact that not all students' first language will be English. As the majority of programming languages utilise English-like keywords such as "print" in Python (Veerasamy & Shillabeer, 2014), there is an increased potential for misconceptions, similar to that of linguistic transfer described in Section 2.5 and, as such, this factor should be taken into consideration when developing the predictive model.

3.2.3 Students' Mental Characteristics

One potential method of predicting students' programming abilities; which has had reasonable success (Bergin & Reilly, 2005a, 2005b; 2006; Quille & Bergin, 2018; Wilson & Shrock, 2001), is by using a student's own estimation of their abilities, including their beliefs of how they are performing or will perform within their introductory programming module. The term "comfort level" has been used to represent a series of variables that are indicative of a student's level of anxiety surrounding a programming course. These variables include: a

student's ease in asking questions in class and during one-to-one sessions with the tutor; a student's anxiety level while working on assignments; a student's perceptions of the difficulty of the course; a student's perceived understanding of concepts compared to classmates; and a student's perceived difficulty in completing assignments (Bergin & Reilly, 2005b; Wilson & Shrock, 2001). These factors can be combined to produce a single continuous variable for use in comparative analysis against students' performance.

Studies by both Bergin and Reilly (2005a, 2005b) as well as by Wilson and Shrock (2001) have revealed comfort level to be a significant and relatively powerful predictor of student performance, although comfort level questions were ultimately not included within the final predictive model developed by Bergin and Reilly (Bergin & Reilly, 2005a, 2005b; 2006; Quille & Bergin, 2018). However, some factors that are used to calculate the comfort level score, such as students' levels of anxiety when asking questions in class, would not be appropriate for use in an aptitude test, which would be administered prior to any teaching. Nevertheless, students' initial beliefs regarding how difficult the course and learning to program are going to be may, in fact, be indicative of their subsequent programming performance as students' fear of programming has been shown to form a "very real, almost physical barrier that causes intense emotions, a loss of confidence" and ultimately results in a block in students' learning (Rogerson & Scott, 2010, p.167).

Interestingly, Curzon and Rix (1998) revealed that one of the major motivations for students wanting to learn to program at the beginning of their courses was their desire to become a professional programmer. However, as students progress through their course, the proportion wanting to become a professional programmer dwindles. Additionally, Bergin and Reilly (Bergin & Reilly, 2005a) reveal that intrinsically motivated students who are motivated by the satisfaction that they can gain from performing well in their course, show increased levels of performance as opposed to extrinsically motivated students who are primarily motivated to complete tasks by the rewards they can gain, or to avoid punishment.

Although comfort level has been shown to be a strong predictor of programming performance, a number of the factors it examines, such as students' anxiety levels when answering questions, can only be measured once a student has been studying the course for a period of time, and are therefore, not appropriate for an aptitude test designed to be initially distributed to students at the beginning of their course. It should be noted, however, that some

factors that are examined within the comfort level rating, such as how apprehensive students feel about programming, would be applicable to an aptitude test. A potentially more appropriate metric for use within an aptitude test is students' "self-efficacy", which is a representation of their own judgments of their capabilities (Bandura, 1977, 2006). Bandura states that a person's self-efficacy can influence the activities they choose and how much effort they exert, their level of persistence when faced with a problem as well as their overall performance level (Bandura, 1977; Ramalingam & Wiedenbeck, 1998). Recent research has also revealed significant relationships between students' levels of self-efficacy and factors including their course performance, levels of emotional engagement, occurrence of misconceptions (Kallia & Sentance, 2019; Kanaparan et al., 2019; Tek et al., 2018), thus making it an appropriate metric for developing a predictive model of students' performance.

As self-efficacy is not a personality trait that can measured by generic tests (Bandura, 1977, 2006; Ramalingam & Wiedenbeck, 1998), it is essential that a self-efficacy scale that is specific to introductory programming be used when evaluating students' programming abilities. One such scale that has seen widespread use is Ramalingam and Wiedenbeck's (1998) "Computer Programming Self-Efficacy Scale", which has been cited 300 times at the time of writing and has been employed in various studies ranging from investigations of students' computer anxieties (Doyle et al., 2005) to evaluating alternative pedagogic approaches and assessment styles for introductory programming courses (Sharmin et al., 2019; Ventura & Ramamurthy, 2004; Zingaro, 2014). Ramalingam and Wiedenbeck's (1998) Computer Programming Self-Efficacy Scale consists of 32 questions originally written to be used to evaluate self-efficacy in students studying object-oriented C++. The questions in Ramalingam and Wiedenbeck's (1998) scale vary in complexity, with answers being recorded using a 7-point scale ranging from 1 (not at all confident) to 7 (absolutely confident).

In order to validate their scale, Ramalingam and Wiedenbeck (1998) performed an investigation with 421 students enrolled on an introductory computer science course. The scale was administered to students twice, once during the first week of the course, to establish students' "pre-self-efficacy", and also after the thirteenth week of the course, to establish "post-self-efficacy". Ramalingam and Wiedenbeck (1998) reported a Cronbach's alpha score of 0.98 for the first administration of the scale and 0.97 for the second administration, which therefore indicates that their scale is a highly reliable measure of students' programming self-

efficacy. They go on to report significant gains in self-efficacy between the two tests and suggest that students with the lowest levels of self-efficacy are likely to be those who have had no prior programming experience, or who have had previous bad experiences with programming and therefore may exhibit higher levels of apprehension and fear towards programming, which as Rogerson and Scott (2010) state, can cause a barrier to students' learning.

Ramalingam and Wiedenbeck's (1998) scale was later used by Wiedenbeck et al., (2004) when examining how students' performance in a course is affected by their previous programming experience (or lack thereof), their self-efficacy, and the mental models that they hold relating to programming. Wiedenbeck et al. (2004) hypothesized that that pre-self-efficacy should not affect programming performance directly but should instead affect performance *indirectly* through its effect on post-self-efficacy. Using a sample of 75 students studying introductory C++, Wiedenbeck et al. (2004) identified that self-efficacy increased significantly during the course, thus supporting Ramalingam and Wiedenbeck (1998) original findings.

Wiedenbeck et al. (2004) also conducted a Path Analysis that revealed that students' self-efficacy prior to teaching (pre-self-efficacy) influences course performance through post-self-efficacy (the measurement taken after completion of teaching), which acts as a mediator variable. In addition to self-efficacy, the strength of a student's mental models was also found to have an effect on their course performance, with strong mental models increasing a student's self-efficacy due to an increase in program comprehension. Additionally, it was determined that prior experience of programming is a strong predictor of both pre- and post-self-efficacy.

The apparent relationship between mental models and self-efficacy as described by Wiedenbeck et al. (2004) would suggest that Ramalingam and Wiedenbeck's (1998) scale would be potentially beneficial when used in conjunction with the proposed aptitude test design in the present research. However, some modifications may need to be made to bring the scale in line with the proposed language-independent philosophy being taken, as has been done previously by Zingaro (2014) who modified Ramalingam and Wiedenbeck's (1998) original scale to fit the context of his investigation into peer instruction. The Computer

Programming self-efficacy scale has also been modified to be used to assess students' self-efficacy within an introductory algorithms course (Danielsiek et al., 2018).

Ramalingam and Wiedenbeck's (1998) scale provides a measurement of students' levels of self-efficacy associated with programming in general. An alternate approach, which was posed by Duran et al. (2019), focused on measuring students' own estimations of their understanding of fundamental programming concepts. In their study, conducted with students studying an online course in introductory programming, Duran et al. (2019) found that through multiple administrations of their online self-evaluation form at different stages within the course, it was possible to identify students growing in confidence as they gained more experience through practice, with differences in confidence being reported between the concepts. Duran et al. (2019) go on to suggest that there is an 'overlap' between self-evaluation and self-efficacy, which is evident from the increase in students' perception of their programming related abilities as the course progresses. There is, however, need to further validate Duran et al.'s (2019) tool outside the context of an online course.

An additional area of interest which is believed to be related to self-efficacy (Luxton-Reilly et al., 2018) stems from Dweck's (2000) notion of 'mindsets' in relation to how a student believes they can grow and develop, with two categories being identified – fixed and growth (Cutts et al., 2010; Dweck, 2000). Students who hold a fixed mindset that ability (in this context, programming ability) is fixed, will likely give up if they cannot do something as they believe that it is not possible to change their abilities (Cutts et al., 2010; Morales-Navarro et al., 2023). On the other hand, students who hold a growth mindset believe that their abilities can change through practice, and view failure as an opportunity to grow through feedback, suggesting higher levels of resilience and self-efficacy (Cutts et al., 2010; Morales-Navarro et al., 2023; Tek et al., 2018).

The difficulties of learning to program can often lead students to adopt a fixed mindset given that there are so many ways a student can become stuck (Cutts et al., 2010; Luxton-Reilly et al., 2018; Murphy & Thomas, 2008; Simon et al., 2008). Like low levels of self-efficacy, a fixed mindset can directly impact on a student's performance within their introductory programming module, which is reflected in students' levels of self-efficacy being associated with the mindset that they possess (Cutts et al., 2010; Morales-Navarro et al., 2023; Quille & Bergin, 2020; Tek et al., 2018).

Cutts et al. (2010) devised an intervention to support students in developing a growth mindset in order to improve their performance in their introductory programming module. The results of the study revealed an increase in performance amongst students receiving mindset training, which was also confirmed in a similar investigation by Quille and Bergin (2020) designed to re-validate Cutts et al.'s (2010) work. However, Quille and Bergin (2020) highlighted that mindset training promoted a growth mindset for some students, and a fixed mindset for others, suggesting that implementing mindset-based interventions within introductory programming modules may be more complex than first thought.

While conducting a series of observations with school students studying introductory programming, Perkins et al. (1986) identified a behaviour amongst students that may be indicative of their level of confidence with programming. The behaviour Perkins et al. (1986) uncovered relates to students' responses when faced with a problem with their program that does not immediately have an obvious solution. Students' behaviour broadly fell into one of two categories: "stoppers" or "movers" (Perkins et al., 1986). When faced with a problem, a student who is classified as a stopper will often feel at a complete loss as to what to do to try and produce a solution and will inevitably give up at attempting to find a solution (Perkins et al., 1986). Perkins et al's (1986) description of a stopper is highly similar to previous descriptions of a student who is holding a fixed mindset (Murphy & Thomas, 2008; Simon et al., 2008). On the other hand, when faced with the same problem, a student who is classified as a mover will attempt to solve the problem by trying one solution after another until they eventually find the correct one (Perkins et al., 1986).

Although the mover category may appear to be the more positive of the two, Perkins et al. (1986) identify a further subset called "extreme movers", which relates to students who attempt to fix code in ways which would clearly not work if the student thought carefully about what they were doing. Students who are classified as extreme movers are, in essence, trying to develop programming solutions by brute force rather than trying to come up with a logical solution, which will probably result in them going round in circles. Perkins et al. (1986) also explain that students who are classified as stoppers tend to feel unsure about what they are doing when attempting to write programs, and harbour such significant levels of fear that, in some cases, they have essentially given up trying to learn to program. In addition, while extreme movers may have higher levels of motivation than stoppers, extreme movers

tend to become emotionally distant from the task and often try to move on to the next task in order to avoid an issue they simply cannot fix.

Although Perkins et al. (1986) do not perform any analysis of the relationship between students' behaviour when faced with an issue and course performance, it is reasonable to assume that students' who are classified as stoppers or extreme movers are more likely to perform worse as both neglect to address any issues or misconceptions that they develop. Further research is required to identify what factors can be used to predict whether a student is likely to be a stopper, a mover or an extreme mover. However, Ramalingam and Wiedenbeck's (1998) Computer Programming Self-Efficacy scale was considered to be useful for the present research programme as it provides an insight into students' estimation of their own abilities.

An additional measure that may provide insight into programming efficacy is Cacioppo and Petty's (1982) concept of "Need For Cognition". Need For Cognition is explained by Cacioppo and Petty (1982) as being "the tendency for an individual to engage and enjoy thinking" (p. 116), which potently makes it useful for determining students who are likely to become stoppers or extreme movers (Perkins et al., 1986), as both of these categories relate to students exerting less mental effort in attempting to develop solutions to problems. The Need For Cognition scale originally advanced by Cacioppo and Petty consisted of 34 questions, which were validated over a series of four separate studies (Cacioppo & Petty, 1982), with a shortened version of the scale consisting of only 18 questions later being produced and confirmed as equally valid to the original 34 question scale (Cacioppo et al., 1984). This shortened scale lends itself perfectly for use within an aptitude test, and although Need For Cognition has not previously been applied in the context of introductory programming, it may provide additional insight into students' tendencies to seek out answers to problems, which in turn may help predict course performance.

An additional factor that has previously been seen to be directly related to cognitive task performance is cognitive load (Morrison et al., 2014). Cognitive load is discussed in detail in Section 2.4, however, it can be summarised here as the amount of load that performing a particular task poses on a person's cognitive system (Sweller, 1994). Paas and Van Merriënboer (1994) describe two primary techniques for measuring cognitive load, that is, psychophysiological indices, such as pupil diameter and heart rate, and subjective indices,

which primarily take the form of mental effort rating scales. Naturally, the psychophysiological indices would not be appropriate for an aptitude test designed to be given to all first year computer science students upon entry to their course. Instead, previous research has shown that a 9-point Likert scale ranging from very, very low mental effort (1) to very, very high mental effort (9) is a highly reliable measurement of mental effort and in turn, cognitive load (Paas et al., 1994), making it a viable factor for inclusion within the present aptitude test design.

3.2.3 Working Memory Capacity and Spatial Ability

It has been suggested that because of its complex nature, programming poses a high demand on students' working memory and therefore increases the probability of cognitive overload (Yousoof et al., 2007; see Section 2.4). Due to the widespread application of the concept of working memory capacity, ranging from measures of intelligence to expanding theories of Alzheimer's disease and reading disabilities (Redick et al., 2012), various techniques have been developed to measure a person's working memory capacity. For example, in their seminal paper, Engle et al. (1999) presented a number of standardised techniques for measuring working memory capacity, which included the following:

Reading Span (SPAN) – Participants are presented with sentences that have an unrelated capitalised word at the end. Participants are tasked with reading the sentence and then the capitalised word aloud. When the participant reads the capitalised word aloud the screen is immediately changed to the next sentence-word combination. This sequence is repeated between two and six times (with three trials of each size) until three question marks are displayed, at which point participants are required to recall and write down all of the capitalised words which have been displayed in the set in the correct order. An example set can be seen below.

For many years, my family and friends have been working on the farm. SPOT
Because the room was stuffy, Bob went outside for some fresh air. TRAIL
We were fifty miles out at sea before we lost sight of the land. BAND
???

After participants record the words, they are asked a random comprehension question such as “Did Bob go outside?”. Participants with comprehension scores less than 85% are removed from any further testing.

Counting Span (CSPAN) – Participants are shown a display of randomly arranged dark blue circles, dark blue squares and light blue circles. Participants are tasked with counting the total number of dark blue circles aloud and repeating the digit corresponding to the final tally. For example, when there are three dark blue circles a participant should say “one, two, three, three”. The light blue circles and dark blue squares act as distractors. When the participant repeats the final tally, the display is changed to the next set. After between 2 and 8 sets, the message RECALL is displayed on the screen, and the participant must recall and write down all of the tallies which have been displayed since the last RECALL prompt in the correct order. The number of target shapes (dark blue circles) varies between three and nine for each set, whilst the number of distractor variables also varies with there being one, three, five, seven or nine dark blue squares and between one and five light blue circles being displayed. Any participant whose error rate is greater than 15% is removed from any further testing.

Operation Span (OSPAN) – Participants are presented with individual operation-word strings and a math problem which they must read aloud. For example, “Is (8/4) - 1 = 1?” The participant should answer aloud if the equation is correct or not by answering “yes” or “no”. The participant then reads aloud an additional word (e.g., “bear”), which prompts the next operation string to be displayed. Similar to the Reading Span method, this sequence is repeated between two and six times, with three trials of each. At the end of each set, three question marks are displayed, prompting participants to recall and write down the words that followed the operation strings in the correct order. An example set can be found below.

Is (8/4) - 1 = 1? bear

Is (6 X 2) - 2 = 10? beans

Is (10 X 2) - 6 = 12? dad

???

Participants who have an error rate on the equations greater than 15% are removed from further testing.

Although these popular techniques proposed by Engle et al. (1999) offer a relatively simple method for measuring working memory capacity, they would not be appropriate for use within an aptitude test in their current form due to the fact that they require students to read aloud. Furthermore, as these measures of working memory capacity take at least an hour to administer (Engle et al., 1999) they would greatly increase the time needed to complete the aptitude test and are therefore unsuitable for this investigation.

An alternative measurement technique that is potentially more suited to use within an aptitude test is the Corsi Block Test (Corsi, 1973). Kessels et al. (2000) describe the Corsi Block Test as a widely used measurement of visuospatial short-term memory, which is a component of working memory. As there are several variations of the Corsi Block Test that all differ in their implementation, a standardised testing procedure is provided by Kessels et al. (2000) and is as follows. A set of 9 black cubes measuring 30 x 30 x 30 mm are mounted on a black board (225 x 205 mm). Each cube has a number between one and nine printed on its side so only the examiner can see it. The examiner taps blocks in a sequence, initially consisting of two blocks, which the participant then repeats. A second sequence of the same length is then demonstrated, and the participant is again given the opportunity to repeat it. If the participant successfully repeats at least one of these sequences, then the next two trials of sequences of an increased length are administered. Cubes are tapped at a rate of approximately one cube per second, and the test is terminated if the participant fails to reproduce two sequences of equal length (Kessels et al., 2000). Kessels et al. (2000) also note that the Corsi Block Test can either be completed forwards, where the participant repeats the sequence in the same order as the examiner, or backwards with the participant reversing the examiner’s sequence, although this has been suggested to place additional load on the participant’s working memory (Claessen et al., 2015). The relative simplicity of the Corsi Block Test, combined with its short administration time and the fact that it has already been adapted into an electronic format (Berch et al., 1998; Claessen et al., 2015; Vandierendonck et al., 2004) makes it a potentially useful measure for inclusion within the aptitude test.

In addition to working memory capacity, another cognitive factor that has been found to have an apparently strong relationship with programming skill, which could potentially be evaluated within the aptitude test, is students' spatial ability (Jones & Burnett, 2008; Margulieux, 2020; Parkinson, 2022; Parkinson & Cutts, 2018). Halpern (as cited in Jones & Burnett, 2008) defined spatial ability as being a measure of a person's ability to conceptualise the spatial relations between objects. It is believed that spatial ability is an important factor in determining a person's capacity to comprehend programs due to the fact that skills similar to that of navigation are required to visualise program operations (Cox et al., 2005).

Students' spatial abilities can be measured by means of them performing a series of mental rotation tasks (Jones & Burnett, 2008; Vandenberg & Kuse, 1978), such as the mental rotation tests posed by Vandenberg and Kuse (1978) and Shepard and Metzler (1971). Vanderberg and Kuse's (1978) test was originally designed to be conducted on paper and requires participants to identify two correct rotations of a 3D object from a list of four possible rotations. The test consists of 20 items in five sets of four and requires both correct rotations to be identified in order to be marked as correct. Shepard and Metzler's (1971) mental rotation test is somewhat simpler than Vanderberg and Kuse's test (1978) as only two 3D objects are displayed side-by-side, with the original implementation stipulating that participants should pull levers to indicate whether the two images are of different objects, or if they are of the same object in different rotations (Shepard & Metzler, 1971). The simplistic nature of both tests lend them well to being converted into computerised versions such as Strong's (2000) implementation of Vanderberg and Kuse's (1978) test and Wright et al.'s (2008) version of Shepard and Metzler's (1971) test, both of which are viable candidates for inclusion within the aptitude test.

3.3 Predictive Model Considerations

Although the main focus of the second phase of this investigation is the development of a predictive model to answer RQ 3, it is important that potential approaches are considered prior to the development of the aptitude test, which acts as the primary data collection mechanism for the model. Two closely related fields of research whose methodologies and techniques are appropriate for developing a model capable of identifying students who are likely to require support are Learning Analytics (LA) and Educational Data Mining (EDM).

Learning Analytics is seen to be a new and expanding field, which utilises expertise from computer science, sociology and psychology to develop and apply predictive models that provide actionable information to educators, allowing them to tailor curriculums and educational interventions to support both the individual learner and cohorts as a whole (Avella et al., 2016; Siemens, & Baker, 2012; Siemens, 2012). Similar, to Learning Analytics, Educational Data Mining is also a relatively new area of research, which brings together researchers from computer science, learning science and psychology amongst others (Siemens & Baker, 2012), and is also focused on developing a better understanding of students and how they learn by exploring the unique types of data that are produced in education settings (Avella et al., 2016). However, EDM places a greater focus on automated discovery, with the models that are produced often being used within automated systems such as intelligent tutoring systems (Siemens & Baker, 2012). Siemens and Baker (2012) provide a brief comparison of both EDM and LA, which is presented in Table 3.1.

Table 3.1

Comparison between Learning Analytics and Educational Data Mining (derived from Siemens & Baker, 2012)

Focus	Learning Analytics	Educational Data Mining
Discovery	Leveraging human judgment is key; automated discovery is a tool to accomplish this goal.	Automated discovery is key; human judgment is a tool to accomplish this goal.
Reduction and Holism	Stronger emphasis on understanding systems as wholes, in their full complexity.	Stronger emphasis on reducing to components and analysing individual components and relationships between them.
Origins	Stronger origins in semantic web, "intelligent curriculum", outcome prediction and systemic interventions.	Stronger origins in educational software and student modelling, with a significant community in predicating course outcomes.
Adaptation and Personalisation	Greater focus on informing and empowering instructors and learners.	Greater focus on automated adaption (e.g., by the computer with no human in the loop).
Techniques and Methods	Social network analysis, sentiment analysis, influence analytics, discourse analysis, learner success prediction, concept analysis, sensemaking models.	Classification, clustering, Bayesian modelling, relationship mining, discovery with models, visualisation.

Bienkowski et al. (2014) provided a useful summary of the differences between EDM and LA by stating that the focus of EDM is on the development of new tools for discovering patterns, whereas LA focuses on applying the tools and techniques at scale. Consequently, the outputs of each field differ somewhat, with EDM being positioned to answer questions such as "What sequence of topics is most effective for a specific student?" or "What student actions indicate satisfaction, engagement and learning progress?", whereas LA is most suited to answering questions such as "When are students falling behind in a course?" or "What grade is a student likely to get without intervention?" (Bienkowski et al., 2014).

Although both EDM and LA are interested in predicting students' performance in some shape or form and as such use broadly similar data analysis techniques, there are some techniques that are predominately restricted to their respective fields. For example, Social Network

Analysis is used within LA to examine the relationships between instructors and learners in order to identify influencers or disconnected students (Avella et al., 2016; Bienkowski et al., 2014). Alternatively, Discovery with Models is a concept primarily used in EDM to support in-depth analysis of different factors (e.g., intelligent tutoring system design, types of student behaviour, etc.), by developing a model of a phenomenon through processes such as prediction or knowledge engineering, which can then be validated and fed into further analysis (Baker & Yacef, 2009).

Despite there being some techniques that are field-specific, LA is able to take advantage of many of the popular EDM techniques when analysing large datasets (Avella et al., 2016). The techniques used in EDM often differ from traditional data mining techniques due to the need to account for (and where possible exploit) the multi-level hierarchy and non-independence in education-related data and, as such, psychometric-based models are not uncommon in EDM research (Baker, 2010; Baker & Yacef, 2009). Baker (2010) provides a comprehensive overview of the techniques commonly used within EDM, which are set out below.

Prediction

The aim of predictive models is to be able to estimate reliably a value (predicted variable) by using a combination of other variables in the dataset (predictor variables; Avella et al., 2016; Baker, 2010). Baker (2010) lists the three main types of prediction that are used within EDM as being classification, regression, and density estimation. Classification is used when attempting to predict binary or categorical data, with common methods including decision trees, logistic regression and support vector machines (Avella et al., 2016; Baker, 2010). Classification techniques are not suitable for predicting continuous variables therefore, regression methods such as linear regression, neural networks, and support vector machine regression are employed (Baker, 2010). Finally, density estimation can be used to predict both continuous and categorical variables using a probability density function (Baker, 2010).

Clustering

Clustering is used to split datasets by identifying datapoints that naturally group together, which makes it useful when analysing datasets without predefined categories (Avella et al., 2016; Baker, 2010). Baker (2010) describes how clustering algorithms such as K-Means (with randomised restart) can start either with no pre-existing hypothesis or can take into

account hypotheses that have been developed through prior research, such as with the Expectation Maximization algorithm, thus giving researchers flexibility in their analyses.

Relationship Mining

Relationship mining is used to uncover relationships between variables in large datasets (Avella et al., 2016; Baker, 2010). Baker (2010) describes how different techniques can be used to identify different types of relationships within the data. One such technique is association rule mining, which is used to identify "if-then" relationships based on the premise that if a specific set of variable values is found, another variable will have a particular value, with Baker (2010) giving an example of such as rule:

{student = frustrated AND student goal of learning > student goal of performance} → {student frequently asks for help}

Correlation mining, on the other hand, attempts to identify any positive or negative linear correlations within the data, whereas sequential pattern mining attempts to find temporal associations between events, such as what type of behaviour leads to a student losing interest in a course (Baker, 2010). The final method that Baker (2010) lists is casual data mining. This is focused on attempting to find out whether one event was caused by another, which is determined by either measuring their covariance or by using information about how each of the events is triggered.

Discovery with Models

As noted above, Discovery with Models is utilised within EDM research to develop models using automated methods such as prediction and classification, or in some cases, using human judgment through knowledge engineering. The resulting models are then applied in further analysis (Avella et al., 2016; Baker, 2010).

Distillation of Data for Human Judgment

Distillation of Data for Human Judgment offers EDM researchers an alternative to purely automated analysis, as by displaying data with appropriate visualisation techniques, it is possible for humans to make valid inferences that may have otherwise been missed by automated techniques (Avella et al., 2016; Baker, 2010). This method of analysis is used

within EDM for either identification or classification, with Baker (2010) giving the example of visualising students' performance on a learning curve.

The development of prediction models has been the focus of many EDM investigations (Baker, 2009). However, the nature of the present research programme limits some of the techniques that can be reliably applied. This is due to the relatively small amount of data that are available for analysis, as data will be collected from voluntary participation in aptitude tests as opposed to the log files obtained from intelligent tutoring systems that are used by entire classes, which are commonly deployed in EDM research. Nevertheless, a reliable, predictive model developed around students' responses to an aptitude test specifically designed for evaluating their potential for grasping the fundamental concepts of programming should still be possible.

In both the fields of EDM and LA, numerous methods have been developed to evaluate and predict students' programming performance. A popular approach has been to analyse snapshots of students' code that are collected each time they attempt to compile their programs. Among others, this approach was taken by Blikstein et al. (2014) and Fernandez-Medina et al. (2013), who attempted to analyse the snapshots to shed light on students' behaviour during their introductory programming modules. Additionally, code-snapshot analysis has been incorporated into the development of algorithms designed to evaluate students' performance, such as Jadud's Error Quotient (Jadud, 2006) and the Watwin Algorithm (Watson et al., 2013).

The Error Quotient evaluates students' performance by producing a score based on the number of persistent errors (i.e., errors that are still present after the student is first confronted with the error) within their programs. Students' progress is tracked through an extension to the BlueJ IDE on a university computer and an overall score between 0 and 1 is computed by averaging the scores from their successful compilations during the session (Jadud, 2006). Jadud (2006) notes that a score of 0 does not represent a student making no mistakes, rather, it indicates that a student's program did not contain the same mistake for more than one compilation in a row. However, a score of 1 would indicate that every compilation included an error.

The Error Quotient was initially found to be a weak predictor of performance (Jadud, 2006). However, Rodrigo et al. (2009) claim to have been the first to demonstrate that the Error Quotient can successfully predict students' achievement on their midterm exam, when using a more constrained dataset as opposed to the one used in Jadud's (2006) original study. Additionally, Rodrigo et al. (2009) believe that students with high Error Quotient scores may be exhibiting Stopper or Extreme Mover tendencies as defined by Perkins et al. (1986). Although the apparent success of the Error Quotient is encouraging, Watson et al. (2013) noted a significant methodological flaw in Jadud's (2006) approach. Watson et al. (2013) state that Jadud's (2006) approach is predicated on the assumption that students only work in a single source file or if they are using multiple source files, this is done so linearly. Watson et al. (2013) claim that students do not work this way as they often switch between files, thus compromising Jadud's (2006) approach.

Watson et al. (2013) present their own "Watwin" algorithm, which incorporates a scoring system that penalises students based upon how long it takes them to resolve specific types of error compared to their classmates. Like the Error Quotient (Jadud, 2006), the Watwin algorithm analyses compilation snapshots. However, the Watwin algorithm constructs a set of compilation pairs for each file a student has worked on during the session, with compilation events associated with each file being ordered by timestamp, thus accounting for any students working on multiple files simultaneously (Watson et al., 2006).

When compared using the same dataset, the Watwin algorithm significantly outperformed the Error Quotient when attempting to predict students' performance, with the Watwin algorithm explaining 30% of the variance in performance on average, and 42.4% by the end of the course, compared to 14% and 19%, respectively, for the Error Quotient (Watson et al., 2013). Watson et al. (2013) also applied their pairing pre-processing method to the Error Quotient, thus negating the methodological deficiencies, which resulted in a slight increase in the R^2 value, albeit with this value still being much smaller than the values obtained from the Watwin algorithm.

Other approaches to predicting programming abilities within the fields of EDM and LA have utilised methods such as neural networks, support vector machines, decision trees and clustering techniques such as K-medoids, which is a variant of the K-means clustering

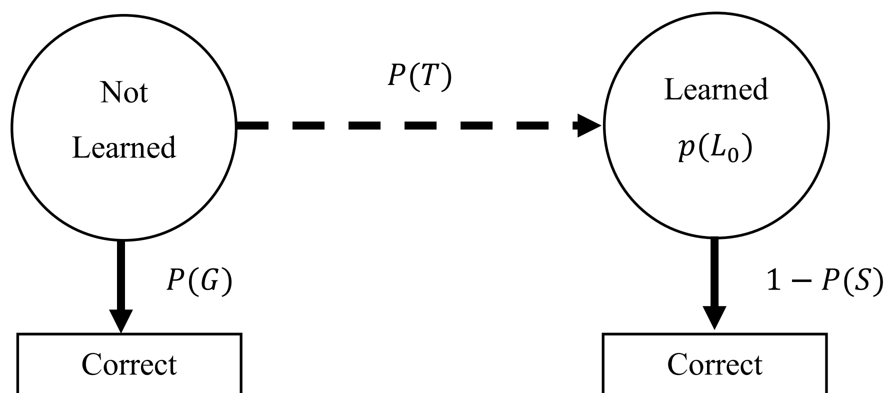
algorithm (Britos et al., 2008; Costa et al., 2017; ElGamal, 2013; Piech et al., 2012). Additionally, a method that features heavily within EDM research into Intelligent Tutoring System design – despite predating the field by over a decade – is Bayesian Knowledge Tracing (Baker et al., 2008; Corbett & Anderson, 1994), which could potentially be adapted for use in the design of a predictive algorithm.

Corbett and Anderson's (1994) Bayesian Knowledge Tracing (BKT) originated from their work with their Cognitive Tutor, which was used to teach Lisp and Prolog in introductory programming classes at Carnegie Mellon University, as well as to teach Pascal at a Pittsburgh high school. BKT was introduced in response to a number of students floundering during the course, in order to provide a means of monitoring students' changing states of knowledge while they are practicing a particular topic (Corbett & Anderson, 1994).

BKT examines students' knowledge as a latent variable and assumes that skills (knowledge components) can be represented in a binary fashion by being either mastered (learned) or not (not learned; Corbett & Anderson, 1994; Yudelson et al., 2013). This approach to modelling students' knowledge can be represented as a Hidden Markov Model as seen in Figure 3.1 (Baker, 2020).

Figure 3.1

Bayesian Knowledge Tracing Hidden Markov Model



Note. From "Big Data and Education" by R. Baker, 2020, University of Pennsylvania

BKT utilises four parameters in order to model students' knowledge (Baker, 2020; Corbett & Anderson, 1994):

- $P(L_0)$ – the initial probability that a student knows a particular skill.
- $P(G)$ – the probability that a student does not know a skill but guesses the answer correctly.
- $P(S)$ – the probability that a student does know a skill but has made a small error (a slip).
- $P(T)$ – the probability that a student will learn the skill and transition from the not learned state to the learned state. This is assumed to be a constant value.

As students practice each skill the estimated probability of them knowing the skill, $P(L)$, is updated by first calculating the probability of whether the student knew the skill before answering the question using either Equation 3.1, where they answered correctly, or Equation 3.2, where they answered incorrectly, and then accounting for the possibility that the student has learned the skill whilst completing the task using Equation 3.3 (Baker et al., 2008).

3.1) $$P(L_{n-1}|Correct_n) = \frac{P(L_{n-1})*(1-P(S))}{P(L_{n-1})*\left(1-P(S)\right) + \left(1-P(L_{n-1})\right) * (P(G))}$$

3.2) $$P(L_{n-1}|Incorrect_n) = \frac{P(L_{n-1})*(P(S))}{P(L_{n-1}) * P(S) + \left(1-P(L_{n-1})\right) * (1-P(G))}$$

3.3) $$P(L_n|Action_n) = P(L_{n-1}|Action_n) + \left(\left(1 - P(L_{n-1}|Action_n)\right) * P(T)\right)$$

The initial values for the four parameters can be established using the Expectation Maximization algorithm (Pardos & Heffernan, 2010). However, work by Beck and Chang (2007) has revealed that different combinations of the four parameters can fit the data equally well yet yield significantly different predictions. They termed this issue the *identifiability* problem.

Beck and Chang (2007) proposed an alternative method for fitting the parameters using Dirichlet Priors in order to overcome the identifiability problem, although issues have also been found with this approach. Baker et al. (2008) demonstrated that Beck and Chang's

method (2007) suffers from *model degeneracy,* which is where guess (G) and/or slip (S) parameters are initialised to values greater than 0.5 and are, therefore, deviating from the theoretical concept of BKT, which is that correct performance generally implies that a student knows the relevant skill. By having a guess parameter greater than 0.5 it is implied that a student who does not know a skill is more likely to get the answer correct than incorrect. Likewise, a slip parameter greater than 0.5 also implies that a student who knows the skill is more likely to get the answer incorrect than correct (Baker et al., 2008). Baker et al. (2008) proposed an extension to BKT that overcomes both the identifiability problem and model degeneracy, by contextually estimating whether a student has guessed or slipped using the responses to the two subsequent questions (n+1 and n+2) in order to provide a more comprehensive evaluation of the student's response at time n. Baker et al., (2010) also proposed a brute-force approach to fitting the four BKT parameters, which they believe allows for better fits to be achieved for the parameters than were possible with previous approaches. In order to avoid issues relating to model degeneracy, parameter estimates for guess and slip were bounded to be below 0.3 and 0.1 respectively (Baker et al., 2008). Further extensions to BKT have also been proposed, such as student-specific parameter values (Yudelson et al., 2013) or the inclusion of additional parameters to account for students' forgetting material either immediately after being taught it or on separate days, as traditional BKT assumes students do not forget a skill once it has been learnt (Qiu et al., 2011). BKT has also been adapted from its original intended use with Intelligent Tutoring Systems for use with Massive Open Online Courses (MOOCs), which do not facilitate real-time processing of students' responses and therefore require adaptations to be made to BKT in order to evaluate students' learning (Pardos et al., 2013).

Despite these latter extensions, traditional BKT is still the most popular implementation within Intelligent Tutoring Systems (Yudelson et al., 2013), as many of the extensions often work well with certain datasets, but not others (Baker, 2020). Although BKT is not designed for use with aptitude tests, Pardos et al. (2013) have already demonstrated that it can be adapted for use in contexts other than Intelligent Tutoring Systems. BKT is likely to offer a useful insight into students' understandings of fundamental programming concepts when analysing their responses to the aptitude test. It can be used to provide an estimate of the probability that a student holds an appropriate mental model for a given concept, thus allowing for how students' mental models develop, as per RQ 1, to be analysed. The

estimates produced by BKT are also ideal for inputting into a predictive model in order to represent students' understandings of core programming concepts.

3.4 Aptitude Test Design

3.4.1 Section Outline

Drawing on the original methodology posed by Dehnadi (2006), the aptitude test designed for this investigation was intended to be used for both the research data collection that would later be used for development of the predictive model, and also as a means for identifying future students who are likely to need support in combination with the proposed model. As such, it was important to consider the limited amount of time available for students to complete the test and also what platform should be used to distribute the test to students when deciding on what factors to include from Section 3.2.

During the research investigation the aptitude test was designed to be issued twice to first year Computing students at the University of Central Lancashire (UCLan), once at the start of their course and once at the end of the first semester (September – December). This would allow for students' progress to be analysed alongside their introductory programming module, which runs during the first semester.

In order to allow greater flexibility with question design and to ease distribution and result collation, it was decided to distribute the aptitude test to students online, as opposed to Dehnadi's (2006) paper-based method. The survey platform Qualtrics (http://www.qualtrics.com/) was chosen due to the fact that custom HTML questions can be created, thus allowing for measures of working memory and spatial ability (discussed later in this section) to be easily integrated into the test.

This section presents an overview of the aptitude test design process, which is informed by the research questions at the heart of this investigation, and how it has been validated prior to commencement of the primary data collection.

3.4.2 Initial Aptitude Test Design

The initial version of the aptitude test was devised to assess the suitability of various question designs in order to ensure the appropriateness of the questions and to evaluate whether it is feasible for students to complete the aptitude test within a one-hour timeslot, as whilst

participation was voluntary, time was made available for students to complete the aptitude test. However, if the aptitude test was found to be too long, students may become disengaged or may not be able to fully complete it within the available time. Therefore, the initial version of the aptitude test was trialled with all components that were being considered for inclusion in the final test, which could then be refined as required. As such, the initial version of the aptitude test consisted of the following sections:

Section 1: Student Details
This section collates a number of factors relating to the student's background, which could potentially assist with predicting a student's performance, and aids in answering RQ 2. These factors included:

- Gender
- If the student had previously studied computer science (or computing) at any level (yes/no)
- If the student had studied any post-16 mathematics-based subjects such as mathematics, engineering, physics, etc. (yes/no)
- Whether the student had any prior programming experience (yes/no)
- Whether the student considered themselves to be a "self-taught programmer" (strongly agree – strongly disagree)
- Whether English was the student's first language. (yes/no)

A number of additional questions were also included within this section, which draw from prior research into students' motivation and comfort levels (Bergin & Reilly, 2005a, 2005b; Curzon & Rix, 1998):

- If the student intends on working in software engineering/programming after graduating from university (yes/no/undecided)
- How difficult they expect their degree to be (1 – 10)
- How difficult they expect learning to program will be (1 – 10)
- Whether they fear learning to program (yes/no)

Students' university ID numbers were also recorded in order to allow for their aptitude test results to be compared to their introductory programming module grades. It is important to stress that participation in this study was optional, and all responses are anonymised in accordance with the ethical approval obtained for this research. Students were incentivised to take part by being able to be entered into a prize draw for one of five £10 Amazon gift cards, as well as being able to receive feedback on their answers.

Section 2: Modified Programming Self-Efficacy Scale
The second section of the aptitude test consists of a slightly modified version of Ramalingam and Wiedenbeck (1998) Computer Programming Self-Efficacy Scale.
The original scale analyses students' programming self-efficacy using a series of 32 questions relating to object-orientated C++, with a focus on "meaningful programming tasks: designing, writing, comprehending, modifying and reusing programs" (Ramalingam & Wiedenbeck, 1998, p. 369). Given its widespread use (Zingaro, 2014), Ramalingam and Wiedenbeck's (1998) scale provides a firm foundation for assessing students' self-efficacy levels related to programming in general. However, Bandura (2006) states, one measure does not fit all scenarios, meaning a number of small modifications are required to make Ramalingam and Wiedenbeck's (1998) scale suitable for use within the aptitude test.

Ramalingam and Wiedenbeck (1998) partitioned their original scale questions into four factors:

Factor 1: Independence and persistence
Factor 2: Complex Programming Tasks
Factor 3: Self-Regulation
Factor 4: Simple Programming Tasks

Given that the aptitude test is focused towards assessing students' fundamental programming skills, it was decided to omit the Complex Programming Task (Factor 2) questions as the majority of the questions bared no relevance to the objectives of the aptitude test. Additionally, several of the Factor 2 questions related to object-orientated programming, but as this is not currently taught until much later in the introductory programming module that students are studying, it would be inappropriate to measure their self-efficacy on this topic. The remaining questions were presented in the same order as Ramalingam and Wiedenbeck's

(1998) original scale, with one minor alteration being made through the removal of any specific references to C++ and replaced with "any programming language" in order to make the aptitude test language independent. The full list of questions can be found in Section 2 within Appendix A.

Additionally, Ramalingam and Wiedenbeck (1998) provided the following instructions to students when completing the scale, which have been replicated in this study, without referring to a specific programming language.

> Rate your confidence in doing the following C++ programming related tasks using a scale of 1 (not at all confident) to 7 (absolutely confident). If a specific term or task is totally unfamiliar to you, please mark 1. (Ramalingam & Wiedenbeck, 1998, p.6)

The inclusion of the modified Programming Self-Efficacy scale in the aptitude test allows for students' confidence associated with programming to be examined directly, thus providing support in answering RQ 2.

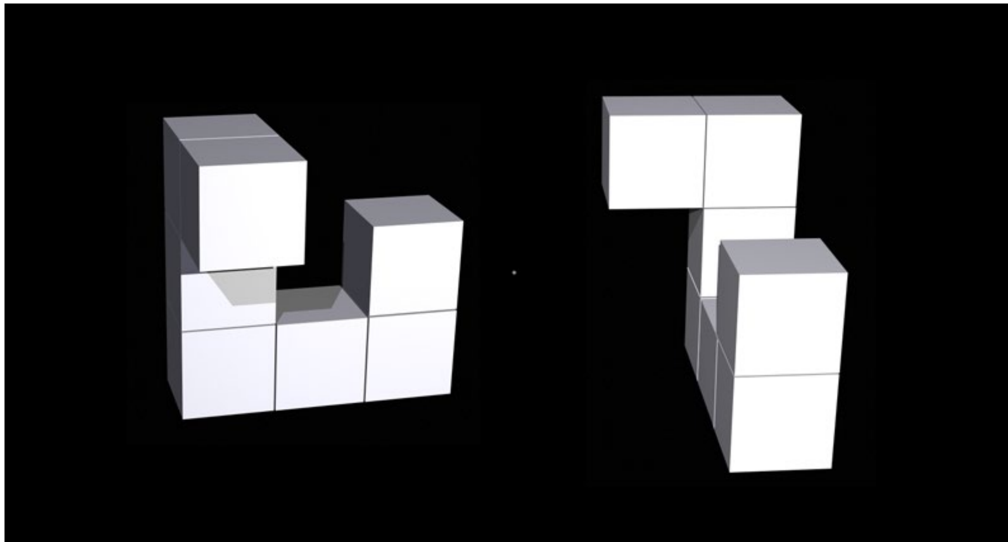
Section 3: Spatial Ability Measurement
Given the purported links between students' spatial abilities and success within their programming courses (Jones & Burnett, 2008), two different styles of mental rotation test designs were considered for inclusion within the aptitude test based on those used by Vandenberg and Kuse (1978) and Shepard and Metzler (1971), respectively. Unfortunately, due to a lack of availability of high-resolution versions of the testing materials, Vandenberg and Kuse's (1978) version of the mental rotation test was unable to be implemented within the aptitude test.

A mental rotation test in the style of Shepard and Metzler's (1971) original test was created using Ganis and Kievit's (2015) 3D object dataset. The mental rotation test was built using JavaScript in order to allow it to be integrated into the Qualtrics environment. The test followed Ganis and Kievit's (2015) procedure by displaying the 3D objects side-by-side, to which respondents must indicate if the images are of the same object (see Figure 3.2 for an example) by pressing the "B" key on their keyboard, and the "N" key if they are different. Students had a maximum of 7.5 sec (as specified by Ganis and Kievit, 2015), to provide an answer for each of the 48 trials, of which half included the same image in different rotations.

Trials occurred in a random order with no more than three trials of four possible rotations of a given object occurring consecutively. Unlike Ganis and Kievit (2015), a second block of 48 trials was not included due to the limited amount of time being available for students to complete the aptitude test.

Figure 3.2

Example of Objects Used in Mental Rotation Test



Before students started the mental rotation test, they were presented with written instructions specifying what the task involved, what keys to use and that they had 7.5 sec to provide an answer for each pair of images. They were then given the opportunity to perform twelve practice trials using images that were not included in the main test and were given feedback as to whether they had answered correctly or incorrectly. Upon completion of the mental rotation test the student's error rate was passed back as an embedded variable using the Qualtrics JavaScript API (Qualtrics, n.d.). Due to the nature of the Mental Rotation Test, any student who specified they required a screen reader or other visual aid was permitted to skip this task.

After completing the Mental Rotation Test students were prompted to record how much mental effort they felt was required to identify which of the pairs of images were the same shape using a 9-point Likert scale ranging from very, very low (1) to very, very high mental effort (9), as this has previously been shown to be a reliable measurement of mental effort (Paas et al., 1994). This allowed for the identification of any significant relationships that

exist between the measurement of students' spatial abilities, the amount of mental effort required and their programming abilities, thereby indicating students who are more likely to experience cognitive overload.

Section 4: Need For Cognition Scale

As discussed previously, the Need For Cognition Scale, both in its original and revised formats (Cacioppo et al., 1984; Cacioppo & Petty, 1982) has not previously been applied to the context of introductory programming. However, the abstract nature of programming requires students to be actively seeking out solutions to problems, and there is a risk that students who are unable to do so, the stoppers and extreme movers (Perkins et al., 1986), will be at a greater risk of falling behind in their course. The Need For Cognition Scale could potentially identify students who are likely to become stoppers or extreme movers, which links to RQ 2, hence the inclusion of the revised eighteen question version (Cacioppo et al., 1984) in the aptitude test.

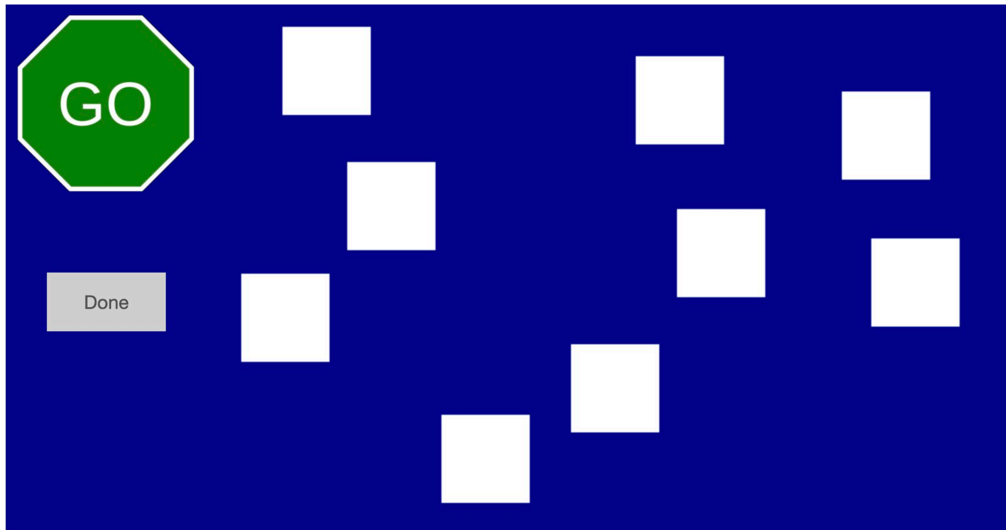
Section 5: Working Memory Capacity

Given the reported demands that programming places on a student's working memory (Yousoof et al., 2007), a custom implementation of the Corsi Block Test (Corsi, 1973) was used to measure students' Working Memory capacities. The Corsi Block Test was chosen due to its relative simplicity, ease of administration and short administration time. A number of electronic versions of the Corsi Block Test already exist (Berch et al., 1998; Claessen et al., 2015; Vandierendonck et al., 2004), which were used to guide its development in the present research.

The Corsi Block Test implementation was constructed using JavaScript in order to allow it to be embedded within the Qualtrics environment. Following Vandierendonck et al.'s (2004) description of their implementation, nine white blocks were placed in the approximate standardised locations as dictated by Kessels et al. (2000), whilst allowing for differences in screen size and resolution, on a dark blue background (see Figure 3.3).

Figure 3.3

Corsi Block Test Implementation Screenshot



The test sequence was triggered by the student pressing a button labelled "Start" on the left-hand side of the window. This initiated a five sec countdown being displayed after which, the first sequence was presented to the student. As in Vandierendonck et al. (2004), blocks turned black for 1 sec with an inter-block time of 0.5 sec and with each block turning black for 0.2 sec after being clicked. However, instead of using a sound to signify the end of the presentation, a green "GO" sign was used to prompt the student to replicate the sequence, as headphones/speakers could not be guaranteed to be available for every student completing the test. A student indicated that they had finished replicating the sequence by clicking a button labelled "Done" on the left-hand side window. If the student entered the correct sequence a green "Correct" sign was displayed in the bottom-left portion of the window and a red "Incorrect" sign was displayed if an incorrect or incomplete sequence was entered. There was a gap of 2 sec between the student signifying they had completed the sequence and the start of the next presentation.

Before beginning the Corsi Block Test students were presented with instructions on how to complete the test and then were given a chance to practice with randomly generated sequences consisting of two and three blocks. After completing the demonstration students began the actual test, which used sequences from lengths of three to eight. As per Kessels et al. (2000), students had the chance to replicate two sequences for any given length. If the student successfully replicated at least one of these sequences, they were allowed to proceed

to the next sequence length. If neither of the sequences was replicated the test ended. Only the forward direction version of the Corsi Block Test was implemented within the aptitude test due to time restrictions.

Once the student had completed the Corsi Block Test, the Qualtrics JavaScript API (Qualtrics, n.d.) was used to pass the student's score (the last completed sequence length) as an embedded data variable, allowing it to be included with the student's responses to the other sections in the overall test. After completing the Corsi Block Test, students were also asked to record how much mental effort they felt was required to complete the task in order to attempt to identify students who are likely to experience cognitive overload (Yousoof et al., 2007). Any students who specified that they required screen readers or other visual aids were able to skip this section in addition to the Mental Rotation Test.

Section 6: Programming Diagnostic
The Programming Diagnostic portion of the aptitude test posed a series of questions on a number of key topics that students will encounter as part of their introductory programming module. These questions were designed to identify any misconceptions held by students and would subsequently allow for estimates of how likely they are to be holding appropriate mental models for each of the concepts being examined using Bayesian Knowledge Tracing (see Section 4.3). The questions within the Programming Diagnostic can therefore be seen to be supporting both RQ 1 and RQ 3.

The main topics covered in the Programming Diagnostic within the initial version of the aptitude test included: Variable Assignment, Conditional Statements, Iteration and Recursion. However, additional areas that may cause misconceptions were also examined alongside the main topics, including Program Flow (parallelism misconception), Output Statements, and whether the names of variables affect what they can hold. The questions for each of the main topics were contained within their own sub-sections, which also included questions on how much mental effort students felt was needed to answer the questions as well as how many they felt they had answered correctly.

In order to ensure that the aptitude test is language-independent, pseudocode based on the OCR GCSE Pseudocode Guidelines (OCR, 2015) was used for all questions within the aptitude test, as the test places more emphasis on students' abilities to logically deduce

answers rather than their understanding of the syntax of a particular language. The variable assignment sub-section was comprised of six questions derived from Dehnadi's (2006) original study. The questions included both single assignment operations (Figure 3.4) and multiple assignment operations (Figure 3.5), thus allowing for the examination of misconceptions relating to the direction of the assignment (=) operator and also students' understandings of how variables function.

Figure 3.4

Example of Single Assignment Operation Question

The variables 'A' and 'B' are initialised in the lines of code below.

```
A = 10
B = 20
```

What are the values of 'A' and 'B' after carrying out the following operation?

```
A = B
```

Figure 3.5

Example of Multiple Assignment Operations Question

The variables 'A', 'B' and 'C' are initialised in the lines of code below.

```
A = 5
B = 3
C = 7
```

What are the values of 'A', 'B' and 'C' after carrying out the following operation?

```
A = C
B = A
C = B
```

Although the assignment questions themselves are similar to those of Dehnadi (2006), students were not presented with a list of answers to choose from in the way Dehnadi did. Instead, students were prompted to input their own values for each of the variables after the statement had been executed.

As Pea and Kurland (1984) state, conditional statements in the form of "if" statements are a major part of programming that, if misunderstood, are likely to cause students' significant difficulties when writing programs of their own. As such, the second sub-section consisted of eight multiple choice questions which were designed to highlight whether a student is holding misconceptions relating to "if" statements, Boolean operators (AND, OR and NOT) (Grover & Basu, 2017) and parallelism (Pea, 1986). Simple hints were included to explain any elements of syntax that students might have been unfamiliar with.

Figure 3.6

Example of "If" Statement Question

What is the output of the following code?

Hint: '>=' represents 'Greater than or equal to'

```
Module1 = 30
Module2 = 20
PassMark = 100

Total = Module1 + Module2

if Total >= PassMark then
     print 'You have passed with a combined score of: '
     print Total
else
     print 'You have failed'

Module3 = 60
Total = Module1 + Module2 + Module3
```

For example, Figure 3.6 would have the following options:

A) You have passed with a combined score of: 50
B) You have passed with a combined score of: 110
C) You have failed
D) There would be no output

Any student who selected either C or D would be demonstrating a potential inability to correctly trace the execution of an "if" statement. Additionally, students who selected D may also not fully understand the function of output statements, which in this scenario, are indicated with the keyword "print". Students who selected either A or B are demonstrating that they can correctly trace an "if" statement, with option A being the correct answer. However, if a student chose option B, they would be demonstrating that they potentially hold the parallelism misconception by not recognising that programs flow linearly from top to

bottom. Following on from the questions on conditional statements, the two subsequent sub-sections examine students' understandings of iteration in the form of "for" and "while" loops. Misconceptions relating to the concept of iteration were identified using multiple choice questions, with two questions focusing on "for" loops and two focusing on "while" loops, as shown in Figures 3.7 and 3.8.

Figure 3.7

Example of "For" Loop Question

What is the output of the following code?

Hint: '++' increments a variable by 1

Hint: '<' represents 'Less than'

```
for i = 0; i < 4
      print i
      i++
```

Figure 3.8

Example of "While" Loop Question

What is the output of the following code?

Hint: '++' increments a variable by 1

Hint: '<=' represents 'Less than or equal to'

```
i = 0

while i <= 5
      print i
      i++
```

The available answers for both “for” and “while” loop questions account for students failing to recognise that code contained within the loop is repeated as well as whether a student misidentifies when a loop should begin or terminate. The questions on iteration also allow for students’ understandings of the flow of control within the program to be examined, for example, the subsequent question to that shown in Figure 3.8 is exactly the same but the statement “i++” is placed above “print i” instead of below it as shown in Figure 3.8. If a student correctly understands the flow of control, then they will recognise that i is now being incremented before being outputted and as such, produces an answer of “1 2 3 4 5 6” as opposed to “0 1 2 3 4 5” for the original question. Students who hold the parallelism misconception will fail to recognise the difference and, assuming they correctly understand iteration, would answer “0 1 2 3 4 5” for both questions.

A single recursion question was included in the aptitude test, as shown in Figure 3.9. Recursion is an advanced topic not covered within the introductory programming module that students taking part in this experiment were studying. However, it was included within the aptitude test to investigate claims of its relationship with iteration (Kessler & Anderson, 1986).

Figure 3.9

Example of Recursion Question

What would the result of the following function be when n = 4?

Hint: A function is a block of code which can be re-used without the need for it to be rewritten.

Hint: ‘<=’ represents ‘Less than or equal to’

Hint: ‘return’ is used to exit the function at a specific point and pass back the result. i.e. ‘return 3’ will exit a function with a result of 3

Hint: ‘*’ represents multiplication

```
n = 4

function fun1(n)
      if (n <= 1)
            return 1
      else
            return n * fun1(n-1)
```

Following on from the question shown in Figure 3.9, the same question was posed to students using a “for” loop, thus allowing their answers for both iteration and recursion to be compared. Students were also asked whether they found either the iteration or recursion question easier to answer, the mental effort required, as well as which (if either) question they felt they answered correctly.

In order to assess the aptitude test design a pilot study was conducted with first year Computing students. Thirty six students volunteered to take part and although this sample size is too low to draw strong conclusions, it did allow for the questions and proposed analysis technique to be evaluated.

3.4.3 Subsequent Modifications

A number of changes were made to the aptitude test following the first pilot study, the most notable being how students’ misconceptions were to be analysed. The original intended

analysis method was based on Dehnadi's (2006) mental model approach, where the intention was to an establish students' "dominant mental model" for each concept; essentially how the student usually approaches a particular topic. However, it was felt that this approach was too restrictive as it does not take into account students who may be exhibiting more than one misconception. Instead, it was decided to measure the frequency that each misconception is demonstrated by a student, thus allowing for a more complete picture of students' difficulties to be established. Bayesian Knowledge Tracing can be used to assess the likelihood of students holding appropriate mental models for each concept by examining the misconceptions students demonstrate on a question by question basis (see Section 4.3). A list of all misconceptions examined as part of the final version of the aptitude test can be found in Appendix B.

It should be noted that it is possible for a student to exhibit more than one misconception in a single question. For example, in a question focusing on variable assignment, such as the question shown in Figure 3.5, a student can exhibit misconceptions relating to the direction of the assignment operator, as well as not correctly carrying values forward when performing multiple assignment operations.

In addition to the change in analysis approach, a number of changes to the aptitude test were identified as being required. These included replacing the "fear of programming" yes or no option with a 1 to 10 scale to allow for a more detailed review of how much students fear learning to program. It was revealed that students were spending a significant amount of time completing the Mental Rotation Test and as such it was decided to remove it from the aptitude test to allow more time to be allocated to identifying students' programming misconceptions.

As a result of the removal of the Mental Rotation Test the number of variable assignment questions was able to be expanded from six to nine, and also included questions which examined whether students mistakenly believe, through a misconception which is similar in nature to Pea's (1986) Intentionality bug, that the variable names affect the values they can hold. For example, a student could mistakenly believe that a variable called "largest" will always hold the largest value (Kaczmarczyk et al., 2010; Qian & Lehman, 2017) as shown in Figure 3.10. Each of the questions that examines whether a student believes variable names influence the values they can hold has an identical counterpart question, which uses single

character variable names, therefore, allowing for students' answers to be compared in order to establish whether students hold this misconception.

Figure 3.10

Example of Variable Assignment with Inferred Meaning Variables Question

The variables 'smallest', 'middle' and 'largest' are initialised in the lines of code below.

```
smallest = 1
middle = 8
largest = 11
```

What are the values of 'smallest', 'middle' and largest after carrying out the following operation?

```
largest = smallest
middle = largest
smallest = middle
```

The pilot study also revealed that the multiple-choice conditional statement questions were potentially too easy, as almost all students answered the questions in this section correctly. Therefore, it was decided to make the questions open-ended rather than multiple-choice to prevent students' responses being influenced by the presented options, as well as to allow any new misconceptions to be identified. The number of conditional statements and iteration questions were expanded to take advantage of the removal of the Mental Rotation Test, with questions on iteration also being presented as open-ended to allow for additional misconceptions to be uncovered.

A second pilot study was conducted in order to evaluate the alterations to the aptitude test. Again, the pilot study was open to all first year Computing students, as well as second year students studying the Advanced Programming module. Students in the second year were able to choose the modules that they were studying so, as such, weaker programming students were likely to avoid this module. However, it was decided that opening up the pilot study to

second year students would allow for a more comprehensive evaluation of the aptitude test, as such students should be more familiar with the concepts being examined, and would also enable further insight to be gained into the prevalence of the different misconceptions.

As in the first pilot study, students were offered the chance to be included in a prize draw for one of five £10 Amazon vouchers, as well as being offered feedback on their answers. In total 29 first year students and 20 second year students took part and although the low participant number limits the nature of any detailed analyses, it was noted that second year students predictably displayed less misconceptions relating to variable assignment and conditional statements than first year students. However, the majority of the first year and a small number of second year students demonstrated difficulty with iteration questions, suggesting this is likely to be a troublesome topic.

The second pilot study revealed a number of additional changes that needed to be made to the aptitude test before the commencement of the actual data collection, the most notable being the need for the aptitude test length to be reduced further in order to allow it to be completed well within an hour timeslot. This was achieved by slightly reducing the overall number of programming question and the removal of the Need For Cognition scale and the Corsi Block Test.

Upon examination of the results from the pilot study, a significant moderate correlation was observed between the Need For Cognition Scale and the modified Computer Programming Self-Efficacy scale $r_s = .49$, $p < .001$, $N = 49$. A Spearman's rank coefficient was utilised for this analysis as although responses on the Need For Cognition scale were revealed to be normally distributed by a Shapiro-Wilk test ($W(49) = 0.97$, $p = .330$), scores on the modified Computer Programming Self-Efficacy scale were not normally distributed ($W(49) = 0.91$, $p = .007$). Despite being the longer of the two scales, the modified Computer Programming Self-Efficacy scale explicitly measures factors relating to students' programming abilities and has previously been shown to be a reliable predictor of students' performance, whereas the Need For Cognition scale is a general scale that has not previously been applied in the context of programming. This made it appropriate for the modified Computer Programming Self-Efficacy scale to be retained in the aptitude test and for the Need For Cognition scale to be removed.

Additionally, the Corsi Block Test was also removed due to the amount of time required for students to complete it and a lack of variation in the data. However, the data collected during the pilot study would appear to validate this implementation of the Corsi Block Test for use in a future study as the block span results (M = 6.41, SD = 0.99) are comparable to Kessels et al.'s (2000) results (M = 6.20, SD = 1.30).

Further to the need to reduce the overall length of the aptitude test, a number of other changes were also identified as being necessary. First, a number of questions on iteration required small modifications to allow them to be more easily mapped to the misconceptions set out in Appendix B. Furthermore, the conditional statement questions were redeveloped to allow for a more direct evaluation of students comprehension of the AND, OR and NOT operators using multiple choice questions based on Grover and Basu's (2017) assessment design, as shown in Figure 3.11.

Figure 3.11

Example of Boolean Operator Question

Which of the following words starts with a ‘d’ OR ends with an ‘e’?

Select all words this applies to.

☐ dance

☐ delicious

☐ soccer

☐ share

A minor alteration was also made to the pseudocode style used within the questions, with the inclusion of “braces” to denote the scope of statements. Although this is a departure from the OCR guidelines (2015), it was decided that their inclusion would make the questions more comprehensible, particularly those including iteration as shown in Figure 3.12. Hints were also removed in favour of making the code easier for students to logically deduce the meaning of statements, for example, by replacing i++ with i = i + 1.

Figure 3.12

Example of Addition of "Braces" to Questions

Examine the following code.

What would be outputted on the screen when it is run?

```
i = 0

while i <= 5 {
      print i
      i = i + 1
}
```

Additionally, it was decided to remove the recursion question from the aptitude test due to the fact the majority of students struggled with this topic that is beyond the scope of first year programming, which consequently made it difficult to accurately map students' answers to specific misconceptions. The questions included within the final version of the Programming Diagnostic section can be found in Section 3 within Appendix A.

In previous versions of the aptitude test, questions had been grouped together by topic. However, students commented that this made the test feel overly repetitive and therefore, questions on different topics were mixed together to negate this. Additionally, students were asked to rate how confident they were that they had answered each question correctly using a 0 – 100 scale to provide a more comprehensive evaluation of levels of confidence for each concept to aid in answering RQ 2. This is as opposed to using techniques such as Duran et al's (2019) self-evaluation instrument, which focuses on specific concepts, as students' confidence will be measured throughout the aptitude test, as discussed below. Given these changes it was necessary to move the mental effort questions to the end of the aptitude test, with students being asked to rate the amount of mental effort they felt was required to answer questions on each of the concepts. Examples of questions for each of the concepts were provided to aid students in making their estimations.

Aside from the alterations made to the Programming Diagnostic portion of the aptitude test, a question was added to Section 1 asking students to rate how difficult they find mathematics

using a 1 – 10 scale, given the purported link between programming abilities and mathematics (Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Wilson & Shrock, 2001). The aptitude test was also renamed as the "Programming Checkup" to alleviate any negative connotations towards the word "test" from the students.

Due to the limited number of students who took part in the two pilot studies, it is not possible to draw any statistically reliable conclusions from the results. However, a number of key findings within the data encouraged the commencement of the main data collection once the changes discussed above had been made. These included how having prior programming experience appeared to improve student confidence and also corresponded to a lower number of misconceptions being demonstrated, although previously studying computer science did not appear to have the same effect. Students who had studied a mathematics-based subject after finishing school also appeared to be more confident in their abilities. Furthermore, there was evidence to suggest a relationship between students' levels of confidence; particularly their Self-Efficacy levels, and the number of misconceptions that they demonstrate when answering questions. As has been mentioned previously, students appear to have significant difficulties understanding the concept of iteration (recursion also was identified as causing significant difficulty, which was why it was ultimately removed from the Programming Checkup), regardless of whether students had prior programming experience or not. Second year students who took part in the second pilot study demonstrated significantly less misconceptions that first year students, which is to be expected given they have had more time to develop appropriate mental models of the concepts, although misconceptions associated with iteration were still present in some students' responses. This provides an indication of students' mental model development progressing over time, the assessment of which, is incorporated into the main data collection process.

Commencing in September 2019, for a period of three years, the Programming Checkup was presented to all first year Computing students at the University of Central Lancashire (UCLan) during the first week (T1) of their degree through an introductory video that set out the purpose of the study, what the Programming Check Up involved, and how they could take part. It was stressed to students on multiple occasions that participation in the research investigation was optional, with the chance to be entered into a draw to win one of five £10 Amazon vouchers as well as being able to receive feedback on their answers serving as encouragement for taking part.

The Programming Checkup was also released to students towards the end of the first semester (T2), which was approximately 10 weeks after T1, thus allowing for students' progress to be evaluated over the course of the semester in a similar fashion to Ramalingam and Wiedenbeck's (1998) pre and post self-efficacy tests. Students were, again, offered the chance to win an Amazon voucher, as well as to receive feedback on their answers.

A full account of all questions included within the version of the Programming Checkup used for data collection is presented within Appendix A. Subsequently, an analysis of students' responses to the Programming Checkup is discussed within Chapter 5.

3.5 Overview of Machine Learning Algorithms

As mentioned previously, regression is a technique commonly used to estimate a numerical outcome (i.e., a dependent variable), based on the values of one or more independent variables (Maulud & Abdulazeez, 2020). Therefore, predicting students' assessment grades is a task naturally suited to regression (Nasiri et al., 2012; Strecht et al., 2015; Tomasevic et al., 2020; Wakelam et al., 2020) as it provides staff with a tangible estimation of what result the student is likely to achieve.

An alternative approach to predicting an exact mark for a student using regression techniques, is to predict a categorical outcome using a branch of machine learning referred to as "Classification" methods (Dietrich et al., 2015; Kotsiantis, 2007). The outcomes predicted by classification methods can be either binary or multinomial (Russell & Norvig, 2020; Tomasevic et al., 2020) and as such, it is possible to develop models to predict whether a student passes or fails an assessment (binary classification), or the grade band they are likely to achieve (multinomial classification) (Castro-Wunsch et al., 2017; Tomasevic et al., 2020). Although classification lacks the granularity of regression, the categorical output reduces the need for interpretation of the results by teaching staff, which could be beneficial when directing students towards support interventions. As such, binary classification methods were chosen to be evaluated as opposed to multinomial methods in order to maximise the interpretability of the outputs.

In sum, both regression and classification approaches to predicting students' Assessment 1 results were considered in the present research in order to allow for future pedagogic interventions to be developed that may benefit from either the clear-cut nature of binary classification, or the more granular predictions produced by regression models.

As this is the first time that the different factors examined by the Programming Checkup have been brought together to produce a prediction of students' abilities, it was felt that a wide selection of classification and regression methods should be explored. By drawing on previous research within the fields of Educational Data Mining and Learning Analytics (Baker & Yacef, 2009; Jacob et al., 2016; Rastrollo-Guerrero et al., 2020; Romero & Ventura, 2010; Strecht et al., 2015), a comprehensive analysis and assessment could be undertaken to establish what machine learning algorithms work well with the data produced

by the Programming Checkup. The following section provides a description of the different classification and regression algorithms which were included in the analysis and assessment, the process and results of which, is presented within Chapter 4.

OLS (Linear) Regression

Ordinary Least Squares (OLS) Regression utilises a linear function in order to make predictions, as seen in Equation 3.4 (Dietrich et al., 2015; El Aissaoui et al., 2020; James et al., 2013):

3.4)

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_P X_P$$

The Scikit-Learn (Pedregosa et al., 2011; Scikit-Learn, n.d.-o) implementation of OLS Regression was utilised within this investigation. It aims to establish a plane to model the multidimensional nature of the data when using multiple predictors. Establishing such a plane aims to minimise the sum of squared residual (RSS) between the observed and predicted responses, as can be seen in Equation 3.5 (Kuhn & Johnson, 2013):

3.5)

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Although the simplicity of OLS Regression allows for the effective modelling of linear relationships between predictors, it naturally struggles to account for nonlinear relationships within the data. Furthermore, it is also suspectable to being influenced by outlying observations that do not follow general trends in the data and therefore, have exceptionally large residuals, resulting in the model having to be adjusted to account for them (Kuhn & Johnson, 2013).

Ridge, Lasso and Elastic Net Regression

Complex OLS Regression models has a tendency to overfit the training data, whereby they fit the training data very well, but do not generalise effectively when attempting to make predictions with new, unseen data (Claesen & De Moor, 2015; Kuhn & Johnson, 2013). Therefore, a number of penalisation techniques have been developed to improve performance

on unseen data compared to OLS Regression by making a bias-variance trade-off (Zou & Hastie, 2005).

Within machine learning, bias is the error between the predictions made by the model and true values (James et al., 2013; Kuhn & Johnson, 2013). For example, the assumption of a fully linear relationship made by OLS Regression is likely to be an oversimplification of the relationships between the variables and as such, results in high bias and errors in prediction that cannot be accounted for by increasing the number of samples in the training dataset (James et al., 2013).

Variance refers to the variability of the error in the predictions made by the model if a different training set is used, as a model with high levels of variance will overfit the current training data and will likely struggle when applied to new unseen data (Bruce & Bruce, 2017; James et al., 2013). The bias-variance trade-off therefore relates to balancing the complexity of the model, as a highly complex model will experience a high level of variance and overfit the data, whereas an overly simplistic model will show strong levels of bias and underfit the data (Claesen & De Moor, 2015).

One example of a penalisation technique that attempts to improve on the performance of OLS Regression is Ridge Regression (Zou & Hastie, 2005). Ridge Regression attempts to negate the tendency to overfit data, as seen in OLS Regression (Kuhn & Johnson, 2013; Zou & Hastie, 2005) by reducing the variance of the model at the expense of bias (Zou & Hastie, 2005), which consequently often has the effect of reducing the overall error exhibited by the model when compared to OLS Regression (Kuhn & Johnson, 2013). Ridge Regression reduces variance by adding an L_2 squared magnitude of coefficients penalty to the parameter estimates, which has the effect of only allowing the individual regression parameters to become large if there is a proportional decrease in RSS (Kuhn & Johnson, 2013). The size of the penalty applied to the parameters is represented with the hyperparameter λ, as seen in Equation 3.6 (Kuhn & Johnson, 2013), which is referred to as "alpha" within the Scikit-Learn documentation (Scikit-Learn, n.d.-ab) due to lambda being a reserved keyword in Python:

3.6)

$$RSS_{L_2} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

The values of hyperparameters can have significant impacts on the performance of a model, for example, $\lambda = 0$ would make Ridge Regression equivalent to OLS Regression, whereas larger values shrink parameter estimates towards 0 (Claesen & De Moor, 2015; Kuhn & Johnson, 2013). In order to find the optimal hyperparameter values that minimise the error produced by the model it is necessary to examine how different configurations of hyperparameter values effect a model's performance. This process is referred to as "Hyperparameter Tuning", which essentially involves trialling a range of values for each hyperparameter in the models and identifying the combination that results in the lowest amount of error. However, the testing dataset must remain isolated from this process and cannot be used to evaluate the different hyperparameter values (Bengio & Grandvalet, 2004; Claesen & De Moor, 2015; Mantovani et al., 2015).

Instead, K-Fold Cross Validation is applied to the training dataset, which involves repeatedly splitting the dataset *K* times, with *1/K* of the data being reserved for testing (formally, the *validation* dataset) and the remaining data being used to train the model until each subset has been used as the validation set (Bengio & Grandvalet, 2004; Kuhn & Johnson, 2013; Vabalas et al., 2019; Wong & Yeh, 2020). Although this is a computationally expensive process, it allows for average performance to be estimated. This average performance is formally referred to as the *Expected value of Predication Error* (EPE) of the model with a given set of hyperparameter values. The EPE can be computed and subsequently compared with alternate hyperparameter configurations, allowing for an optimal configuration to be identified (Bengio & Grandvalet, 2004).

Within this investigation, hyperparameter tuning is implemented using Grid Search, which exhaustively searches through all possible hyperparameter combinations within a pre-determined parameter space (Mantovani et al., 2015). Although there are less computationally expensive hyperparameter tuning methods available, in this investigation Grid Search – and specifically, SciKit-Learn's GridSearchCV (Scikit-Learn, n.d.-k) –

remains a very popular technique, thus making it an appropriate choice as the size of the training dataset is not likely to result in excessive processing times (Mantovani et al., 2015).

Given that GridSearchCV (Scikit-Learn, n.d.-k) incorporates K-Fold Cross Validation, a value of 10 was set for *K*, as this has been shown through empirical studies to produce error rate estimates that experience neither high levels of bias or variance (James et al., 2013; Kuhn & Johnson, 2013), with a range of values on a log-10 scale from 10e-5 to 100 being evaluated in order to find the optimal value for λ.

Additionally, when evaluating techniques such as OLS Regression which do not require hyperparameter tuning, traditional K-Fold Cross Validation is used in place of Grid Search using SciKit-Learn's cross_val_score function (Scikit-Learn, n.d.-d).

One of the drawbacks to Ridge Regression is that it is unable to create a "parsimonious" model, meaning that whilst Ridge Regression is able to shrink parameter estimates towards 0, it cannot set them to absolute 0 and therefore, retains all predictors in the model and does not perform feature selection by removing any which are unimportant (Kuhn & Johnson, 2013; Zou & Hastie, 2005). A common alternative to Ridge Regression which is able to create a parsimonious model is Least Absolute Shrinkage and Selection Operator (lasso) which can be seen in Equation 3.7 (Kuhn & Johnson, 2013; Tibshiranit, 1996; Zou & Hastie, 2005).

3.7)

$$RSS_{L_1} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p}|\beta_j|$$

Lasso is fundamentally very similar to Ridge Regression as it allows parameter estimates towards 0 however, as the L_1 penalty is being applied to the absolute value of the magnitude of coefficients thus allowing parameter estimates to be set to absolute 0 (Kuhn & Johnson, 2013; Zou & Hastie, 2005). This ability to set parameter estimates to absolute 0 allows Lasso to perform feature selection and produce simpler, sparse models (James et al., 2013; Kuhn & Johnson, 2013).

Like Ridge Regression, Lasso is sensitive to values of λ, as an excessively large value will result in more parameter estimates to be set to 0, whereas too smaller value of λ would prevent unimportant variables from being removed from the model (Kuhn & Johnson, 2013). Hyperparameter tuning is employed in the same as Ridge Regression in order to obtain an optimal value of λ, which is also denoted as "alpha" within the Scikit-Learn documentation for the Lasso function (Scikit-Learn, n.d.-n).

Additionally, a middle ground between Ridge Regression and Lasso can be found within Elastic Net Regression which combines both the L_1 and L_2 penalties; as seen in Equation 3.8, which has been shown to be more effective at dealing with multicollinearity amongst predictor variables (Kuhn & Johnson, 2013; Zou & Hastie, 2005).

3.8)

$$RSS_{Enet} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda_1 \sum_{j=1}^{p} \beta_j^2 + \lambda_2 \sum_{j=1}^{p} |\beta_j|$$

As with both Ridge Regression and Lasso, Elastic Net requires hyperparameter tuning to be carried out in order to find an optimal value of λ (alpha). There is an additional hyperparameter; "l1_ratio", which acts as a mixing parameter for the L_1 and L_2 penalties where a value of 0 indicates only a L_2 penalty is applied, a value of 1 indicates a L_1 penalty is applied, and any value in between 0 and 1 is a combination of both penalties and therefore, requires tuning in order to find an optimal model (Scikit-Learn, n.d.-h).

Bayesian Ridge Regression

The regression methods described thus far can be considered frequentist methods as predictions are made in the form of single values, as opposed to Bayesian methods which make predictions based upon a probability distribution (Wakefield, 2013).

As part of the model evaluation process, the Bayesian variant of Ridge Regression was implemented which, unlike traditional Ridge Regression, does not require GridSearchCV to be utilised to find the optimal hyperparameter values. Instead, the SciKit-Learn implementation of Bayesian Ridge Regression (Scikit-Learn, n.d.-c) estimates hyperparameter values during the fitting process, as per the method posed by MacKay (1992)

in the description of the algorithm in Appendix A of Tipping (2001), and as such, only standard K-Fold Cross Validation is required when comparing Bayesian Ridge Regression to the other methods being evaluated.

Ridge Classification

Scikit-Learn also includes a classification variant of Ridge Regression wherein binary target variables are converted to {-1, 1} and then treated as a standard Ridge Regression problem (Scikit-Learn, n.d.-ac). Consequently, the Ridge Classifier has the same hyperparameters as the Ridge Regressor and are therefore trained in the same way as described previously.

Logistic Regression

Logistic Regression is an example of a Generalised Linear Model (GLM) which extends traditional linear (OLS) regression into new contexts (i.e., classification tasks), therefore making Logistic and OLS Regression akin to each other with the exception that Logistic Regression produces a binary outcome (Bruce & Bruce, 2017; Kuhn & Johnson, 2013). A GLM is characterised by two main components: a probability distribution; which in the case of Logistic Regression is a binomial distribution, and a link function to map the response to the predictors, which for Logistic Regression is a Logit function (Bruce & Bruce, 2017, p. 187). Logistic Regression is fit using Maximum Likelihood Estimation (MLE); as opposed to least squares used by OLS Regression which iteratively evaluates parameter estimates until the model no longer improves and is said to have "converged" (Bruce & Bruce, 2017; Miles & Shevlin, 2001).

As with the methods discussed previously, the Scikit-Learn implementation of Logistic Regression (Scikit-Learn, n.d.-r) has a number of hyperparameters, which can be tuned to optimise performance. In order to reduce the likelihood of overfitting the data the "penalty" hyperparameter can be used to determine what regularisation penalty is applied (L_1, L_2, elastic net or none), with the strength of the penalty being controlled with the "C" hyperparameter, where smaller values indicate stronger regularisation (Scikit-Learn, n.d.-r; Vabalas et al., 2019).

An additional hyperparameter, "solver", specifies the optimisation algorithm to use whilst training the Logistic Regression model (Scikit-Learn, n.d.-r). Although it is possible to use GridSearchCV to compare different solver methods, the solver "liblinear" has been chosen as

it works well with small datasets however, liblinear is not compatible with the elastic net penalty and as such, cannot be included in the parameter grid (Scikit-Learn, n.d.-r; Vabalas et al., 2019). It should be noted that GridSearchCV utilises stratified K-Fold Cross Validation when applied to a binary classification problem, which ensures that the distribution of the classes is the same in each fold (Scikit-Learn, n.d.-k).

Support Vector Machines

Support Vector Machines (SVMs) are a set of highly adaptable modelling techniques originally developed by Vapnik (2000) as a classification method and were later expanded to allow them to also be applied to regression tasks (Kuhn & Johnson, 2013; Vapnik, 2000). Briefly, the concept of a Support Vector Machine Classifier (SVC) revolves around the concept of a "margin", which in the context of classification, refers to the distance between the decision boundary (hyperplane) that separates two classes and the closest data points (Kotsiantis, 2007; Kuhn & Johnson, 2013).

By maximising the margin (i.e., finding the maximum distance between the decision boundary and the data points on either side), the generalisability of the model can be improved (Bishop, 2006; Kotsiantis, 2007; Kuhn & Johnson, 2013). However, it may not always be possible for a decision boundary to be established due to misclassified instances (Kotsiantis, 2007). This issue can be accounted for through the use of a "soft margin", which allows some misclassifications of the training data to be accepted (Bishop, 2006; Kotsiantis, 2007). The soft margin is another example of the bias-variance trade off and is controlled using the regularisation hyperparameter "C", where lower values allow for as wide a margin as possible by allowing more misclassifications, and higher values tighten the margin and reduce the number of misclassifications being accepted (Bishop, 2006; Hsu et al., 2008; Kotsiantis, 2007; Scikit-Learn, n.d.-ae).

One of the advantages of Support Vector Machines is their ability to produce extremely flexible decision boundaries through the use of the "kernel trick" (Kuhn & Johnson, 2013). The kernel trick works by mapping the original "input space" data into a higher dimensional (possibly infinite) "transformed feature space", thus allowing for an appropriate decision boundary to be established (Hsu et al., 2008; Kotsiantis, 2007; Kuhn & Johnson, 2013).

This therefore allows SVMs to handle the complexities in relationships between variables, which often occur in real-world data and would otherwise make it impossible for a linear decision boundary to be established (Kotsiantis, 2007).

Numerous kernel functions are available to transform the data into a higher dimensional space (Hsu et al., 2008). However, for this investigation only two kernel functions will be evaluated. The first is the Linear Kernel Function, assumes that the classes are linearly separable. However, the second kernel function, the Radial Basis Function (RBF) does not make any assumptions about the classes being linearly separable and is therefore able to handle nonlinear relationships between classes, making it a popular choice for use with SVMs (Hsu et al., 2008).

Both kernels require the C hyperparameter. However, RBF requires an additional hyperparameter, γ. This dictates how far each training example's influence on the decision boundary reaches, with lower values of γ meaning examples that are further away can influence the decision boundary, whereas high γ values indicate that only examples that are close can influence the decision boundary (Scikit-Learn, n.d.-z, n.d.-ae). In order to optimise the model's performance on new data, values for both C and γ should be established using cross-validation.

Support Vector Regression (SVR) follows the same principles as Support Vector Classification and is designed to be less affected by outliers than OLS by penalising any training examples that fall outside of the margin, hence being referred to as the ϵ-tube (Awad & Khanna, 2015; Kuhn & Johnson, 2013; Zhang et al., 2014). In essence, SVR allows predictions to be made within a range of tolerance and as such, requires an additional hyperparameter ϵ to be tuned, which dictates the width of the ϵ-tube (Scikit-Learn, n.d.-af). Like SVC, during the current model evaluation process both Linear and RBF kernels will be employed with SVR, with the LinearSVC (Scikit-Learn, n.d.-p) and LinearSVR (Scikit-Learn, n.d.-q) implementations being used to trial the Linear kernel, as opposed to the standard SVC and SVR, due to the increased performance that is offered (Scikit-Learn, n.d.-ad).

K-Nearest Neighbor

K-Nearest Neighbor (KNN) is a simplistic machine learning algorithm with both classification and regression variants (Batista & Silva, 2009; Bruce & Bruce, 2017; Dudani, 1976; Kuhn & Johnson, 2013). KNN classification works on the premise that data-points that are determined to be close together using a distance metric will share the same classification (Batista & Silva, 2009; Bruce & Bruce, 2017; Dudani, 1976). Numerous distance metrics are available, with the SciKit-Learn implementation of KNN using the Minkowski metric (Equation 3.9) with a power (p) value of 2 by default (Scikit-Learn, n.d.-l, n.d.-m). This is equivalent to Euclidean distance, which is one of the most commonly used distance metrics (Kuhn & Johnson, 2013).

3.9)

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

The parameter k represents the number of "neighbors" that, according to the distance metric, are closest to the element being classified, with its class being established by finding the majority class amongst the neighbors (Bruce & Bruce, 2017; Chomboon et al., 2015; Kuhn & Johnson, 2013). Similarly, for regression tasks, the average value of the k closest neighbours to the datum are used to make predictions.

It is therefore important that an appropriate value for k (n_neighbors within the SciKit Learn documentation; Scikit-Learn, n.d.-l, n.d.-m) is selected through cross-validation in order to produce an optimal model and minimise the chance of overfitting the data by selecting a value of k which is too small, or underfitting by selecting a value of k which is too large (Batista & Silva, 2009; Kuhn & Johnson, 2013). Within the standard implementation of KNN all of the k closest neighbors are weighted equally (uniform weighting), regardless of how far away they are from the datum being classified or predicted (Dudani, 1976). Alternatively, Distance-Weighted KNN accounts for the distance of the neighbors by applying a weight such that closer neighbours have a greater weight and therefore exert a greater influence when making the classification or prediction (Batista & Silva, 2009; Dudani, 1976). Within SciKit-Learn the "weights" hyperparameter dictates whether unform or distance weighting functions are applied, with the distance weight function weighing points by the inverse of

their distance (Scikit-Learn, n.d.-l, n.d.-m). Both variants will be evaluated as part of this investigation.

Decision Tree, Bagging Decision Trees and Random Forests

Tree-based methods are a popular family of machine learning algorithms, which can be applied to both classification and regression problems (Bruce & Bruce, 2017; Kuhn & Johnson, 2013). Decision Trees are the most basic form of tree-based models, which aim to establish smaller and more homogenous groups within the training data by partitioning the data into multiple nested if-then statements (Kuhn & Johnson, 2013). This approach enables complex relationships to be uncovered within the data, whilst also allowing for a highly interpretable model to be produced that can be plotted visually, making decision trees extremely useful for carrying out exploratory data analysis (Bruce & Bruce, 2017; Kuhn & Johnson, 2013; Myles et al., 2004).

There exist multiple algorithms for constructing Decision Trees, with SciKit-Learn's implementation using an optimised version of the CART (Classification and Regression Tree; Scikit-Learn, n.d.-g). Trees are constructed through recursive partitioning in which the data are repeatedly partitioned to create increasingly homogeneous segments by a predictor, which is found to provide the best separation at that level of the tree (Bruce & Bruce, 2017; Myles et al., 2004). This therefore requires a measure of homogeneity, or class impurity, in order to establish the most appropriate predictor to perform the partition. The SciKit-Learn Decision Tree Classifier allows for one of two different measures of class impurity to be used when creating partitions; Gini Impurity (Equation 3.10) and Entropy of Information (Equation 3.11), where p is the proportion of misclassified results for a partition A (Bruce & Bruce, 2017; Scikit-Learn, n.d.-f).

3.10)

$$I(A) = p(1 - p)$$

3.11)

$$I(A) = p\ log_2(p) - (1 - p)\ log_2\ (1 - p)$$

The method to be used is specified using the "criterion" hyperparameter, with Gini being the default choice that will subsequently be used within this investigation. For Regression Trees, the default splitting criterion is the squared error of each sub-partition (Bruce & Bruce, 2017; Kuhn & Johnson, 2013; Scikit-Learn, n.d.-g).

One of the major drawbacks of the basic Decision Tree is that by default, the tree will grow to an extent where it will likely account for the variation within the training set and the partitioning rules being used have become overly complex and generally only reflect noise within the data. This therefore means the tree has overfit the training data and will as such struggle to generalise to new data (Bruce & Bruce, 2017; Kuhn & Johnson, 2013; Myles et al., 2004).

There are two ways to minimise the potential for overfitting when training a Decision Tree. The first method is to arbitrarily limit the tree's growth using hyperparameters such as: max_depth (maximum tree depth), min_samples (minimum number of samples required to split an internal node) and min_samples_leaf (minimum samples needed to be a leaf node) (Bruce & Bruce, 2017; Scikit-Learn, n.d.-f, n.d.-g), the values of which are established using hyperparameter tuning such as GridSearchCV.

The second approach, which is generally accepted to be a better (Myles et al., 2004), is to allow the tree to grow unimpeded and then "pruned back" to produce an overall smaller tree that is more generalisable (Bruce & Bruce, 2017; Kuhn & Johnson, 2013; Myles et al., 2004). Pruning within Scikit-Learn is achieved through a process known as Minimal Cost-Complexity Pruning where, as shown in Equation 3.12, a complexity parameter α is used to calculate the cost-complexity, $R_\alpha(T)$ for a given tree T with $|\tilde{T}|$ representing the number of terminal nodes (i.e., leaves), in the tree T, and $R(T)$ representing the total misclassification rate of the terminal nodes (Hastie et al., 2009; Kiran & Serra, 2017; Scikit-Learn, n.d.-e).

3.12)

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

Furthermore, Equation 3.13 demonstrates how the cost-complexity can be calculated for an internal node, that is, all nodes which are not terminal leaf nodes or a root node (Kiran & Serra, 2017; Scikit-Learn, n.d.-e).

3.13)

$$R_\alpha(t) = R(t) + \alpha$$

SciKit-Learn provides the hyperparameter ccp_alpha, which allows the value of α to be set. Trees are then pruned by removing a subtree with the largest cost-complexity that is also smaller than the value of α (Scikit-Learn, n.d.-e). Consequently, an optimal value for the ccp_alpha hyperparameter should be found using a method such a GridSearchCV (Bruce & Bruce, 2017; Kuhn & Johnson, 2013).

In addition to the issues relating to overfitting, Regression Trees may have higher error rates than other types of regression models, with Kuhn (2013) stating:

> By construction, tree models partition the data into rectangular regions of the predictor space. If the relationship between the predictors and the outcome is not adequately described by these rectangles, then the predictive performance of a tree will not be optimal. (p. 181)

This is a fundamental drawback of Regression Trees, as only a finite number of terminal leaf nodes can be constructed, limiting the tree's ability to fully reflect trends within the data.

To overcome the issue of high variance leading to poor performance on unseen data, ensemble techniques, which combine predictions from multiple models (Kuhn & Johnson, 2013), have been developed in an attempt to produce tree-based models that have lower levels of variance and as such perform better on unseen data. One such technique is Bootstrap Aggregation, commonly referred to as *Bagging***,** which attempts to reduce variance by using bootstrapping to construct an ensemble of trees, although bagging can be applied to any classification or regression algorithm (Bruce & Bruce, 2017; Dietterich, 2000; James et al., 2013; Kuhn & Johnson, 2013).

Bootstrapping works by taking a random sample of the data with *replacement* (Efron & Tibshirani, 1986) wherein a data point can be selected to be part of the bootstrap sample, and still be available to be selected again (Efron & Tibshirani, 1986; Kuhn & Johnson, 2013). This selection process is repeated until the sample is the same size as the original dataset, which consequently means that some data-points will occur multiple times within the bootstrap sample, whereas others will not be included (Efron & Tibshirani, 1986; Kuhn & Johnson, 2013). The subset of data-points not selected for inclusion in the bootstrap is referred to as the "out-of-bag" sample (Kuhn & Johnson, 2013).

Each of the individual Decision Trees within the ensemble will be fit to a different bootstrap sample, with the number of trees being controlled by the n_estimators hyperparameter within SciKit-Learn (Bruce & Bruce, 2017; Kuhn & Johnson, 2013; Scikit-Learn, n.d.-a, n.d.-b). When making predictions, each individual tree will make its own prediction, which will then be averaged in order to produce a prediction for the entire ensemble. This therefore decreases the variance of predictions as each of the trees will have a different structure as they will have been fit using a different bootstrap sample. However, the individual trees within a bagging model are not fully independent of each other as all predictors are available when fitting. With a relatively large dataset, this can lead to "tree correlation", where trees exhibit similar structures, particularly at higher levels, due to the underlying relationships within the data (Kuhn & Johnson, 2013).

The second ensemble technique, Random Forest, overcomes the issue of tree correlation, and therefore reduces the variance further as opposed to Bagging, by using a random subset of the available predictors in addition to a bootstrap sample when fitting each tree (Bruce & Bruce, 2017; Kuhn & Johnson, 2013). The SciKit-Learn implementation of Random Forest considers the square root of the number of available predictors by default when partitioning data (Scikit-Learn, n.d.-x, n.d.-y).

Like Bagging, Random Forest requires the number of trees in the forest to be specified using the n_estimators hypermeter (Scikit-Learn, n.d.-x, n.d.-y), with an optimal number being determined with GridSearchCV. Furthermore, hyperparameters such as max_depth, min_samples_split and min_samples_leaf can be used to optimise the individual trees in the forest. Additionally, the max_features hyperparameter can be used to control the number of

features being considered when identifying the best split during the tree growing process (Scikit-Learn, n.d.-x, n.d.-y).

Random Forests are more computationally efficient than Bagging on a tree-by-tree basis as only a subset of the available predictors need to be evaluated at each partition (Kuhn & Johnson, 2013). However, both techniques sacrifice the interpretability of individual Decision Trees in favour of a decrease in variance.

Gradient Boost and XGBoost

Gradient Boost, like Bagging and Random Forests, is an ensemble method commonly used with decision trees (Bruce & Bruce, 2017), which can be applied to both classification and regression problems. Briefly, Gradient Boost is an additive model that utilises "weak learners" in the form of decision or regression trees for their respective problems (Kuhn & Johnson, 2013; Ye et al., 2009). The model starts with a "best guess" of the target variable, which consists of a single leaf, and the mean of the target variable for regression problems. For classification problems, the log odds of the target variable, which can be converted into a probability using a logistic function, which is the inverse of the logit function. Subsequently, the pseudo-residuals are calculated, as this is for an individual tree as opposed to the entire ensemble, and a new tree is fit to the pseudo-residuals as opposed to the target variable (James et al., 2013; Kuhn & Johnson, 2013). This process repeats until the specified number of iterations has been completed with each successive tree minimising the error of the previous one through the use of Gradient Decent. Within SciKit-Learn the maximum number of iterations is determined using the n_estimators hyperparameter (Scikit-Learn, n.d.-i, n.d.-j).

Gradient Boost can be considered a Greedy algorithm as the optimal weak learner is selected at each stage such that the overall ensemble cannot be guaranteed to be optimal, which can also lead to overfitting (Kuhn & Johnson, 2013). However, the potential for overfitting can be reduced by constraining the learning rate through the use of a regularisation parameter, which is represented by the hyperparameter learning_rate within SciKit-Learn (James et al., 2013; Kuhn & Johnson, 2013; Scikit-Learn, n.d.-i, n.d.-j). Furthermore, it is common to constrain the size of trees using the max_depth hyperparameter, which also aids in reducing overfitting (Scikit-Learn, n.d.-i, n.d.-j).

Stochastic Gradient Boost follows the same process as Gradient Boost with the addition of taking a random sample of the training data when training each individual tree (Kuhn & Johnson, 2013). XGBoost (eXtreme Gradient Boost) is one of the most widely used implementations of Stochastic Gradient Boost due to its scalability and efficiency (Bruce & Bruce, 2017; Chen et al., 2018). XGBoost is highly tuneable with a large number of hyperparameters, which can be adjusted to optimise performance using GridSearchCV. Two particularly significant hyperparameters are "subsample", which specifies the proportion of the training data to be sampled, and "eta", which controls the learning rate and helps to prevent overfitting (Bruce & Bruce, 2017; XGBoost, n.d.).

Neural Networks

Neural Networks are one of the most popular and flexible machine learning algorithms, which have been a focus for researchers dating back to the 1940's, with modern advances in computing power allowing for larger, more complex neural networks to be developed (Awad & Khanna, 2015; Russell & Norvig, 2020). Briefly, Neural Networks are modelled on the biological brain and comprise a series of nodes (neurons), which have been aggregated into layers (Karsoliya, 2012; Russell & Norvig, 2020; Tomasevic et al., 2020). Layers are connected together by directed links between nodes, which all travel in the same direction within a feedforward Neural Network (Russell & Norvig, 2020), with each link having an associated weight which determines the strength of the connection (Russell & Norvig, 2020). Neurons within a biological brain receive and process signals; similarly, the nodes within the Neural Network sum the weights from all of the incoming links, which with the addition of a bias, are applied to an activation function (Awad & Khanna, 2015; Russell & Norvig, 2020). Numerous types of activation functions are available to suit different network configurations (Awad & Khanna, 2015), but they all serve the same purpose, which is to "activate" the connected node(s) in the subsequent layer in the network when a given threshold is exceeded.

There are three main types of layer within a Neural Network (Awad & Khanna, 2015; Géron, 2022; Karsoliya, 2012):

- Input layer - External data are presented to the network via the input layer. Every node represents an independent variable that can influence the output of the network.
- Output layer – This outputs the results from the network to the external world. The number of nodes is proportional to the output of the network (see below).

- Hidden layer(s) – The hidden layers lie in between the input and the output layers and are where nodes perform different transformations on the input data through the use of the activation functions. There is no theoretical basis for the optimal number of hidden layers nor the number of nodes, as each network is application-specific and must therefore be developed through trial and error (Awad & Khanna, 2015). However, the weights and bias within a network can be optimised using backpropagation (backwards propagation of error), which utilises the error from the output layer and gradient decent in order to maximise the performance of the network (Awad & Khanna, 2015; Géron, 2022).

Karsoyila (2012) states that one or two hidden layers are sufficient to solve most complex non-linear problems and therefore in the present investigation the Scikit-Learn Neural Network implementations for classification (MLPClassifier) (Scikit-Learn, n.d.-t) and regression (MLPRegressor) (Scikit-Learn, n.d.-u) will be used to find an optimal network topology.Although alternative Neural Network libraries such as Keras (Chollet, 2015) allow for greater customisation of complex networks, which goes beyond the focus of this investigation, SciKit-Learn makes it easy to compare different basic network configurations using the hidden_layer_sizes hyperparameter, which can be optimised using GridSearchCV (Scikit-Learn, n.d.-t, n.d.-u). Activation functions for the hidden layers can be set using the activation hyperparameter, with ReLU being the default option. The optimizer can also be configured using the solver hyperparameter (Scikit-Learn, n.d.-t, n.d.-u). Furthermore, the lbfgs solver will be used for this investigation as it as it has been shown to perform well on smaller datasets and the alpha hyperparameter can be used to apply L_2 regularisation in order to minimise the chance of the networks overfitting the data (Scikit-Learn, n.d.-t, n.d.-u).

3.6 Summary and Methodology Reflection

This chapter provides an account of the methodological approach to data collection within this investigation, which culminates in the development of the Programming Checkup as a means of collecting data to aid in answering the research questions, and to also support the development of the predictive model. An overview has also been provided of the different machine learning algorithms that form the basis of the analysis presented within the following chapter.

As this investigation sits firmly within the realms of quantitative research, it is also important to acknowledge the educational context of the investigation and how this influences the research design. Reflection, which is defined as "deliberation, pondering, or rumination over ideas, circumstances, or experiences yet to be enacted, as well as those presently unfolding or already passed" (Alexander, 2017, p. 308), is an important process within the field of education as a means of improving practices. Reflection alone does not always translate into changes in teaching practice (Feucht et al., 2017). Feucht et al. (2017) argue that the process of *reflexivity*, which is broadly defined as an internal dialogue leading to transformative actions within a classroom (Archer, 2012; Feucht et al., 2017), may support the ideas that emerge during reflection being acted upon.

Subsequently, the processes of reflection and reflexivity can be applied to this research investigation in a similar way to how they are applied within classrooms. Kamler and Thomson (2014) state that a reflective researcher applies the same critical stance towards their own work as they do with their research data. However, reflexivity takes this process one step further by analysing the researcher's role in the investigation and challenges the perspectives and assumptions of both the researcher and the wider world that the investigation is being conducted in (Palaganas et al., 2017; Parahoo, 2014). Reflexivity is seen as a continuous process which permeates every aspect of the research investigation and allows for the researcher's values and beliefs, which inevitably influence the research process, to be made transparent. Additionally, the researcher's interactions with the participants can be detailed from the first point of contract until the end of the study, thus enabling the results to be understood not only in terms of what was discovered, but also in terms of how it was discovered (Etherington, 2007; Hertz as cited in Etherington, 2007; Parahoo, 2014).

Palaganas et al. (2017) claim that a researcher's personality does not exist independently of the research process, nor does it completely determine it, and should therefore be viewed as a dialogue for challenging perspectives and assumptions. Nevertheless, the personality or rather, the "personal epistemology", a person's cognitions about the nature of knowledge and the process of knowing (Pintrich, 2002) undoubtedly influence the research investigation. However, the overall quality and validity of the research can be improved by engaging in the process of reflexivity and acknowledging the limitations of the knowledge being produced (Etherington, 2007).

In order to ensure the integrity of this research investigation, it is important to acknowledge my own personal epistemology and interactions with participants, thus allowing for a complete understanding of the methodological decisions which have been made, as well as any factors that may influence the outcomes of this research.

It should be noted that I have a background in computer science and as such, was previously unaccustomed to conducting in-depth research in an education setting. This has undoubtedly influenced the methodology of this investigation, with a primarily quantitative approach being taken to the research design, although this is supported by previous more qualitative-focused research of others, particularly relating to the discovery of students' misconceptions. Although as a science-based researcher I am expected to remain as objective as possible, this investigation draws heavily on my own experiences with programming, to the extent that I can be considered a "relative insider" (Griffiths, 1998) for two primary reasons:

1. I studied a previous version of the introductory programming module that students are currently studying, making me familiar with some of the difficulties of learning to program first-hand. However, the experience I had gained in programming prior to starting university aided me in my learning, so my own personal experiences do not necessarily reflect those of all students, particularly those who come to university having never programmed before.

2. I also contribute to the teaching of the introductory programming module that participants are studying, and subsequently became the module leader during the second year of data collection.

Being a relative insider allows me, to an extent, to understand students' perspectives on learning to program. In addition, my involvement in teaching them to program provides me with valuable insight into the difficulties that they face, particularly when I can observe students demonstrating misconceptions while completing their work. However, it raises the issue of how my involvement with the participants influences this research, particularly relating to the dual personalities that I must project to students (Etherington, 2007), as both a researcher and as an educator. I personally view myself as a practitioner-researcher, whereby as the research I am conducting is within my own work setting, a symbiotic relationship exists where my research informs my teaching, and my experiences in the classroom inform my research direction. However, my students must always remain my main priority on a day-to-day basis.

In order to carry out a rigorous and credible investigation, I feel I must adopt a clear, unbiased and analytical persona towards the participants taking part in the investigation in order to prevent my own personal biases from influencing the outcomes of the research. Furthermore, it is important that I make students aware that their participation is completely voluntary, and all responses are treated with the strictest confidence. However, by being involved with the delivery of the introductory programming module to students, I feel that I must also project a sense of approachability to students, making it known to them that I am there to support them during their weekly classes, as well as outside of the lesson if required. I have regular contact with the entire first year Computing cohort through lectures, programming labs and also an optional support lecture, supporting the perceived persona that I am the person for students to turn to if they are struggling with their programming work. Although these two personas do appear to be polar opposites, a series of compromises were made throughout the course of this investigation, which both ensured the integrity of the research, as well allowing students to be appropriately supported.

The first issue that arose was the potential for students to feel compelled to take part in the study due to the fact that I am a member of the teaching staff, and that not taking part may negatively affect their grades. Equally, some students; particularly those who are the least confident with programming, may be reluctant to take part due to a lack of confidence in their abilities. To alleviate these issues students were told throughout the investigation that participation was completely voluntary and would in no way affect their grades. Students were also told that their responses would be confidential and that they would only receive

feedback if they requested it. Given that sizeable numbers of students opted not to participate in the investigation, I am confident that efforts to inform students that the Programming Checkup was not a mandatory exercise were successful. However, from experience, students who are often the most in need of support are also those who do not engage with additional activities, such as the Programming Checkup, meaning potentially important data may have been missed.

The question of how much the results from students' first attempt at the Programming Checkup should influence my teaching, prior to the students attempting it for a second time was also raised during the investigation. An external researcher would be able to collect data from participants at the two time points with little or no additional contact with students. However, by having regular contact with students there is a greater opportunity for students to ask questions about the research. Any general questions about the research process were answered as clearly as possible but a number of students had more in-depth questions relating to the underlying theories behind the research. Providing students with a detailed description of how the Programming Checkup works prior to the completion of data collection could potentially influence their responses, and therefore, students were promised an in-depth explanation in one of the optional programming support lectures during the second semester, thus allowing the data collection to be completed without interference, while also not taking up any lesson time in the lead-up to the students' assessments. Additionally, there is also the issue of whether the misconceptions students demonstrate in their first attempt at the Programming Checkup should be addressed. Making students explicitly aware of their misconceptions may potentially increase the probability of them overcoming them relative to what would be the case without intervention.

As examining the probability of a student overcoming their misconceptions with or without intervention is beyond the scope of this project, withholding support from students could be deemed unethical from a teaching perspective, as students' misconceptions can become more deeply engrained over time if they are not addressed. In order to account for this, students who requested feedback on their first attempt at the Programming Checkup were given generalised feedback for each concept they demonstrated difficulty with. Furthermore, students were advised to attend the optional support lecture where the concepts that had been introduced in the main lecture were reinforced through additional examples and in-depth explanations. However, the misconceptions identified within the Programming Checkup were

not explicitly mentioned during these explanations, although, after completing their second attempt at the Programming Checkup, students were provided with additional feedback that explicitly addressed their misconceptions in both their direct feedback, and in the support lecture.

Students were generally happy with the approach taken for the provision of feedback from the Programming Checkup, with a number of students commenting that their main motivation for taking part was the feedback that they would receive. The generalised feedback given after students completed the Programming Checkup for the first time successfully prompted a large number of students to regularly attend the optional support lecture. The Programming Checkup results also informed my discussions with struggling students who requested one-to-one support by allowing me to have an understanding of where the students were likely to be struggling prior to meeting with them, which reaffirms my intentions for implementing the Programming Checkup, in conjunction with the predictive model, as a formal part of the introductory programming module upon completion of this investigation.

4. Predictive Model Development

4.1 Model Objectives

The second phase of this investigation was the development of a predictive model which, as RQ 3 states, can be used to predict students' introductory programming assessment results. Previous research into developing predictive models of programming performance have used students' results in their introductory programming module (sometimes referred to as CS1) as the outcome variable to be predicted. For example, in their study of motivation and comfort-level, Bergin and Reilly (2005a) were able to produce a regression model which was able to account for 60% of the variance in students' overall performance in their introductory programming module. An alternative approach taken by some researchers has been to dichotomise students' results into "pass" and "fail" categories, with some placing the decision boundary at 50% (Castro-Wunsch et al., 2017) and others at 40% (Liao et al., 2019; Tomasevic et al., 2020). Furthermore, an important area of study within both Educational Data Mining and Learning Analytics is "Latent Knowledge Estimation", wherein students' knowledge of specific skills is assessed by their pattern of correctness (Baker & Siemens, 2014) using methods such as Bayesian Knowledge Tracing; as described in Section 3.3.

Although the assessments undertaken by students differ across institutions, it was decided that the most appropriate dependent variable for this investigation would be the results students achieve on their first introductory programming assessment (Assessment 1), which is completed at the end of the first semester. This assessment was chosen rather than the overall module grade as it focuses on evaluating students' core programming skills (use of variables, text input/output, conditional statements, loops, and functions) all of which; with the exception of functions, are examined within the Programming Checkup. Furthermore, the grades students achieve in their first assessment are significantly correlated with that of their second assessment, which is undertaken at the end of the module, $r_s = 0.509$ $p = < .001$. This therefore makes the results students achieve in their first assessment indicative of their future performance and as such, will allow for students who are likely to require support to be identified through their responses to the Programming Checkup at the beginning of the academic year (T1).

Previous research has shown that students who lack the appropriate mental models of fundamental programming concepts will generally find the learning process more difficult (Ben-Ari, 1998; Sorva, 2013). As such, during the model development process I aimed to evaluate the use of Bayesian Knowledge Tracing to estimate the likelihood of students holding appropriate mental models for core programming concepts. This evaluation was done in conjunction with the other factors examined within the Programming Checkup. The use of Bayesian Knowledge Tracing was therefore implemented to support the analysis of students' mental models when answering RQ 1, as well as the identification of students who are likely to require support by making predictions based on their responses to the Programming Checkup. Having students complete the Programming Checkup at the earliest available opportunity would allow for future interventions to be developed and implemented in the early stages of a course to directly support students in the construction of their mental models; which may otherwise be more difficult at a later stage due to the misconceptions that can develop (Omer et al., 2021; Omer & Farooq, 2020; Winslow, 1996). What form these interventions might take is outside the scope of the present investigation; however, they are being considered for future research stemming from this work.

Data were collected over the course of three years in the form of responses to the Programming Checkup at both T1 and T2 together with students' Assessment 1 results, with 70% of the data being randomly selected to be used to train and develop the model and the remaining 30% being reserved as a testing holdout set (James et al., 2000, p. 176). Using a separate testing dataset allows for the model to be evaluated independently of the data used to train it, thus giving a closer estimate of the real-world performance of the model (Russell & Norvig, 2020). The total dataset contained 285 responses after removal of students who did not complete Assessment 1 or who skipped 25% or more of the Programming Diagnostic questions within the Programming Checkup, therefore resulting in a training and testing dataset sizes of 200 responses and 85 responses, respectively.

It should be noted that Assessment 1 was changed from a written exam to a practical assignment after the first year of data collection, due to a change in assessment policy which was outside the control of this investigation. The practical assignment which was completed by subsequent year groups was written to assess the same learning outcomes as the written exam, which tests students' abilities to construct a structed solution to a simple problem, explain the importance of code readability and maintainability, and check the robustness of

code using an appropriate testing strategy. A Kruskal Wallis test confirmed that there was no significant difference between the results of students who completed the written exam during Year 1 of data collection (M = 69.81, SD = 22.57) and those who completed the practical assignment in the subsequent years of data collection (Year 2, M = 69.29, SD = 21.87, Year 3, M = 69.85, SD = 14.22), $H(2) = 2.43$, $p = .296$, $\eta^2 = .009$.

Using a model to predict students' Assessment 1 results based on their responses to the Programming Checkup at the beginning of the academic year allows teaching staff to provide dedicated support to aid them with constructing appropriate mental models before moving on to more complex topics, which are typically covered in the second semester of teaching. The subsequent sections of this chapter intend to detail the steps taken during the development and validation of the predictive model in aid of answering RQ 3.

4.2 Data Pre-Processing

A wide variety of potential regression and classification models have been described within the section 3.5, each with their own advantages and disadvantages. Although many of these models have been utilised in the previously mentioned EDM-based research investigations, it is difficult to pre-empt which models will perform the best when attempting to predict students' Assessment 1 results, hence the need to explore a range of potential models.

Before any machine learning models can be applied to the dataset there are several stages of pre-processing that must first take place. This section will describe how the raw output from the Programming Checkup needs to be prepared in order to allow both classification and regression methods to make predictions. It should be noted that the analysis carried out in this section was done before and separate to the in-depth analysis of results presented within Chapter 5. This was in order to prevent the holdout testing set from having any influence on the decisions made during the development of the predictive model. As such, any statistics presented within this section only apply to the training dataset.

Stage 1 – Preparation of Data Exported from Qualtrics

As the Programming Checkup was being distributed via a Qualtrics survey, the first step was to extract the required data from the raw export file generated by Qualtrics. In order to reduce the potential for human error affecting the dataset, a Python script was utilised to automate the extraction process. Much of the data could be extracted directly from the export file, including students' background details, previous experiences, estimations of difficulty, and mental effort levels. However, some aspects of the dataset required additional processing before they could be included in the final dataset. For example, students' average values needed to be calculated for their confidence level for each of the question concept categories (i.e., Variable Assignment, Conditional Statements, and Iteration), as well as for each of the three self-efficacy factors examined within the Programming Checkup as described in Section 3.4.

Additionally, students' answers to the Programming Diagnostic questions needed to be "coded" in order to indicate what misconceptions students were exhibiting based on their answers. This approach was based upon the work conducted by Dehnadi (2006), where a set of pre-defined answers for each question was developed using the literature discussed in

Section 3.4, to capture one or more misconceptions arising for each question (see Appendix A, Section 3). Where a student's response matched one of the pre-defined answers, the student's response to the question was coded with the corresponding misconception(s). If no matching answer could be found then the answer was coded as "NA" and the student's entire response to the Programming Checkup was flagged for review, meaning that the unmatchable answer could be investigated. These unexpected answers could range from formatting issues (i.e., a student included commas in their answer when none were expected), to genuine answers, which did not correspond to any expected misconceptions. Although these types of answers were generally infrequent, an attempt was made to determine how the student had arrived at their answer and map it to an appropriate misconception (or multiple misconceptions, if appropriate). However, if no reasonable mappings could be made, the answer was coded as "NA". To ensure consistency, any mappings made to unexpected answers were recorded to ensure that any other responses that included the same answer were also coded with the same misconception(s). Additionally, if a student did not answer a question, it was coded as "SK" for skip. The VN (Variable Naming) and PL (Parallelism) misconceptions also required some extra processing in order to be coded, as both require comparisons to be made between two questions, as explained in Section 3.4. Where appropriate, the misconception code for VN or PL was appended to any other misconceptions already identified within the student's response.

Aside from ensuring that answers were coded consistently, the script also enabled Bayesian Knowledge Tracing to be used to evaluate whether students held appropriate mental models (see Stage 3), by recording if a student had demonstrated a misconception associated with the mental model(s) being examined within each question, as shown in Table 4.1. If the student demonstrated the misconception, then the response for that question was coded as a 0, thus showing that the student made an error in answering that particular question and, therefore, may not be in possession of an accurate mental model. If a student answered the question correctly then the question was coded as 1. However, if the student skipped or provided an answer which could not be mapped to a specific misconception (NA) then the response was coded as 0. This information was necessary in order for Bayesian Knowledge Tracing to evaluate whether a student held an appropriate mental model or not, as well as for training the initial hyperparameters for each mental model.

Table 4.1

Associated Misconceptions of Each Mental Model

Mental Model	Associated Misconceptions
AND	AND
Conditional Statements	AND, OR, NOT, IF
IF	IF
Iteration	ET, LT, NI, SE, SM, SP
NOT	NOT
Output	OP
OR	OR
Parallelism	PL
Variable Assignment	AD, EX, MA, NC, REV, SW
Variable Naming	VN

By using the script to process responses to the Programming Checkup, students' anonymity was protected by replacing students' University ID numbers with participant numbers. However, this means it was necessary for students to enter their ID numbers correctly at both T1 and T2 in order for their responses to be linked. A text file was used to store each student's ID and participant number, which were checked when processing each response; if a matching ID was found then the corresponding participant number was applied to the response, otherwise a new number was assigned. Only the participant number of the students was used within the dataset with the only record of students' identity being the text file the script used to keep track of existing ID/participant number pairs.

Prior to the development of the automated marking script, the responses to the first data collection in September 2019 were initially manually coded. In order to verify the coding technique an external marker, who had no prior involvement with the development of the Programming Checkup, was asked to independently mark 10% of the responses collected so far. This external marker was provided with a list of examples of correct answers and a non-exhaustive list of incorrect answers, which indicated particular misconceptions for each question, complemented by a description of these misconceptions (see Appendix B). Whilst these examples were derived from the previously discussed literature relating to students' misconceptions and the answers provided by students, it is likely that they will have been unavoidably influenced by my own epistemological viewpoint, particularly when dealing

with unexpected answers. Likewise, the coding being carried out by the external marker would also be influenced by their own viewpoint.

Time constraints did not allow for a sufficient number of students to be interviewed about their responses in order to make any generalisable conclusions about whether the coded misconceptions completely reflected the issues they were experiencing. However, this approach did still allow for a perception to be established of the mistakes that students were making. Going forward, interviews and code walkthroughs should be an essential part of future work stemming from this investigation as discussed further in Section 6.4 below.

An interrater reliability analysis between myself and the external marker was conducted on each question using the Kappa statistic and was found to be between .67 ($p < .001$) and 1.0 ($p < .001$). According to Landis and Koch (1977), these Kappa statistics can be interpreted as ranging from substantial agreement to almost perfect agreement. Additionally, the outputs from the automated marking script were continually checked to ensure the consistency of the mapping, as well as to handle any unexpected answers from students. However, it was not possible to use Kappa to provide an interrater reliability analysis in this case, as the assumption of independence was violated because the script simply matched answer and misconception combinations that were provided.

Stage 2 – Dataset Splitting

In order to effectively evaluate the real-world predictive power of any potential models, the next stage involved extracting 30% of the data at random to form the holdout test set (Kuhn & Johnson, 2013; Raschka, 2018; Russell & Norvig, 2020). This data subset was completely isolated from the training process and was only used to test the final models, thus simulating their real-world performance, and making it possible to identify any models which overfitted the training data.

The holdout test set was extracted using SciKit-Learn's train_test_split method (Scikit-Learn, n.d.-ag). It was decided to use a randomly selected subset rather than using the results from a single academic year in order to produce an un-biased test set that takes into account nominal variations in teaching and assessments as well as any effects of the Covid-19 pandemic that occurred during data collection.

Stage 3 – Bayesian Knowledge Tracing Calculations

The next step in the data preparation process involved calculating the probabilities of students having appropriate mental models of each of the programming concepts being examined using Bayesian Knowledge Tracing. As mentioned in Section 3.3, it was necessary to establish the initial values of the four hyperparameters for each of the different programming concepts: L_0, G, S and T, which was achieved using a tool provided by Baker et al. (2010), which takes a brute force approach to fitting the hyperparameters. Baker et al.'s tool was chosen given the lead author's prevalence within the field of Educational Data Mining (including the use Bayesian Knowledge Tracing; BKT), as well as because of the way that the tool provides a simplistic method for establishing the initial values of the hyperparameters, which can then be applied to the BKT calculations. Nevertheless, fitting the BKT models for each programming concept is a manual process, so future work could explore how newer methods such as pyBKT (Badrinath et al., 2021) could be utilised in order to streamline the model development process.

To establish the initial hyperparameter values, the binarized responses for each mental model, which were mentioned in Stage 1, were converted to a compatible format to be used with Baker et al.'s (2010) tool. However, only responses contained within the training dataset were used to establish the initial values for each of the programming concepts, thus keeping the testing set completely isolated from the training process. Furthermore, any student who skipped 25% or more of the Programming Diagnostic questions was removed to prevent them from introducing bias into the dataset.

After the initial values for the hyperparameters had been found, the BKT equations shown in Section 3.3 were used to calculate the probability that a student had an appropriate mental model for each of the programming concepts, with the binarized responses being used to indicate whether the student answered questions relating to each concept correctly. The responses to each question were ordered in the same way that they were presented to the students during the Programming Checkup. This ensured that an appropriate estimation of students' mental models was established. The dataset was also checked for multiple responses, ensuring that only the first complete response was retained.

Equation 3.3 within the BKT calculations, accounts for the possibility that a student has learned a task whilst answering a question. It was debated whether to exclude this part of the

calculation, as when compared to intelligent tutoring systems, where BKT is typically employed, the Programming Checkup does not offer students any feedback on their answers. However, it was ultimately decided to remain consistent with the original BKT procedure and retain Equation 3.3. This decision was based on Corbett and Anderson's (1994) assumption that a student can make the transition from being in an unlearned state to being in a learned state at any opportunity where they can apply their knowledge, which, in this case, relates to their metal models of the concepts being examined.

Stage 4 – Dataset Preparations

Following the mental model estimations being incorporated into the dataset, several pre-processing steps were required prior to starting the model training process. These included:

Handling of missing data – Missing data are an unavoidable issue in real-world datasets (García et al., 2015; Kotsiantis et al., 2006). As mentioned previously, any student who had skipped 25% or more of the Programming Diagnostic questions was removed from the dataset. Furthermore, if a student did not provide a response to any of the other sections of the Programming Checkup and/or their assessment results were unavailable, they were also removed from the dataset. Although it is possible to predict missing values through methods such as "Most Common Feature Value" or "Mean Substitution" (García et al., 2015; Kotsiantis et al., 2006), it was felt that given the variability in students' backgrounds and behaviours, it would not be appropriate to attempt to estimate missing values. Additionally, the Programming Checkup requires students to provide answers to many of the sections before they can progress, meaning the number of responses being removed for having missing values was relatively low.

Data encoding – In order for different machine learning algorithms to process categorical features stemming from questions that could only be answered with either a "yes" or "no" (e.g., whether a student has previously studied a math-based subject), the responses needed to be binarized with 0 representing "no" and 1 representing "yes". However, when questioning whether students intended to work in a software engineering role after university, three answers were possible: "yes", "no" or "undecided". Therefore, in order to avoid introducing a potentially invalid ordering to the variable, one-hot encoding was used to split responses into three separate binarized variables, each representing one of the possible answers (Bruce & Bruce, 2017).

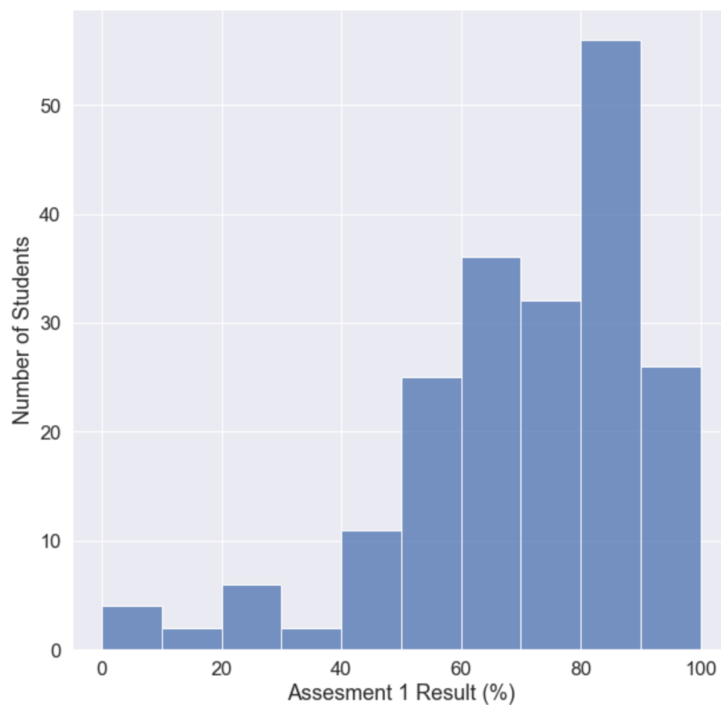
Feature Normalisation – A number of machine learning algorithms, such as K-Nearest Neighbour and Support Vector Machines, are significantly influenced by the scales of the input features (Hsu et al., 2008; Kotsiantis et al., 2006). Therefore, features are "normalised" using the MinMaxScaler (Scikit-Learn, n.d.-s) which converts all features into the range 0 to 1 whilst maintaining their original distributions (Esposito & Esposito, 2020; García et al., 2015).

Dependent Variable Preparation – As both regression and classification algorithms were being trialled, it was necessary at this point to duplicate the dataset to allow the dependent variables to be prepared for use with the different types of algorithms. As the Semester 1 assessment result was already a percentage, then no further processing was required for use with regression models. However, classification algorithms require continuous variables to be dichotomised into groups, and as binary classification was being carried out as part of this investigation, only two groups were possible – 0 and 1.

In order to binarize the assessment results, a threshold must first be established. A natural choice would be 40%, as was done so by Tomasevic et al. (2020), as this represents the minimum pass mark for the assessment. However, given the distribution of the assessment results (as shown in Figure 4.1), 50% was chosen as a more appropriate threshold. This is because the intended model was designed towards identifying students who are likely to require support, rather than to identify students who are likely to pass an assessment. Nevertheless, this still leaves a strong imbalance within the classification dataset, the consequences of which are discussed in Section 4.4.

Figure 4.1

Assessment 1 Grade Distribution within the Training Dataset



Feature Selection – The Programming Checkup collects data on a wide variety of factors. However, in order to reduce the likelihood of overfitting the training data, it is important to remove features that do not aid in generating predictions, thus reducing the dimensionality of the dataset and allowing for the algorithms to operate more effectively (Jovic et al., 2015; Kotsiantis et al., 2006).

Numerous automated methods of feature selection exist (Jovic et al., 2015; Kotsiantis et al., 2006; Kuhn & Johnson, 2019), but given that a core part of this research investigation was to explore how different aspects of the Programming Checkup contribute towards the predictive models, an approach was adopted that was inspired by the work of Tomasevic et al. (2020), whereby features are placed into categories which are trialled in different combinations to find the optimal model. The categories are as follows:

- Background Factors (BF) – Students' gender, prior experiences, whether they intend to work in software engineering, whether they consider themselves to be self-taught.
- Confidence (CO) – Estimations of how difficult students believe learning to program to be, how difficult they believe their degree to be, how difficult they find mathematics, how much they fear learning to program, their programming self-efficacy levels, how confident students are in their answers and their mental effort levels.
- Mental Models (MM) – Estimates of holding appropriate mental models of each concept established using Bayesian Knowledge Tracing.

Nevertheless, it was still necessary to remove any features that did not appear to be of benefit to the model. This was achieved by carrying out a series of statistical tests to examine the relationships between each of the individual features and the Assessment 1 results. However, this required tests to be carried out on pre-processed datasets for both classification and regression, as detailed below, due to the fact the Assessment 1 results were binarized for the classification dataset. Figures 4.2 to 4.4 present the distributions of each of the features within each category. These plots confirmed that the features were generally not normally distributed, and therefore, require the use of non-parametric statistical tests when analysing data.

Figure 4.2

Distributions of Results Relating to Students' Background Factors

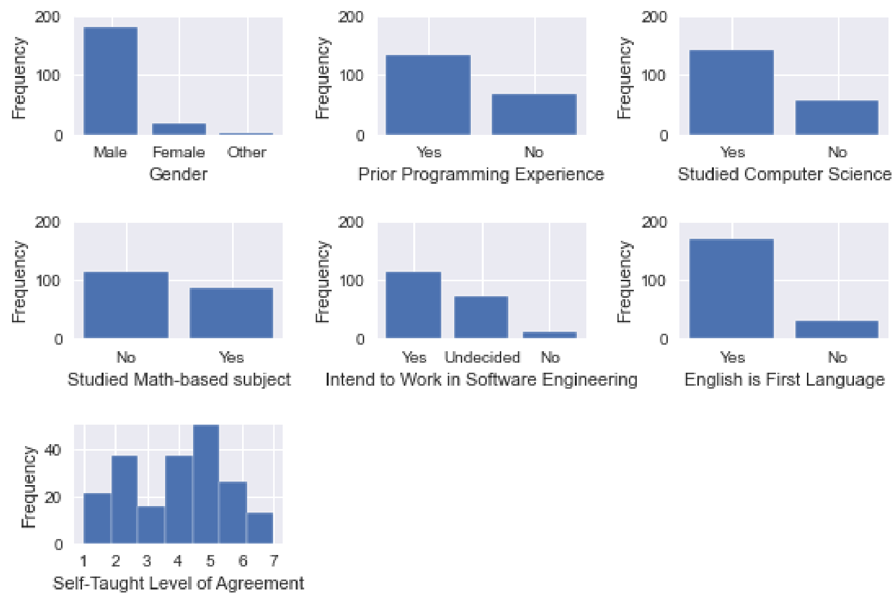
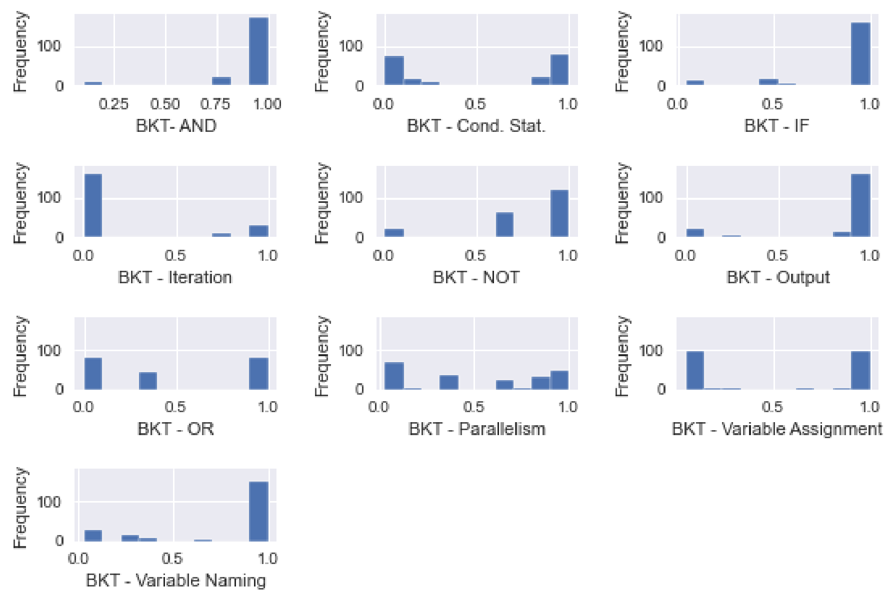


Figure 4.3

Distributions of Results Relating to Students' Confidence Factors

Figure 4.4

Distributions of Results Relating to Students' Mental Model Estimates Established Using Bayesian Knowledge Tracing



The Bonferroni correction was considered when conducting these tests, but given its overconservative nature, and following Cabin and Mitchell's (2000) recommendations, it was decided not to include the correction as to do so would likely exclude most features from the model. Most "Background Factors" are dichotomous features, including the one-hot encoded "Work in Software Engineering" responses, therefore, a chi-squared test can be utilised to examine the relationships between these features and the binarized Assessment 1 results for the classification dataset, as shown in Table 4.2. Naturally, a chi-squared test would not be appropriate for the regression dataset and therefore, Mann Whitney U tests were employed given the non-parametric nature of the data, as shown in Table 4.3.

Table 4.2

Chi-Squared Test Between Binarized Assessment 1 Results and Dichotomous Background Factors

Feature	X^2	p	V
Prior programming experience	0.80	.777	0.02
Previously Studied computer science	0.68	.410	0.06
Previously Studied mathematics-based subject	2.45	.117	0.11
Intend to work in software engineering – No	0.20	.653	0.03
Intend to work in software engineering – Undecided	2.96	.085	0.12
Intend to work in software engineering – Yes	3.58	.058	0.13

Note. The Chi-squared tests have been performed on the training dataset.

* "English is student's first language" violates Chi-Squared expected count assumption (Bruce & Bruce, 2017) therefore, a Fisher's Exact Test is performed yielding a result of p = .999.

* $df = 1$

Table 4.3

Mann Whitney U Tests Between Assessment 1 Results and Dichotomous Background Factors

Feature	*U*	*z*	*p*	*r*
Prior programming experience	3752.00	-1.82	.068	0.13
Previously Studied computer science	3515.00	-1.63	.104	0.12
Previously Studied mathematics-based subject	3977.00	-2.25	.024	0.16
Intend to work in software engineering – No	883.00	-1.26	.207	0.09
Intend to work in software engineering – Undecided	1800.50	-2.12	.207	0.15
Intend to work in software engineering – Yes	2807.50	-2.67	.034	0.19
English is student's first language	2265.00	-0.98	.330	0.07

Note. The Mann Whitney U tests have been performed on the training dataset.

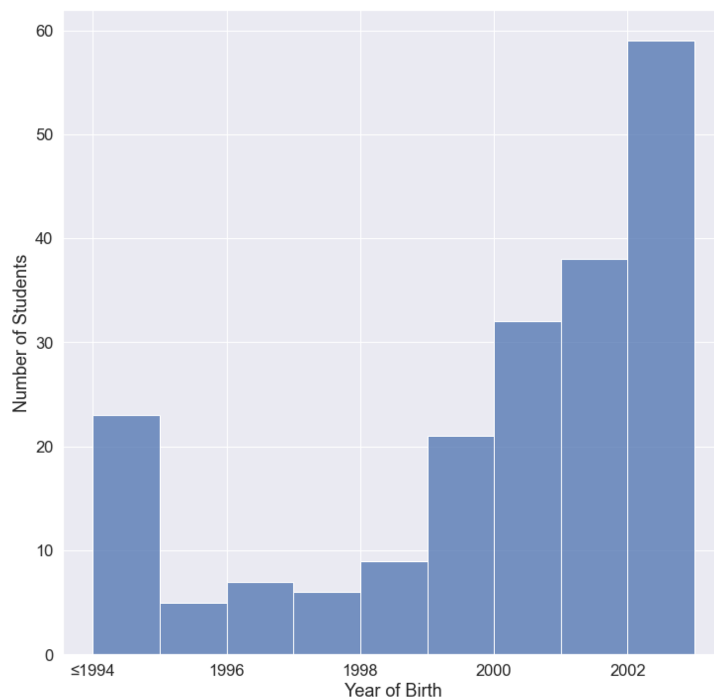
Students were also asked within the Programming Checkup to state how strongly they agreed or disagreed with considering themselves as being a "self-taught programmer" using a Likert scale. Subsequently, a Jonckheere-Terpstra test confirmed a significant relationship between how strongly a student agreed/disagreed that they were a self-taught programmer and their Assessment 1 result within the classification training dataset, $T_{JT} = 2770.00$, $z = 2.19$, $p = .029$, $r = 0.15$. This was then further confirmed within the regression training dataset with a significant correlation of $r_s = .32$, $p < .001$ being identified.

In order to maintain consistency, both classification and regression models used the same set of features, and therefore the results from both sets of tests needed to be taken into account when deciding what features to drop. Upon reviewing the analysis of the Background Factors tests a rather surprising result was that prior programming experience did not appear to have a significant influence on the Assessment 1 results. This could be due to the fact only 33.50% of students within the training datasets indicated they did not have prior programming experience, although several other factors including assessment design and amount of prior experience could also affect this result.

Furthermore, 71% of students stated that they had previously studied computer science. Although data were not collected on exactly what students had studied previously, it is possible that the high proportion of students previously studying computer science – and as such gaining experience in programming – was due, in part, to the resurgence of computer science within schools, as most students within the training set were born after the year 2000 (see Figure 4.5). As such, these students would have been in secondary school when the new computer science curriculum was introduced (Brown et al., 2014).

Figure 4.5

Year of Birth Distribution within the Training Dataset



However, the fact that most students within the training set had previously studied computer science and/or had prior programming experience, diminished the predictive powers of these features and, as such, meant that they were not included in the model.

More detailed information was required to draw any definitive conclusions as to whether the new computer science curriculum is directly influencing students' success at university level. However, questions about whether students have studied computer science at GCSE and/or A Level could easily be added to the Programming Checkup. It would also be prudent to ask

students to rate how much experience they have with studying computer science and programming on a scale, as opposed to simply answering "yes" or "no".

Interestingly, how strongly students considered themselves to be self-taught did appear to be a useful predictor and, as such, was retained for use within the model. Additionally, whether a student had previously studied a math-based subject post school level appeared to be a useful predictor given its relationship with the continuous Assessment 1 results. This supports claims that experience in mathematics aids students when learning to program (Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Wilson & Shrock, 2001), and was therefore retained for use within both the classification and regression models, despite the non-significant chi-squared test on the binarized assessment results. This was done to maintain consistency between the models.

Students' gender was deemed not to significantly influence their assessment results through a Kruskal Wallis test performed with the regression training set, $H(2) = 3.03$, $p = .220$, $\eta^2 = .005$, as well as a Fisher's Exact Test ($p = .543$), which was performed with the classification training set given the violation of the Chi-Squared expected count assumption (Bruce & Bruce, 2017). However, as 90% of students within the training set were male, it is difficult to make any reliable conclusions on the influence of gender on students' assessment results. Similarly, it was not possible to draw any reliable conclusions regarding the influence of students' first language on their assessment results, as only 15% of students within the training set indicated that English was not their first language. As such, neither of these two features were chosen to be included within the model. However, future studies being conducted with larger numbers of students, potentially from institutions in different countries, should seek to explore the impacts of gender and native language on programming abilities, as previous research has highlighted the additional difficulties that non-native English speaking students face when attempting to learn to program (Guo, 2018; Raj et al., 2017), as well as how female students have been observed to outperform males, despite generally having lower levels of self-efficacy (Lishinski et al., 2016; Quille et al., 2017).

As mentioned previously, the responses for "Work in Software Engineering" were one-hot encoded in order to avoid introducing a potentially invalid ordering to the variable, consequently resulting in three separate features for each of the responses – "yes", "no" and

“undecided”. Both the features representing “yes” and “undecided” appeared to be useful predictors, whereas the feature representing “no” did not. However, “no” was not removed from the model given that this would “break the symmetry of the original representation and therefore induce a bias” into the model (Scikit-Learn, n.d.-v).

To summarise, the features included within the Background Factors category were:

- Whether students have studied a mathematics-based subject after leaving school.
- Whether students intended to pursue a career in software engineering (all associated features).
- How strongly students considered themselves self-taught programmers.

The features within the Confidence category all yielded continuous results, and therefore Mann Whitney U tests could be utilised for examining relationships within the classification training set (Table 4.4) and Spearman’s Rank correlation tests could be used with the regression training set (Table 4.5).

Table 4.4

Mann Whitney U Tests Between Binarized Assessment 1 Results and Confidence Factors

Feature	*U*	*z*	*p*	*r*
Estimation of how difficult learning to program is	1699.00	-1.83	.068	0.13
Estimation of how difficult they find mathematics	1374.50	-3.04	.002	0.21
Estimation of how difficult their degree is	2164.50	-0.09	.930	0.01
How much they fear learning to program	1391.50	-2.97	.003	0.21
Self-Efficacy Factor 1 (Independence and Persistence)	1931.00	-0.95	.343	0.07
Self-Efficacy Factor 3 (Self-Regulation)	1708.50	-1.78	.076	0.13
Self-Efficacy Factor 4 (Simple Programming Tasks)	1364.00	-3.04	.002	0.21
Confidence – Variable Assignment	1654.00	-1.97	.048	0.14
Confidence – Conditional Statements	1554.00	-2.34	.019	0.17
Confidence – Iteration	1571.50	-2.28	.023	0.16
Confidence – All Questions	1538.00	-2.40	.016	0.17
Mental Effort – Variable Assignment	1562.00	-2.12	.034	0.15
Mental Effort – Conditional Statements	1664.00	-1.73	.084	0.12
Mental Effort – Iteration	1930.00	-0.70	.484	0.05

Note. The Mann Whitney U tests have been performed on the training dataset.

Table 4.5

Spearman's Rank Correlation Tests Between Assessment 1 Results and Confidence Factors

Feature	r_s	p
Estimation of how difficult learning to program is	-.16	.028
Estimation of how difficult they find mathematics	-.14	.052
Estimation of how difficult their degree is	.03	.673
How much they fear learning to program	-.31	<.001
Self-Efficacy Factor 1 (Independence and Persistence)	.24	<.001
Self-Efficacy Factor 3 (Self-Regulation)	.15	.031
Self-Efficacy Factor 4 (Simple Programming Tasks)	.42	<.001
Confidence – Variable Assignment	.32	<.001
Confidence – Conditional Statements	.31	<.001
Confidence – Iteration	.39	<.001
Confidence – All Questions	.38	<.001
Mental Effort – Variable Assignment	-.11	.123
Mental Effort – Conditional Statements	-.03	.728
Mental Effort – Iteration	-.04	.584

Note. The Spearman's Rank tests have been performed on the training dataset.

Upon reviewing the results within Tables 4.4 and 4.5, the vast majority of features appeared to be potentially useful predictors, although several features stood out as being prime candidates for removal. For instance, students' estimation of how difficult their degree is going to be, performed poorly with both the classification and regression training datasets and as such was removed from the model.

Features which measure different aspects of students' confidence in programming were observed to have a strong relationship with performance within the Semester 1 assessment, thus supporting previous claims that students' anxiety levels surrounding learning to program can have a significant impact on their performance (Bergin & Reilly, 2005b; Wilson & Shrock, 2001). Students' estimations of how difficult they find mathematics also appeared to be a useful predictor, adding further support to the claimed relationship between mathematics and programming (Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Wilson & Shrock, 2001).

In order to help reduce the dimensionality of the model, it was decided to drop the category-specific confidence estimates in favour of an overall estimate, given its good performance with both the classification and regression datasets. Additionally, students' estimations of the amount of mental effort required to answer each of the categories of questions within the Programming Diagnostic portion of the Programming Checkup did not appear to be a very strong predictor, particularly for the Conditional Statement and Iteration categories. Consequently, the mental effort estimations were removed from the model. However, it should be noted that the poor performance may be due to the fact that students were only asked to estimate their mental effort after completing all of the questions rather than after each individual question, as was done with their confidence ratings. The features included in the Confidence category were therefore:

- Estimation of how difficult learning to program is
- Estimation of how difficult they find mathematics
- How much they fear learning to program
- Self-efficacy Factor 1 (Independence and Persistence)
- Self-efficacy Factor 3 (Self-Regulation)
- Self-efficacy Factor 4 (Simple Programming Tasks)
- Confidence (all questions)

As both Tables 4.6 and 4.7 show, most mental model estimations calculated using Bayesian Knowledge Tracing appeared to have strong relationships with the Semester 1 Assessment results for both the classification and regression dataset. However, in order to reduce the dimensionality of the model it was decided to drop the individual estimations for AND, OR,

NOT and IF and retain the estimation for Conditional Statements in their place as this accounts for each of the individual concepts within a single mental model. Therefore, the features included within the Mental Model category were as follows:

- BKT – Conditional Statements
- BKT – Iteration
- BKT – Output
- BKT – Parallelism
- BKT – Variable Assignment
- BKT – Variable Naming

Table 4.6

Mann Whitney U Tests between Binarized Assessment 1 Results and Mental Model Estimates Established Using Bayesian Knowledge Tracing

Feature	*U*	*z*	*p*	*r*
BKT – AND	1611.50	-2.31	.021	0.16
BKT – Conditional Statements	1109.50	-3.98	<.001	0.28
BKT – IF	1470.00	-2.66	.008	0.19
BKT – Iteration	1389.00	-3.18	.001	0.22
BKT – NOT	1760.50	-1.60	.109	0.11
BKT – Output	1469.00	-2.67	.008	0.19
BKT – OR	1339.50	3.16	.002	0.22
BKT – Parallelism	1413.50	-2.87	.004	0.20
BKT – Variable Assignment	1271.50	-3.45	<.001	0.24
BKT – Variable Naming	1617.50	-2.22	.026	0.16

Note. The Mann Whitney U tests have been performed on the training dataset.

Table 4.7

Spearman's Rank Tests Between Assessment 1 Results Mental Model Estimates Established Using Bayesian Knowledge Tracing

Feature	r_s	p
BKT – AND	.30	<.001
BKT – Conditional Statements	.40	<.001
BKT – IF	.33	<.001
BKT – Iteration	.49	<.001
BKT – NOT	.29	<.001
BKT – Output	.41	<.001
BKT – OR	.30	<.001
BKT – Parallelism	.43	<.001
BKT – Variable Assignment	.47	<.001
BKT – Variable Naming	.23	<.001

Note. The Spearman's Rank tests have been performed on the training dataset.

4.3 Model Evaluation and Testing

After completing the pre-processing stages described in the previous section it was then possible to begin the model evaluation process. As mentioned previously, the approach taken to identify the best classification and regression models was inspired by the work of Tomasevic et al. (2020), wherein different combinations of feature categories are trialled in order to find an optimal model. However, in addition to the previously described pre-processing steps, it was also necessary to address the imbalance within the classification training set. To account for the class imbalance, SMOTE (Synthetic Minority Over-sampling Technique; (Chawla et al., 2002) was utilised to artificially generate instances of the minority class (I.e., students who scored less than 50% on the Semester 1 assessment). SMOTE works by randomly selecting an existing instance of the minority class to form the basis of the new synthetic instances. Subsequently, several nearest neighbors that are of the same class are selected and used in combination with the original instance to perform a randomised interpolation in order to generate the new synthetic instances (Bruce & Bruce, 2017; Fernández et al., 2018).

In addition to oversampling the minority class using SMOTE, under-sampling is also performed through the removal of Tomek Links. These are pairs of instances on either side of the decision boundary that are nearest neighbors of each other (Batista et al., 2004; Ramentol et al., 2012). These are removed to improve the separability between the classes and to reduce the chance of overfitting (Batista et al., 2004; Ramentol et al., 2012).

This hybrid approach of performing both over-sampling and under-sampling was implemented using the SMOTETomek class within the Imbalanced-Learn Library (Lemaître et al., 2017). It should be noted that in order to prevent data leakage SMOTETomek along with the MinMaxScaler were implemented using pipelines (Imbalanced-learn, n.d.-a; Scikit-Learn, n.d.-w), thus ensuring that models were only trained using the training data. Furthermore, in order to ensure consistency when training models, an integer seed was used in order to set the random_state hyperparameter of SMOTETomek (Imbalanced-learn, n.d.-b).

Ten-fold cross validation was utilised when training the regression and classification models as shown in Tables 4.8 and 4.9, respectively. The performance of the regression models shown in Table 4.8 was measured using Root Mean Squared Error (RMSE), which is one of the most common metrics for comparing regression models (Bruce & Bruce, 2017). RMSE uses the same units as the variable being predicted, meaning an RMSE of 0.177 would equate to 17.7 marks once multiplied by 100, therefore, models which have a lower RMSE are performing better than those with a higher RMSE.

A Receiving Operating Characteristic (ROC) curve is commonly used to summarise the performance of a classification algorithm across a range of thresholds, which trade-off between the true positive and false positive rates (Bruce & Bruce, 2017; Chawla et al., 2002; Swets, 1988). However, ROC curves do not offer a single measurement of performance that would allow for direct comparison between algorithms (Bruce & Bruce, 2017). Instead, the Area Under the Curve (AUC) metric uses the total area under the ROC curve to evaluate the performance of an algorithm.

The AUC of a model represents the probability of accurately identifying the correct classes arising when the model is presented with random examples of both classes, thus allowing for direct comparisons of performance to be made between different models (Baker, 2020). Therefore, an AUC of 1.0 represents a perfect classifier whereas an AUC of 0.5 indicates that the classifier is unable to distinguish between the two classes. This metric was subsequently used to evaluate the classification algorithms presented in Table 4.9.

Table 4.8

10-Fold Cross Validation Scores of Regression Models (RMSE)

Regression Model	Feature Combinations						
	BF	CO	MM	BF + CO	BF + MM	CO + MM	BF + CO + MM
OLS Linear Regression	0.1925	0.1862	0.1821	0.1859	0.1829	0.1849	0.1864
Elastic Net	0.1923	0.1856	0.1796	0.1850	0.1791	0.1788	0.1791
Lasso Regression	0.1923	0.1862	0.1813	0.1859	0.1816	0.1822	0.1828
Ridge Regression	0.1924	0.1857	0.1799	0.1852	0.1796	0.1796	0.1801
KNN Regressor – Uniform Weighting	0.1967	0.1894	0.1779	0.1898	0.1828	0.1796	0.1770
KNN Regressor – Distance Weighting	0.2077	0.1885	0.1909	0.1889	0.1855	0.1799	0.1776
Bayesian Linear Regression	0.1930	0.1857	0.1802	0.1857	0.1797	0.1794	0.1795
SVR - RBF	0.1906	0.1850	0.1772	0.1822	0.1788	0.1790	0.1783
SVR - Linear	0.1919	0.1854	0.1816	0.1831	0.1802	0.1817	0.1834
Regression Tree	0.1927	0.1855	0.1868	0.1855	0.1868	0.1965	0.1965
Random Forest Regressor	0.1872	0.1817	0.1779	0.1841	0.1786	0.1773	0.1782
Bagging Decision Tree Regressor	0.2053	0.1981	0.1966	0.1918	0.1935	0.1900	0.1902
Gradient Boost Regressor	0.1919	0.1869	0.1868	0.1871	0.1863	0.1847	0.1850
XGBoost Regressor	0.1917	0.1867	0.1782	0.1855	0.1774	0.1784	0.1791
MLPRegressor	0.1931	0.1892	0.1797	0.1868	0.1796	0.1790	0.1822

Note. Lower RMSE values (highlighted green) represent better performance.

BF = Background Factors, CO = Confidence, MM = Mental Models

Table 4.9

10-Fold Cross Validation Scores of Classification Models (AUC)

Classification Model	Feature Combinations						
	BF	**CO**	**MM**	**BF + CO**	**BF + MM**	**CO + MM**	**BF + CO + MM**
Logistic Regression	0.6585	0.7211	0.7454	0.6779	0.7476	0.7420	0.7258
Ridge Classifier	0.6440	0.6980	0.7250	0.6732	0.7209	0.7252	0.7139
SVC - Linear	0.6585	0.7065	0.7275	0.6799	0.7167	0.7363	0.7147
SVC - RBF	0.6665	0.7002	0.7355	0.6822	0.7288	0.7317	0.7150
Decision Tree	0.6572	0.6663	0.6840	0.6284	0.7185	0.7011	0.7185
Bagging Decision Tree	0.5621	0.5841	0.6786	0.6038	0.7252	0.6811	0.7350
Random Forest	0.7167	0.7533	0.7341	0.7297	0.7211	0.7771	0.7783
KNN - Uniform Weighting	0.5779	0.6552	0.7612	0.6733	0.6454	0.7515	0.6529
KNN - Distance Weighting	0.5357	0.6449	0.7459	0.6954	0.6348	0.7438	0.6560
Gradient Boost	0.6551	0.7271	0.7114	0.6688	0.7289	0.7464	0.7663
XGBoost Classifier	0.6474	0.7058	0.7346	0.7096	0.7258	0.7020	0.6951
MLPClassifier	0.6022	0.6715	0.7002	0.6770	0.7025	0.7217	0.6956

Note. Higher AUC values (highlighted green) represent better performance.

BF = Background Factors, CO = Confidence, MM = Mental Models

The results presented in Table 4.8 range from an RMSE of 0.2077 (KNN Regressor – Distance Weighting, Background Factors) to 0.1770 (KNN Regressor – Uniform Weighting, Background Factors, Confidence and Mental Models). Furthermore, the results of the classification models shown in Table 4.9 range from an AUC of 0.5357 (KNN Distance Weighting, Background Factors) to 0.7783 (Random Forest, Background Factors, Confidence and Mental Models). A variety of feature combinations resulted in optimal performance for each of the models with almost all of the best performing models incorporating students' mental models as input features, with a significant number of models also including measures of confidence and/or students' background factors. However, it was necessary to estimate the real-world performance of the models using the testing dataset before any firm conclusions could be made (Russell & Norvig, 2020). This was because the hyperparameter tuning process may have produced models that overfitted the training data and, as such, would not perform well with new, unseen data.

In order to prevent the test dataset from being overfitted by repeatedly testing different models, a single classification model and a single regression model were selected, along with a corresponding feature combination, using the training results presented in Tables 4.8 and 4.9. The classification model chosen to be applied to the testing dataset was Random Forest with a Feature Combination of Background Factors, Confidence and Mental Models, given that this produced the best performance on the training dataset. Additionally, as Random Forest is an ensemble method it is less prone to overfitting (Dietrich et al., 2015; James et al., 2013), as described in Section 4.2, therefore making it an appropriate choice for testing with the holdout test set.

The regression model which performed the best on the training dataset was the KNN Regressor using Uniform Weighting and a combination of Background Factors, Confidence and Mental Models features, which produced an RMSE of 0.1770. However, two other models exhibited similar levels of performance, namely, Support Vector Regressor using the RBF Kernel and Mental Model features (0.1772) and Random Forest Regressor using a combination of Confidence and Mental Model features (0.1773). Owing to the closeness in performance on the training dataset, it was felt that Random Forest Regressor using a combination of Confidence and Mental Models features, was the most appropriate choice in order to minimise the potential for overfitting, as explained previously, especially given the fact that the dataset for this investigation was relatively constrained in size.

To evaluate the performance of the models on the test set, the chosen regression and classification models were trained on the entire training dataset (i.e., not having a split for the validation set), using the optimal hyperparameters that were previously identified by GridSearchCV. The same random_state value for SMOTETomek as used previously was used again and, as before, SMOTETomek was only applied to the training data and not to the test set. Once the models were trained, they could then be tested on the unseen data held within the holdout test set, by attempting to make predictions for the samples within the test set and comparing the results. Given the random nature of Random Forests this process is repeated three times in order to obtain an averaged measure of the performance of the models. The results were as follows:

Random Forest Regressor (Confidence and Mental Models)

Average Training RMSE:	0.1686	SD: 0.0007
Average Testing RMSE:	0.1687	SD: 0.0009

Random Forest Classifier (Background Factors, Confidence and Mental Models)

Average Training AUC:	0.7400	SD: 0.0084
Average Testing AUC:	0.6595	SD: 0.0131

As can be seen, the average training AUC for the Random Forest classifier after being trained on the entire training dataset fell from 0.7783 to 0.7400. The performance of the Random Forest Regressor, however, improved from an RMSE of 0.1773 to 0.1686 when trained on the entire training dataset.

Both the classification and regression models experienced a drop in performance when they were evaluated using the unseen data held within the testing dataset as compared to the training dataset. Although a drop in performance is to be expected (Géron, 2022), there may be a degree of overfitting of the training set taking place, particularly for the Random Forest Classifier. However, the results do demonstrate a reasonable level of generalisability for both the classification and regression models, which would likely be improved by additional data being included in both the training and testing datasets.

The Scikit-Learn implementations of the Random Forest Regressor and Classifier allow for the evaluation of how much each feature contributes to the performance of the model through the use of the "Feature Importance" attributes (Scikit-Learn, n.d.-y, n.d.-x). As such, Figures 4.6 and 4.7 present the feature importance plots for each of the classification and regression models respectively.

Figure 4.6 *Random Forest Regressor Feature Importance Plots*

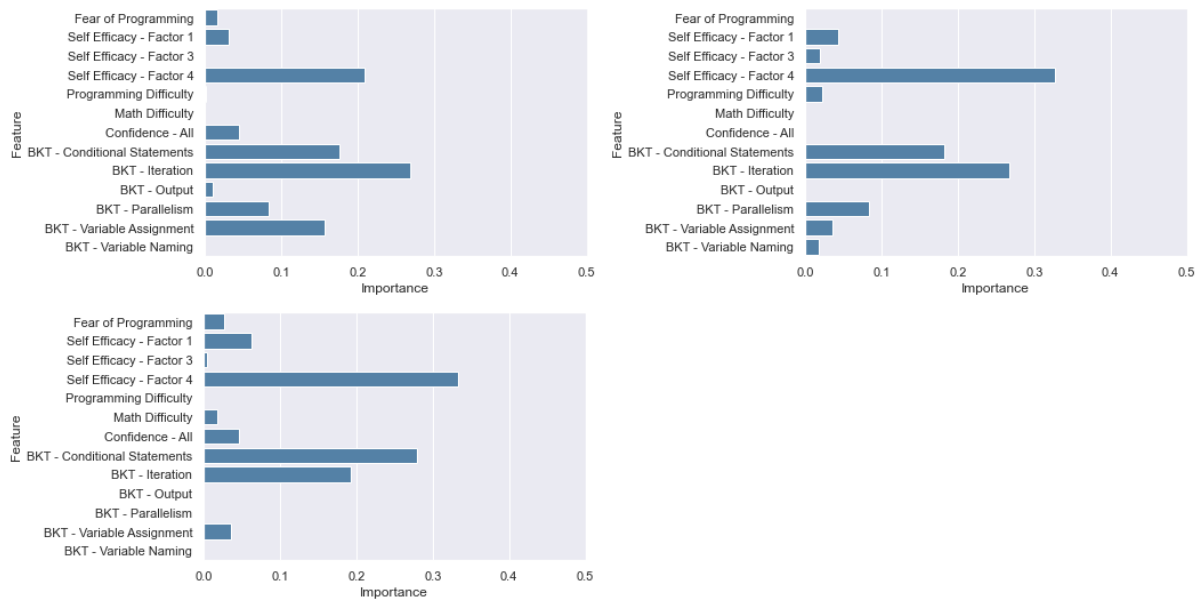
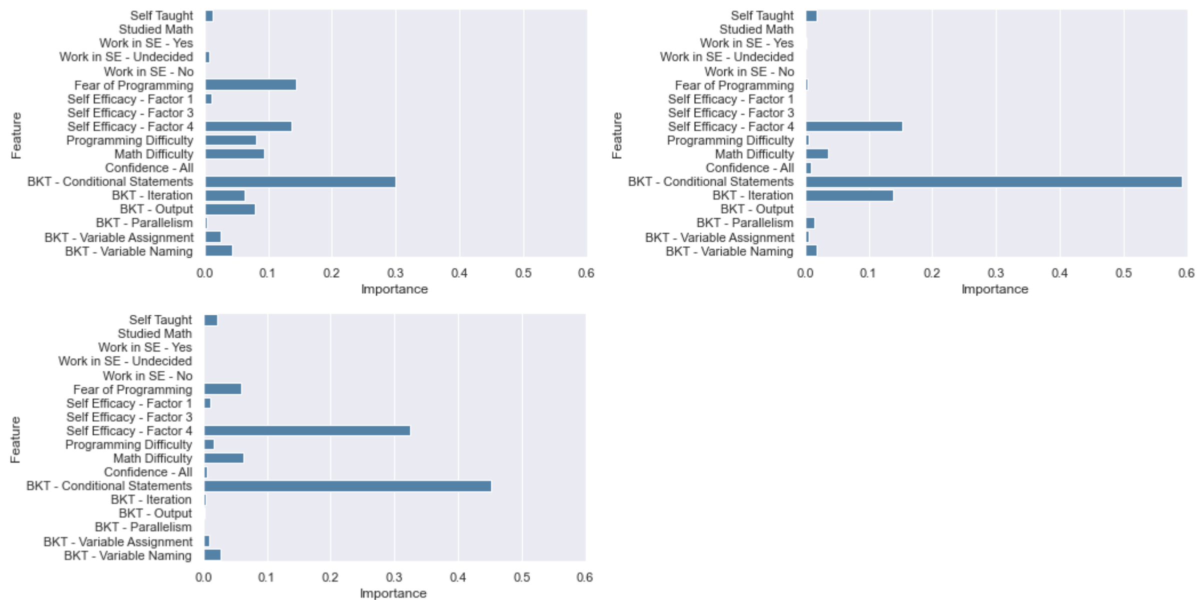


Figure 4.7 *Random Forest Classifier Feature Importance Plots*



The importance scores for each model are normalised, subsequently meaning that the combined importance scores for all features are equal to 1. As such, features with higher importance scores are seen to be having a larger effect on the model (Scikit-Learn, n.d.-y, n.d.-x).. However, it is important to note that whilst the feature importance plots provide useful insight into which features are contributing to the performance of the models, they are specific to a given model and do not allow for any conclusions to be established pertaining to the relationship between the feature and the variable being predicted. Given the random nature of Random Forests, the level of importance for each feature differs between each trial of the model, hence the need for averaging the performance results. However, a number of features were seen to be consistently important to the models. For the Random Forest Regressor (Figure 4.6), students' mental model estimates for Conditional Statements and Iteration, as well as their levels of Self-Efficacy pertaining to completing simple programming tasks (Factor 4), were consistently shown to be important features across all three trials of the model. Additionally, the feature importance plots for the Random Forest Classifier (Figure 4.7) indicate a substantial reliance on students' estimates for holding an appropriate mental model of Conditional Statements within the classification models. Students' levels of Self-Efficacy associated with completing simple programming tasks also appear to relatively important to the classifier models. However, the degree of overfitting which has been observed limits the usefulness of the model and the data presented within Figure 4.7.

4.4 Summary

This chapter has described the process by which the classification and regression models were developed in response to RQ 3. The steps that were required to take the raw output from the Programming Checkup and prepare the data for use in developing the models have been systematically detailed in order to aid reproducibility. Furthermore, a selection of regression and classification methods have been described and subsequently evaluated using the training data, which ultimately culminated in the selection of Random Forest for use in both the regression and classification models. The estimate of the real-world performance of the final models, produced by using the hold-out test set to test the final models, suggested a reasonable level of generalisability, the implications of which, in relation to RQ 3, will be discussed in Section 6.2.

5. Programming Checkup Analysis

5.1 Analysis Scope

In the previous chapter, the testing set was completely isolated from any statistical analysis to enable it to inform the model-development process. However, in order to gain a full understanding of the trends in students' responses to the Programming Checkup as well as to aid in answering the research questions upon which this investigation was based, the test set that was previously isolated from any analysis will now be included with the rest of the data. After the removal of any students who skipped 25% or more of the questions within the programming diagnostic section of the Programming Checkup, the remaining dataset available for analysis consisted of 285 students.

The analysis presented in this chapter will first involve an examination of how students' responses changed between T1 and T2. Subsequently, an examination will be reported of how students' responses relate to their Assessment 1 results, as this is the outcome variable being predicted by the model developed in the previous chapter. In addition, an examination of students' Assessment 2 results will be reported as a comparison dataset to investigate how students' responses to the Programming Checkup related to later performance in their introductory programming module. It should be noted that all analyses presented in this chapter were carried out after the model development process described in the previous chapter to prevent any of the results from influencing the decisions being made.

5.2 T1 and T2 Comparison

Out of the 285 students who completed the Programming Checkup at T1 at the start of the academic year, 119 students also completed the Programming Checkup at T2 at the end of the first semester. The comparison between students' performance at T1 and T2 will form the focus of the analysis in this section.

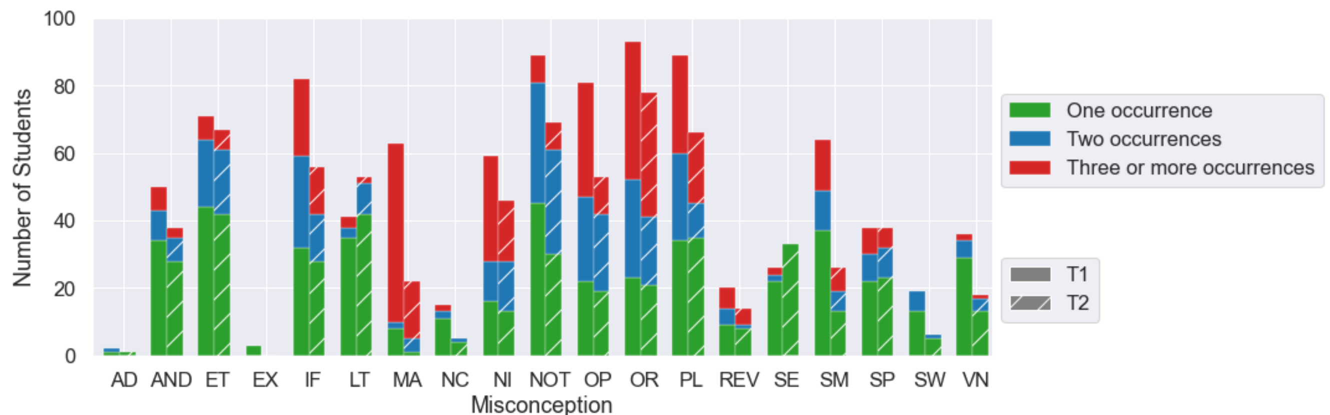
5.2.1 Analysis of Students' Understandings of Core Programming Concepts

Figure 5.1 present the frequency that each misconception was demonstrated by students at T1 and T2, respectively (refer to Appendix B for a description of each misconception). To aid in the visualisation of the prevalence of each misconception, students were divided into three groups based on the number of times they had demonstrated a particular misconception, that is, one occurrence, two occurrences or three or more occurrences. This categorisation

therefore provides an indication of the strength of students' misconceptions, as a strong and more engrained misconception would likely be demonstrated multiple times when answering questions.

Figure 5.1

Distribution of Misconceptions and the Frequency of Occurrences Per Student (Who Completed Both Programming Checkups) at T1 and T2



It is important to note that it is not possible to statistically compare the frequency of each misconception due to the variability in the number of opportunities to demonstrate each misconception. For example, it is only possible for a student to demonstrate the Multiple Assignment (MA) misconception in seven of the nine variable assignment questions, whereas the Output (OP) misconception could be demonstrated in almost all of the questions included within the Programming Diagnostic section of the Programming Checkup. This is due to questions being repeated to examine different misconceptions, such as repeating a Variable Assignment question and changing the names of the variables to MAX and MIN to examine the Variable Name (VN) misconception. Concepts such as Program Output form a key part in the overall design of questions, leading to more opportunities for the associated misconception (OP) to be demonstrated than others. However, the design of the Programming Diagnostic section ensures each misconception is examined a minimum of five times in order for estimations to be made about students' mental models using Bayesian Knowledge Tracing.

When comparing the distributions of misconceptions at T1 and T2 within Figure 5.1, it is clear that there has been an overall decrease in the number of misconceptions being exhibited

by students. However, prior to any in-depth statistical tests taking place, Shapiro-Wilk tests were used to examine the frequency distribution of the misconception occurrences, the results of which confirmed that none of the misconception occurrences were normally distributed. Therefore, in order to identify whether there had been any significant changes between T1 and T2, a series of Wilcoxon Signed-Rank tests were carried out, as shown in Table 5.1.

Table 5.1

Wilcoxon Signed Rank Comparison of Misconception Occurrences at T1 and T2

Misconception	z	p	r
AD	-0.81	.414	0.08
AND	-2.56	.010	0.24
ET	-0.57	.572	0.05
EX	-1.73	.083	0.16
IF	-3.75	<.001	0.34
LT	-0.22	.826	0.02
MA	-5.52	<.001	0.51
NC	-2.59	.010	0.24
NI	-2.19	.029	0.20
NOT	-2.14	.032	0.20
OP	-4.59	<.001	0.42
OR	-2.45	.014	0.22
PL	-3.03	.002	0.28
REV	-0.93	.355	0.09
SE	-0.28	.782	0.03
SM	-4.04	<.001	0.37
SP	-0.20	.839	0.02
SW	-2.71	.007	0.25
VN	-2.32	.020	0.21

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 19) were deemed to be reliable for the purposes of interpretation.

*n = 119

The misconception that showed the most change between T1 and T2 was the Multiple Assignment (MA) misconception, $z = -5.52$, $p < .001$, $r = 0.51$, suggesting that whilst this is a significant issue at the beginning of the course, most students are able to overcome this misconception by the time they reach T2. However, as Figure 5.1 indicates, the vast majority of students who still exhibit the MA misconception at T2 do so three or more times, suggesting that these students have developed a deeply engrained misconception which would require direct intervention to overcome. The other misconceptions associated with Variable Assignment do experience a drop between T1 and T2, although none are as significant as MA. However, a small number of students do appear to be exhibiting the Reverse (REV) misconception, with some demonstrating it three or more times at both T1 and T2. Although the REV misconception may not be widespread, students who do exhibit it frequently might require additional support to overcome it.

The occurrence of the IF misconception also showed a significant decrease between T1 and T2, $z = -3.75$, $p < .001$, $r = 0.34$. Furthermore, half of the students who demonstrated this misconception at T2 only did so on one occasion. There also did not appear to be any significant change in the occurrences of misconceptions associated with Boolean Logic, AND, OR and NOT, between T1 and T2, with the occurrences of OR and NOT remaining particularly high, giving further credence to Grover and Basu's (2017) claims that Boolean Logic is a topic of difficulty for students.

All misconceptions associated with the concept of Iteration did not change significantly between T1 and T2, with the exception of Summation (SM), which dropped significantly between T1 and T2, $z = -4.04$, $p < .001$, $r = 0.37$. This may indicate that some struggling students were beginning to grasp the idea that all lines within the loop are repeated. However, the observed improvement may in fact be related to students developing a better understanding of program output, as the occurrence of students demonstrating misconceptions relating to what is produced by the output statements within the code examples (i.e., outputting a variable name instead of the value) decreased significantly between T1 and T2 (OP, $z = -4.59$, $p = <.001$, $r = 0.42$). The occurrence of SM misconceptions was significantly, albeit weakly, correlated with that of OP at both T1 ($r_s = .19$, $p = .001$) and T2 ($r_s = .33$, $p < .001$) after a Bonferroni correction had been taken into account, which reduced the significance threshold to $p < .025$. Interviews and think aloud

exercises should be conducted as part of a future investigation in order to establish a clearer picture as to why students struggle with iteration and exhibit particular misconceptions, as the results thus far suggest that iteration is a difficult concept for students to grasp.

There was also a significant drop in the number of occurrences of the Parallelism (PL) misconception, wherein students demonstrated a misunderstanding of the flow of control within a program, with the majority of students only exhibiting it once at T2. However, almost a third of students demonstrated PL three or more times, indicating that some students may have been struggling to overcome this misconception.

In addition to the changes in the occurrences of misconceptions, a significant drop in the occurrence of incorrect answers that could not be mapped to a specific misconception (recorded as NA) was observed and confirmed with a Wilcoxon Signed Rank Test, $z = -5.32$, $p < .001$, $r = 0.49$. Although it is possible that some of the unmappable answers provided by students were genuine mistakes, they do demonstrate that some students initially struggled to appropriately comprehend some of the concepts being examined within the Programming Diagnostic questions. This view is supported by the significant reduction in unmappable answer occurrences by T2, as although some students might still have been struggling to fully grasp the concepts being examined, they were at least making progress towards establishing an appropriate mental model.

No significant differences were found in the number of questions being skipped by students at T1 and T2, $z = -1.58$, $p = .115$, $r = 0.01$. However, this may be due in part to the fact that students who skipped 25% or more of the Programming Diagnostic questions were removed from the analysis process.

Aside from examining the frequencies of individual misconception occurrences, it is possible to obtain a more direct estimation of whether students hold appropriate mental models of each of the concepts examined within the Programming Checkup (see Table 4.1) through the use of Bayesian Knowledge Tracing (BKT). As described in the previous section, BKT was utilised to provide estimates of students' mental models at T1 in order to be used within the classification and regression models. The same process was used here to produce estimates of students' appropriate mental models at T1, and subsequently at T2. It should be noted that the same hyperparameter values for L_0, G, S and T were used at both T1 and T2 in order to

provide a consistent basis for comparison between the two tests. Furthermore, T2 was treated independently from T1, meaning that in terms of the BKT calculations, the first question at T2 was treated as $n = 0$, rather than continuing on from T1.

It is believed that this is the first time BKT has been utilised to analyse an aptitude test (or similar) in this way. A future study could investigate how estimates of the appropriateness of students' mental models are affected by examining combined responses at T1 and T2 whilst possibly incorporating some of the more recent extensions to BKT such as the KT-Forget parameter as suggested by Qiu et al. (2011). The latter parameter accounts for the possibility that students forget previously learned knowledge after several days between interactions with an intelligent tutoring system. Although the time between T1 and T2 was several weeks, during which the concepts being examined were introduced and reinforced, it would be a worthwhile exercise to explore if KT-Forget can be adapted for use in such a context. Furthermore, a future study could also examine mental model development throughout the course of a full academic year.

Figure 5.2 show the proportion of students who are deemed likely (green) or unlikely (red) to hold an appropriate mental model of each concept, given a threshold of 0.5, at T1 and T2. Additionally, a series of Wilcoxon Signed Rank tests were carried out on the raw probabilities established by BKT in order to confirm the significance of any changes between T1 and T2, as presented in Table 5.2.

Figure 5.2

Estimates of Whether Students Hold Appropriate Mental Models at T1 and T2, Established Using Bayesian Knowledge Tracing with a Threshold of 0.5

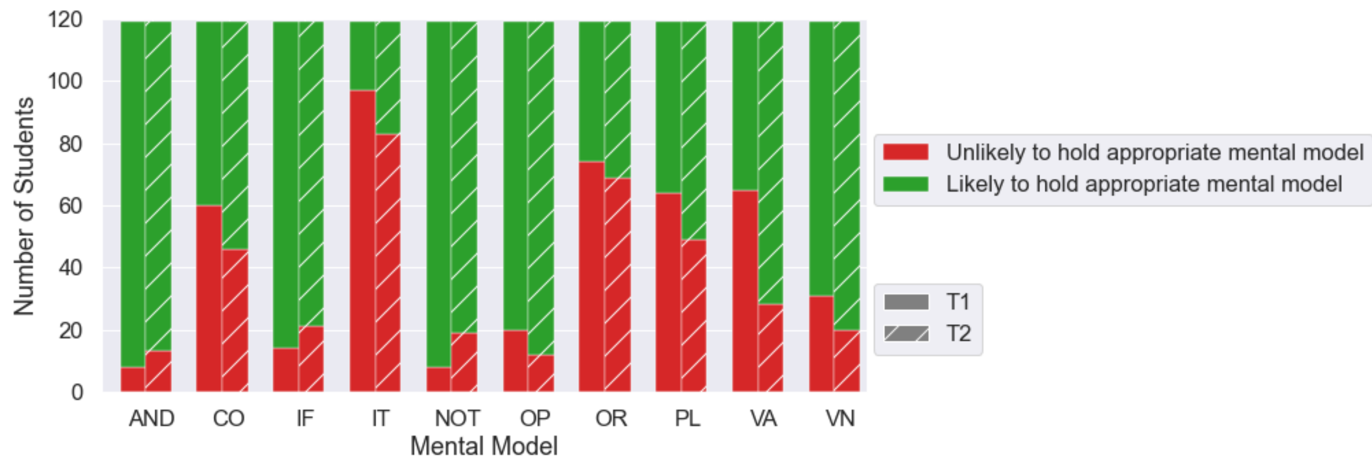


Table 5.2

Wilcoxon Signed Rank Comparison of Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2

Mental Model	***z***	***p***	***r***
AND	-0.70	.481	0.07
Conditional Statements	-2.63	.008	0.24
IF	-0.09	.929	0.01
Iteration	-5.25	<.001	0.48
NOT	-0.90	.371	0.08
Output	-3.15	.002	0.29
OR	-0.51	.613	0.05
Parallelism	-1.61	.107	0.15
Variable Assignment	-5.10	<.001	0.47
Variable Naming	-1.62	.106	0.15

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .005$ (i.e., .05 divided by 10) were deemed to be reliable for purposes of interpretation.

*n = 119

One of the most significant changes between T1 and T2 was Variable Assignment (VA), with over half of the students being estimated as being unlikely to hold an appropriate mental

model. However, by T2 over 75% of students were estimated to hold an appropriate mental model, suggesting that many students had been able to overcome their difficulties, in particular with the MA misconception, with the significant change being confirmed by the Wilcoxon Signed Rank Test as shown in Table 5.2. Additionally, a significant change was identified within the estimates produced by BKT for the Output (OP) mental model. However, this is likely to indicate the strengthening of students' models rather than a transition from an inappropriate model to an appropriate one, given that the percentage of students who are estimated not to be holding an appropriate model only falls from 17% at T1 to 10% by T2.

Variable Assignment and Output are two mental models that students utilise in almost every program they write, so it is not surprising to see that the vast majority of students had developed an appropriate mental model by T2. However, those who had not developed an appropriate model, particularly for VA, would be likely to require direct support to overcome their issues. Additionally, the majority of students were estimated to have an appropriate model for Variable Naming (VN) at both T1 and T2, with most students demonstrating a potential misconception only once, as per Figure 5.1. The Wilcoxon Signed Rank test within Table 5.2 for VN did not reveal a change between T1 and T2.

The probability estimates of students holding appropriate mental models for AND, OR and NOT, as well as If Statements, did not change significantly between T1 and T2, with OR having one of the highest proportions of students estimated to be unlikely to hold an appropriate model. As shown in Table 4.1, the Conditional Statements mental model (CO) combines all Conditional Statement questions together, thus providing a broader estimate of students' models by encompassing each of the individual concepts within a single mental model. The change in the probability estimates of students holding an appropriate model for CO approached significance after applying the Bonferroni Correction. However, 39% of students were still estimated to be unlikely to hold an appropriate model for CO. These estimates further support claims that Boolean Logic is a difficult concept for students to grasp (Grover & Basu, 2017), with OR appearing to the main point of confusion.

A significant change was observed in the estimates that students hold an appropriate model for Iteration (IT) between T1 and T2. However, a substantial number of students were still classified as being not likely to hold an appropriate model at T2, with 67% at T2 being

classified as unlikely to hold an appropriate model due to their estimated probability being less than 0.5, compared to 81% at T1. Although this indicated that only a relatively small number of students had improved their models sufficiently to cross the 0.5 threshold, students did appear to be improving their models but were still struggling with misconceptions, as indicated in Figure 5.1. Additionally, there did not appear to be a significant change in the estimates of students' Parallelism (PL) model. Although there was a drop from 53% being classified as being unlikely to have an appropriate mental model at T1 to 41% at T2, some students appeared to be consistently struggling with this concept, as shown in Figure 5.1, suggesting direct support may be needed for them to develop an appropriate model.

To summarise, there was evidence of the development in students' mental models between T1 and T2, with students quickly acquiring appropriate models for concepts such as Variable Assignment. However, other concepts, such as Iteration, appeared to be more troublesome for students to grasp and develop an accurate model.

5.2.2 Influence of Prior Experiences on Likelihood of Holding Appropriate Mental Models

Given that data collection for T2 took place at the end of the first semester, students were not expected to have fully accurate models for all concepts at this stage. For many students, this was their first time learning to program and they were therefore constructing their models using what they believed to be relevant information, that is, their "pre-programming knowledge", as termed by Bonar and Soloway (1985). Over half (i.e., 63%) of students who participated in both T1 and T2 indicated that they had prior programming experience. However, 70% of students stated that they had previously studied computer science, whilst 39% of students considered themselves "self-taught programmers" prior to starting their degrees. Interestingly, 14% of students indicated that they had previously studied computer science, but did not have any prior programming experience, whereas only 7% of students indicated that they had prior programming experience but did not previously study computer science. As programming is a core component of computer science courses it would be useful for a future study to fully explore students' prior experiences with studying computer science and learning to program, especially with greater numbers of students now passing through the new computer science curriculum. It should also be noted that as only 23% of students

indicated that English was not their first language, then the sample size is insufficient to produce any statistically significant results in relation to this factor.

Tables 5.3 and 5.4 show whether there were significant differences at both T1 and T2 in the mental model estimates amongst students who had indicated that they had prior programming experience or had previously studied computer science. Perhaps not unsurprisingly, the average probability of having an appropriate mental model for each individual model at T1 was higher amongst students who had previous programming experience, as shown in Table 5.5.

Although having prior experience of programming had a substantial influence over students' mental models at T1, it did appear to decrease by T2 as the vast majority of mental models being evaluated by the Mann Whitney U test showed a slight decrease in influence at T2 when compared to the corresponding T1 result. However, interestingly, the influence of prior programming experience appeared to be significant on the NOT mental model at T2, as an increase in significance to the point which surpasses the corrected significance threshold was observed when compared to the T1 result.

When considered as independent factors, the influence of previously studying computer science on the estimates of students holding appropriate mental models for the concepts examined within the Programming Checkup was broadly similar to that of prior programming experience, as shown in Table 5.5. However, previously studying computer science appeared to have less influence on students' mental models than having prior programming experience, which is reflected in the effect sizes of the Mann Whitney U tests presented in Table 5.4.

Table 5.3

Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2

Mental Model	**T1**				**T2**			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
AND	1262.00	-2.39	.017	0.22	1281.50	-2.27	.023	0.21
Conditional Statements	855.50	-4.37	<.001	0.40	1093.00	-3.07	.002	0.28
IF	1037.00	-3.41	<.001	0.31	1216.50	-2.44	.015	0.22
Iteration	897.00	-4.40	<.001	0.40	962.50	-3.87	<.001	0.36
NOT	1189.00	-2.59	.010	0.24	1131.00	-2.92	.003	0.27
Output	797.50	-4.71	<.001	0.43	1233.00	-2.32	.020	0.21
OR	1106.00	-3.01	.003	0.28	1120.00	-2.95	.003	0.27
Parallelism	892.50	-4.18	<.001	0.38	1011.50	-3.55	<.001	0.33
Variable Assignment	794.00	-4.83	<.001	0.44	1285.00	-2.176	.030	0.20
Variable Naming	1293.50	-2.06	.040	0.19	1274.00	-2.29	.022	0.21

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 119

Table 5.4

Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Mental Model Estimates Established using Bayesian Knowledge Tracing at T1 and T2

Mental Model	**T1**				**T2**			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
AND	1281.00	-1.23	.218	0.11	1021.50	-2.93	.003	0.27
Conditional Statements	1008.50	-2.69	.007	0.25	1038.00	-2.52	.012	0.23
IF	1234.50	-1.39	.166	0.13	1158.00	-1.86	.063	0.17
Iteration	972.00	-3.11	.002	0.29	951.00	-3.10	.002	0.28
NOT	1149.50	-1.90	.057	0.17	1041.00	-2.56	.011	0.23
Output	1060.00	-2.40	.016	0.22	1030.00	-2.91	.004	0.27
OR	1137.00	-1.95	.051	0.18	1092.00	-2.26	.024	0.21
Parallelism	956.50	-3.01	.003	0.28	976.50	-2.90	.004	0.27
Variable Assignment	844.00	-3.74	<.001	0.34	1092.50	-2.38	.017	0.22
Variable Naming	1305.50	-1.01	.315	0.09	1033.00	-2.82	.005	0.26

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 119

Table 5.5

Comparison of Prior Programming Experience, Previously Studying Computer Science and Average Estimates of Having an Appropriate Mental Model at T1 and T2

Mental Model	T1				T2			
	Prior Programming Exp. - No	Prior Programming Exp. - Yes	Studied CS - No	Studied CS - Yes	Prior Programming Exp. - No	Prior Programming Exp. - Yes	Studied CS - No	Studied CS - Yes
AND	.903	.928	.912	.922	.826	.905	.761	.924
Conditional Statements	.290	.633	.335	.577	.451	.696	.479	.659
IF	.824	.908	.863	.883	.797	.872	.747	.885
Iteration	.071	.278	.116	.237	.156	.456	.140	.430
NOT	.762	.832	.776	.818	.641	.816	.605	.812
Output	.663	.926	.687	.888	.869	.918	.807	.939
OR	.349	.522	.338	.508	.318	.569	.334	.535
Parallelism	.306	.611	.345	.862	.410	.663	.411	.635
Variable Assignment	.268	.615	.290	.568	.678	.842	.650	.836
Variable Naming	.634	.854	.691	.806	.762	.892	.678	.913

Additionally, Table 5.6 presents the results from a series of Spearman's Rank correlation tests between students' levels of agreement in considering themselves "self-taught programmers" at the beginning of the course, and their estimates of having appropriate mental models. The results indicate that at T1, higher levels of agreement significantly correlate with students' being more likely to hold a number of mental models, with strongest correlation being with estimates for holding an appropriate model of iteration. However, as with prior programming experience, the influence of students' initially considering themselves to be self-taught programmers appeared to reduce by T2, with only estimates for Iteration continuing to show a significant effect.

The results presented in Table 5.7 indicate that there was no significant relationship between whether students had previously studied a mathematics-based subject after leaving school, and the probability of students holding appropriate mental models for each concept at both T1 and T2. From the results presented thus far, it can be concluded that when considered as independent factors, studying mathematics-based subjects prior to starting their degree does not significantly impact the probabilities of students having appropriate mental models of concepts examined within the Programming Checkup.

On the other hand, two factors that have been identified as significantly supporting students with their mental models at the start of their course were found to be having prior programming experience and considering themselves to be a "self-taught programmer". However, the influence of these two factors appeared to be less significant as time progressed, given that all students were gaining experience as part of their course. The results also suggest that previously studying computer science did exert a degree of influence over students' mental models.

Table 5.6

Spearman's Rank Correlation Test Between Students' Agreement in Considering Themselves "Self-Taught Programmers" at the Start of Their Course and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2

Mental Model	T1		T2	
	r_s	p	r_s	p
AND	.10	.273	-.04	.689
Conditional Statements	.34	<.001	.15	.100
IF	.32	<.001	.14	.144
Iteration	.43	<.001	.34	<.001
NOT	.14	.126	.18	.048
Output	.36	<.001	.24	.008
OR	.21	.021	.14	.132
Parallelism	.42	<.001	.19	.038
Variable Assignment	.38	<.001	.23	.013
Variable Naming	.21	.022	.05	.622

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*Self-taught agreements range from Strongly Disagree (1) to Strongly Agree (7)

*$n = 119$

Table 5.7

Mann Whitney U Tests Between Previously Studying a Mathematics-Based Subject (Yes/No) and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2

Mental Model	T1				T2			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
AND	1680.00	-0.41	.680	0.04	1722.00	-0.16	.871	0.01
Conditional Statements	1406.00	-1.84	.067	0.17	1682.00	-0.36	.720	0.03
IF	1359.50	-2.10	.036	0.19	1577.70	-0.94	.349	0.09
Iteration	1416.00	-1.91	.057	0.18	1544.00	-1.12	.262	0.10
NOT	1704.50	-0.24	.809	0.02	1519.50	-1.25	.210	0.11
Output	1494.00	-1.37	.172	0.13	1537.50	-1.14	.253	0.10
OR	1657.00	-0.50	.621	0.05	1619.00	-0.70	.202	0.06
Parallelism	1582.00	0.62	.370	0.06	1512.50	-1.27	.483	0.12
Variable Assignment	1494.00	-1.40	.162	0.13	1641.50	-0.62	.533	0.06
Variable Naming	1678.50	-0.40	.693	0.04	1557.50	-1.13	.257	0.10

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 119

5.2.3 Analysis of Students' Levels of Confidence

Students' confidence levels have previously been shown to be a useful predictor of their performance within an introductory programming module (Bergin & Reilly, 2005b; Ramalingam & Wiedenbeck, 1998; Wilson & Shrock, 2001) and as such, provide additional insight into whether a student is likely to struggle with learning to program. Table 5.8 presents the results of a series of Wilcoxon Signed Rank tests conducted on the different confidence related factors examined within the Programming Checkup, thus allowing for any significant changes between T1 and T2 to be identified.

Table 5.8

Wilcoxon Signed Rank Comparison of Confidence Factors at T1 and T2

Factor	*z*	*p*	*r*
Estimation of how difficult learning to program is	-0.12	.905	0.01
Estimation of how difficult they find mathematics	-1.61	.107	0.15
Estimation of how difficult their degree is	-1.04	.296	0.10
How much they fear learning to program	-1.92	.054	0.18
Self-Efficacy Factor 1 (Independence and Persistence)	-5.95	<.001	0.55
Self-Efficacy Factor 3 (Self-Regulation)	-3.23	.001	0.30
Self-Efficacy Factor 4 (Simple Programming Tasks)	-8.51	<.001	0.78
Confidence – Variable Assignment	-5.73	<.001	0.53
Confidence – Conditional Statements	-3.08	.002	0.28
Confidence – Iteration	-5.84	<.001	0.54
Confidence – All Questions	-5.47	<.001	0.50
Mental Effort – Variable Assignment	-2.48	.013	0.23
Mental Effort – Conditional Statements	-1.19	.233	0.11
Mental Effort – Iteration	-3.44	<.001	0.32

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .004$ (i.e., .05 divided by 14) were deemed to be reliable for purposes of interpretation.

*n = 119

Students' difficulty estimations relating to their degree, to mathematics and to learning to program, did not change significantly between T1 and T2, nor did how much they fear learning to program. However, each of Ramalingam and Wiedenbeck's (1998) self-efficacy factors that were examined within the Programming Checkup showed a significant change between T1 and T2. Students' average levels of confidence that they had answered the questions correctly for each of the question categories, and for all questions, also showed a significant change between T1 and T2. Students' mental effort estimates did not change significantly for Variable Assignment and Conditional Statement questions but did for Iteration.

Tables 5.9 through to 5.12 explore whether students' backgrounds significantly influenced their confidence levels at T1 and T2. Having prior programming experience significantly

benefitted students' confidence levels at T1, as shown in Table 5.9, with responses to self-efficacy Factors 1 and 4 from students with prior programming experience being significantly higher than those without. Students with prior programming experience recorded an average rating of 4.65 out of 7 (*SD* = 1.33) for Factor 1 (Independence and Persistence), and an average rating of 4.96 out of 7 (*SD* = 1.41) for Factor 4 (Simple Programming Tasks). Subsequently, students without prior programming experience demonstrated lower self-efficacy levels with an average rating of 3.32 out of 7 (*SD* = 1.58), being recorded for Factor 1 and average rating of 2.78 out of 7 (*SD* = 1.57) for Factor 4. There was also evidence of a substantial difference in responses to Factor 3 (Self-Regulation), which is nearing the Bonferroni corrected threshold (with experience, *M* = 4.25 *SD* = 1.25, without experience, *M* = 3.51 *SD* = 1.32).

Furthermore, there was also a significant difference in students' confidence in their answers being correct at T1 when comparing those with and without prior programming experience, as students with prior experiences recorded higher average levels of confidence for questions focusing on Variable Assignment (with experience, *M* = 73.54, *SD* = 26.70, without experience, *M* = 46.88, *SD* = 30.59), Conditional Statements (with experience, *M* = 82.69, *SD* = 19.97, without experience, *M* = 67.46, *SD* = 25.14), Iteration (with experience, *M* = 65.71, *SD* = 29.33, without experience, *M* = 35.70, *SD* = 27.70) and for all questions combined (with experience, *M* = 75.93, *SD* = 22.96, without experience, *M* = 53.63, *SD* = 23.66).

Like self-efficacy, students with prior programming experience had more confidence in their answers than those without. However, the influence of having prior experience showed evidence of decreasing by T2 in a similar way to how it affected the likelihood of students having appropriate mental models, given that students were progressing through their course and building their confidence levels. This is reflected in the reduction in the effect sizes between T1 and T2 for responses to both the self-efficacy scale and in students' confidence in their answers, as can be seen in Table 5.9. Furthermore, the differences at T2 between students with and without prior experience are no longer significant at the corrected significance threshold for self-efficacy Factor 1 (with experience, *M* = 5.22, *SD* = 1.07, without experience, *M* = 4.73, *SD* = 1.17) and Factor 3 (with experience, *M* = 4.19, *SD* = 1.30, without experience, *M* = 4.59, *SD* = 1.29) and their confidence in their answers for Variable Assignment (with experience, *M* = 85.83, *SD* = 24.61, without experience, *M* = 69.79, *SD* = 33.94) and Conditional Statements (with experience, *M* = 85.83, *SD* = 24.61,

without experience, *M* = 69.79, *SD* = 33.94). However, self-efficacy Factor 4 (Simple Programming Tasks) did still show a significant difference at T2, with students who had prior programming experience recording an average rating of 5.78 out of 7 (*SD* = 1.05), whereas those without recorded an average rating of 5.04 out of 7 (*SD* = 1.28). Despite the relatively large gap when compared to the other two self-efficacy factors, there was an increase in the efficacy levels of students who did not have prior programming experience across all factors related to self-efficacy, with the largest being in Factor 4.

Table 5.9

Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Confidence Factors at T1 and T2

Confidence Factor	**T1**				**T2**			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
Estimation of how difficult learning to program is	1176.50	-2.64	.008	0.24	1056.50	-3.30	<.001	0.30
Estimation of how difficult they find mathematics	1561.50	-0.50	.620	0.05	1620.00	-0.17	.867	0.02
Estimation of how difficult their degree is	1480.50	-0.96	.336	0.09	1394.50	-1.46	.145	0.13
How much they fear learning to program	1222.50	-2.38	.017	0.22	917.00	-3.77	<.001	0.35
Self-Efficacy Factor 1 (Independence and Persistence)	867.50	-4.31	<.001	0.40	1193.00	-2.17	.030	0.20
Self-Efficacy Factor 3 (Self-Regulation)	1129.50	-2.87	.004	0.26	1358.00	-1.24	.217	0.11
Self-Efficacy Factor 4 (Simple Programming Tasks)	531.50	-6.16	<.001	0.56	1001.00	-3.23	.001	0.30
Confidence – Variable Assignment	826.00	-4.54	<.001	0.42	1159.50	-2.77	.006	0.25
Confidence – Conditional Statements	982.00	-3.68	<.001	0.34	1173.00	2.66	.008	0.24
Confidence – Iteration	756.50	-4.92	<.001	0.45	1067.50	-3.22	.001	0.30
Confidence – All questions	775.00	-4.82	<.001	0.44	1050.00	-3.31	<.001	0.30
Mental Effort – Variable Assignment	1287.50	-2.01	.044	0.18	1066.00	-1.17	.243	0.11
Mental Effort – Conditional Statements	1292.50	-1.99	.047	0.18	1143.50	-0.63	.528	0.06
Mental Effort – Iteration	1273.00	-2.10	.036	0.19	1101.00	-0.92	.356	0.08

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) were deemed to be reliable for purposes of interpretation.

*n = 119

Significant differences were also identified at T2 in students' average confidence in their answers for Iteration focused questions (with experience *M* = 76.40, *SD* = 28.89; without experience *M* = 57.46, *SD* = 34.37) and their average confidence across all questions (with experience *M* = 83.62, *SD* = 24.92; without experience *M* = 67.39 *SD* = 32.62) surpassing the Bonferroni corrected threshold. However, whilst no longer surpassing the corrected threshold, the difference in students' confidence in their answers for questions focusing on Variable Assignment (with experience, *M* = 85.84, *SD* = 24.92; without experience, *M* = 69.80 *SD* = 33.94) and Conditional Statements (with experience *M* = 83.96, *SD* = 24.92; without experience *M* = 71.80, *SD* = 33.22) is still at a near-significant level, indicating that there still is a sizeable gap between students with and without prior programming experience.

The results thus far demonstrate that having prior programming experience does give students an additional boost in confidence. However, the difference between those with and without prior experience begins to reduce over time as students progress through the module and gain confidence with the new concepts. This, therefore, appears to support Wiedenbeck et al.'s (2004) claims that having prior programming experience will eventually lose its predictive value, owing to students' more recent experiences with programming being the more important influence on their confidence levels. Nevertheless, the difference in confidence with answering questions on iteration at T2, as well as the differences in mental model estimates (see Table 5.2) suggests that having prior experience is still benefiting students at T2 on a concept that appears to be troublesome for them to grasp.

The insight that having prior programming experience gives students also appears somewhat to reduce their anxiety surrounding learning to program , with both estimations of how difficult learning to program is at T1 (with experience, *M* = 5.64, *SD* = 1.90; without experience, *M* = 6.59, *SD* = 1.99) and how much students fear learning to program (with experience, *M* = 2.96, *SD* = 2.28; without experience, *M* = 4.20, *SD* = 2.71) demonstrating substantial differences between those with and without prior programming experience, which is nearing the Bonferroni corrected threshold. There does not, however, appear to be any significant difference in how difficult students believe their course to be as a whole (with experience, *M* = 6.77, *SD* = 1.74; without experience, *M* = 7.04, *SD* = 1.60), suggesting there may be common causes of anxiety towards their degrees aside from just concerns with programming.

A notable increase in the influence of having prior programming experience was, however, observed between T1 and T2 amongst students' estimations of how difficult learning to program is (with experience, $M = 5.49$, $SD = 2.26$; without experience, $M = 6.91$, $SD = 1.80$) and how much they fear learning to program (with experience, $M = 2.48$, $SD = 2.10$; without experience, $M = 4.11$ $SD = 2.25$), as indicated by the increase in effect size for both factors. Subsequently, the differences in the responses to both factors now surpass the Bonferroni adjusted threshold at T2. This potentially indicates a slight widening of the gap between those with and without prior programming experience in terms of anxiety surrounding learning to program. Indeed, students without prior programming experience reported a slight increase, on average, in how difficult they believe learning to program to be, suggesting that for some struggling students, learning to program continues to get harder as the module progresses.

Table 5.10

Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Confidence Factors at T1 and T2

Confidence Factor	T1				T2			
	U	*z*	*p*	*r*	*U*	*z*	*p*	*r*
Estimation of how difficult learning to program is	1227.00	-1.44	.151	0.13	1186.50	-1.67	.095	0.15
Estimation of how difficult they find mathematics	1434.50	-0.21	.833	0.02	1435.00	-0.21	.836	0.02
Estimation of how difficult their degree is	1400.00	-0.42	.674	0.04	1460.00	-0.06	.952	0.01
How much they fear learning to program	1229.00	-1.42	.155	0.13	1084.50	-2.27	.023	0.21
Self-Efficacy Factor 1 (Independence and Persistence)	1164.00	-1.79	.074	0.16	1375.00	-0.07	.947	0.01
Self-Efficacy Factor 3 (Self-Regulation)	1313.50	-0.92	.360	0.08	1264.50	-0.74	.461	0.07
Self-Efficacy Factor 4 (Simple Programming Tasks)	837.00	-3.69	<.001	0.34	1275.00	-0.67	.501	0.06
Confidence – Variable Assignment	1001.50	-2.74	.006	0.25	907.00	-3.37	<.001	0.31
Confidence – Conditional Statements	1017.50	-2.64	.008	0.24	1003.50	-2.74	.006	0.25
Confidence – Iteration	883.00	-3.42	<.001	0.31	911.00	-3.27	.001	0.30
Confidence – All questions	922.50	-3.19	.001	0.29	884.00	-3.42	<.001	0.31
Mental Effort – Variable Assignment	1390.50	-0.47	.640	0.04	973.00	-0.58	.564	0.05
Mental Effort – Conditional Statements	1437.00	-0.19	.846	0.02	925.50	-0.93	.352	0.09
Mental Effort – Iteration	1424.50	-0.27	.788	0.02	1042.50	-0.06	.955	0.01

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) are deemed to be reliable for purposes of interpretation.

*n = 119

Previously studying computer science also appears to aid students' confidence at T1, which is evident in students who had previously studied computer science having significantly higher average levels of confidence in their answers for questions relating to Iteration (studied computer science, $M = 61.17$, $SD = 31.31$; did not study computer science, $M = 38.88$, $SD = 28.71$) and for all questions combined (studied computer science $M = 71.78$, $SD = 25.95$; did not study computer science, $M = 57.85$, $SD = 21.80$), as shown in Table 5.10. Substantial differences were also observed in students' confidence in their answers for questions relating to Variable Assignment (studied computer science, $M = 68.07$, $SD = 31.09$; did not study computer science, $M = 51.15$, $SD = 28.15$), Conditional Statements (studied computer science, $M = 79.88$, $SD = 23.03$; did not study computer science, $M = 70.28$, $SD = 22.24$) and although these differences did not reach the Bonferroni corrected threshold, they still provide evidence of increased confidence amongst students with prior experience of studying computer science. Additionally, students' levels of self-efficacy for Factor 4 (studied computer science, $M = 4.56$, $SD = 1.65$; did not study computer science, $M = 3.16$, $SD = 1.81$) were significantly higher amongst students who had previously studied computer science. Although, this is not reflected in the self-efficacy levels for Factor 3 (studied computer science, $M = 4.07$, $SD = 1.25$, did not study computer science, $M = 3.76$, $SD = 1.49$) or for Factor 1 (studied computer science, $M = 4.35$, $SD = 1.44$; did not study computer science, $M = 3.70$, $SD = 1.76$).

By T2, previously studying computer science demonstrated a negligible level of influence on students' responses for all three self-efficacy factors: Factor 1 (studied computer science, $M = 5.04$, $SD = 1.16$; did not study computer science, $M = 5.04$, $SD = 1.05$), Factor 3 (studied computer science, $M = 4.41$, $SD = 1.29$; did not study computer science, $M = 4.55$, $SD = 1.36$), Factor 4 (studied computer science, $M = 5.54$, $SD = 1.20$; did not study computer science, $M = 5.44$, $SD = 1.19$), clearly indicating that the benefits of previously studying computer science are short lived in terms of levels of self-efficacy relating to completing simple programming tasks. However, it is evident that students who had previously studied computer science remain more confident in their answers at T2, with significant differences being evident in students' average confidence in their answers for questions focusing on Variable Assignment (studied computer science, $M = 85.44$, $SD = 24.09$; did not study computer science, $M = 66.62$, $SD = 36.20$), Iteration (studied computer science, $M = 74.98$, $SD = 29.73$; did not study computer science, $M = 55.98$, $SD = 34.51$) and for all questions combined (studied computer science, $M = 82.90$, $SD = 24.53$; did not study computer science,

M = 64.94, SD = 34.77). Additionally, the average confidence levels for questions on Conditional Statement (studied computer science, M = 83.90, SD = 25.83; did not study computer science, M = 68.81, SD = 36.03) is approaching the Bonferroni Corrected threshold.

No significant differences were identified amongst estimates of students' anxiety levels surrounding their degree and learning to program at T1, although students' levels of fear of learning to program do surpass the standard significance threshold of $p < .05$ at T2, but not the Bonferroni corrected threshold. This result, however, could likely be accounted for by the experiences students have gained whilst studying the module.

Students' level of agreement with the notion of considering themselves "self-taught programmers" prior to starting their degrees also appeared to influence their confidence levels, as shown in Table 5.11. Significant correlations were identified at T1 between stronger levels of agreement in considering themselves to be self-taught and higher levels of self-efficacy for all three factors examined within the Programming Checkup, with the strongest correlation being with Factor 4. Students' confidence in their answers was also significantly correlated with their levels of agreement in considering themselves to be self-taught, although correlations with their confidence in answering questions focusing on conditional statements can only be viewed as approaching the Bonferroni adjusted significance threshold. Additionally, students' estimations of how difficult learning to program is, and their levels of fear surrounding learning to program were significantly correlated with how strongly they considered themselves to be self-taught programmers at T1.

The influence of students considering themselves to be self-taught prior to starting their courses appeared to reduce somewhat by T2, with weaker correlations being observed between students' level of agreement of considering themselves to be a self-taught and both their self-efficacy levels and their confidence in their answers. However, interestingly, an increase was observed in the strength of the correlations with how difficult they believe learning to program to be, as well as with how much they fear learning to program, although not as strongly as the former.

It should also be noted that students' level of mental effort when answering questions on Iteration has a significant, albeit weak, negative correlation with their levels of agreement with being "self-taught programmers" at both T1 and T2. However, the poor strength of the correlation, combined with the fact that neither students' mental effort ratings for questions relating to variable assignment or conditional statements show significant correlations with how strongly they consider themselves to be self-taught, makes it difficult to draw any statistically reliable conclusions relating to the broader effects of mental effort.

Additionally, previously studying a mathematics-based subject was found not to aid students' confidence levels as much as other background factors, as shown in Table 5.12, with none of the measured confidence factors reaching the Bonferroni corrected threshold at T1 or T2. Several factors, however, did surpass the standard significance threshold of $p < .05$, although the effect sizes suggest a weak level of influence at best.

Students' initial confidence in their answers and their self-efficacy levels do appear to be positively influenced by having previous experience of programming and to a lesser extent, studying computer science, with students who more strongly consider themselves to be self-taught programmers also showing greater confidence and self-efficacy, although as all students gain experience as they progress through the course the effect of having this prior experience on confidence and self-efficacy levels has been seen to decrease. A study conducted by Wiedenbeck et al.'s (2004) revealed that students with stronger mental models also had higher levels of self-efficacy due to an increased level of program comprehension. However, their study analysed students' mental models of how a given program works rather than at an individual concept level, as is done in this investigation.

Table 5.11

Spearman's Rank Correlation Test Between Students' Agreement in Considering Themselves "Self-Taught Programmers" at the Start of Their Course and Confidence Factors at T1 and T2

Confidence Factor	T1		T2	
	r_s	p	r_s	p
Estimation of how difficult learning to program is	-.32	<.001	-.43	<.001
Estimation of how difficult they find mathematics	-.20	.313	-.22	.018
Estimation of how difficult their degree is	-.06	.538	-.18	.057
How much they fear learning to program	-.32	<.001	-.34	<.001
Self-Efficacy Factor 1 (Independence and Persistence)	.49	<.001	.33	<.001
Self-Efficacy Factor 3 (Self-Regulation)	.42	<.001	.23	.014
Self-Efficacy Factor 4 (Simple Programming Tasks)	.61	<.001	.38	<.001
Confidence – Variable Assignment	.31	<.001	.27	.004
Confidence – Conditional Statements	.26	.004	.18	.045
Confidence – Iteration	.48	<.001	.35	<.001
Confidence – All Questions	.37	<.001	.32	<.001
Mental Effort – Variable Assignment	-.05	.569	-.02	.863
Mental Effort – Conditional Statements	-.10	.296	-.04	.691
Mental Effort – Iteration	-.27	.003	-.21	.030

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) were deemed to be reliable for purposes of interpretation.

*Self-taught agreements range from Strongly Disagree (1) to Strongly Agree (7)

*n = 119

Table 5.12

Mann Whitney U Tests Between Previously Studying a Mathematics-Based Subject (Yes/No) And Confidence Factors at T1 and T2

Confidence Factor	T1				T2			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
Estimation of how difficult learning to program is	1284.00	-2.52	.012	0.23	1366.50	-2.07	.039	0.19
Estimation of how difficult they find mathematics	1305.00	-2.42	.016	0.22	1464.00	-1.54	.123	0.14
Estimation of how difficult their degree is	1582.00	-0.92	.357	0.08	1668.50	-0.45	.656	0.04
How much they fear learning to program	1259.00	-2.65	.008	0.24	1359.50	-2.10	.036	0.19
Self-Efficacy Factor 1 (Independence and Persistence)	1340.50	-2.19	.029	0.20	1547.00	-0.82	.414	0.08
Self-Efficacy Factor 3 (Self-Regulation)	1307.00	-2.37	.018	0.22	1438.50	-1.41	.158	0.13
Self-Efficacy Factor 4 (Simple Programming Tasks)	1471.00	-1.49	.137	0.14	1411.00	-1.56	.118	0.14
Confidence – Variable Assignment	1557.00	-1.03	.304	0.09	1596.50	-0.84	.403	0.08
Confidence – Conditional Statements	1617.00	-0.71	.480	0.07	1622.50	-0.68	.496	0.06
Confidence – Iteration	1527.50	-1.18	.236	0.11	1531.50	-1.17	.244	0.11
Confidence – All questions	1554.50	-1.04	.298	0.10	1529.00	-1.18	.239	0.11
Mental Effort – Variable Assignment	1572.00	-0.95	.340	0.09	1121.50	-1.30	.193	0.12
Mental Effort – Conditional Statements	1576.50	-0.93	.352	0.09	1237.50	-0.52	.600	0.05
Mental Effort – Iteration	1655.00	-0.51	.611	0.05	1293.50	-0.15	.881	0.01

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) were deemed to be reliable for purposes of interpretation.

*n = 119

The three self-efficacy factors examined within the Programming Checkup all yielded similar results at T1, with responses to Factor 1 (having the highest level on average (M = 4.16, SD = 1.56), closely followed by Factor 4 (M = 4.15, SD = 1.81) and Factor 3 (M = 3.98, SD = 1.32). The average self-efficacy levels for all three factors increased at T2, with Factor 4 now having the highest average level (M = 5.51, SD = 1.90), followed by Factor 1 (M = 5.04, SD = 1.28) and with Factor 3 again having the lowest level on average (M = 4.44, SD = 1.31). The significance of the change in self-efficacy levels is confirmed by the Wilcoxon Signed Rank tests presented in Table 5.8, although it is not surprising to see such an increase given the experiences that students gain whilst studying on an introductory programming module – particularly in completing simple programming tasks that are analysed as part of Factor 4.

Table 5.13 examines the correlations between each of the three self-efficacy factors and the mental model estimates established using Bayesian Knowledge Tracing at T1. The results show that for self-efficacy Factor 1, the strength of the correlation ranges from $r_s = .10$, $p = .263$ for students' models of NOT, to $r_s = .42$, $p < .001$, for their models of Iteration. Although a number of the correlations surpass the adjusted significance threshold, the strength of the correlations can be considered moderate at best, with the majority of models exhibiting weak correlations with Factor 1. Additionally, no models were found to have a correlation with self-efficacy Factor 3, which surpassed the adjusted significance threshold at T1. All but two models were deemed to have correlations with self-efficacy Factor 4, which surpassed the adjusted significance threshold at T1. The standard significance threshold of .05 was, however, surpassed by the remaining two models. The correlations between the models and Factor 4 were much more substantial, with the strongest correlation being with students' models of Iteration, $r_s = .71$, $p < .001$. The two models with the weakest correlations were AND, $r_s = .21$, $p = .025$, and NOT, $r_s = .25$, $p = .007$, although, these models along with OR, $r_s = .37$, $p = < .001$, can in fact be substituted for the Conditional Statement model, which encompasses all questions related to Boolean Logic and selection statements, and does in fact have a moderately strong correlation with self-efficacy Factor 4, $r_s = .56$, p < .001. By T2, however, the strength of the correlations between mental model estimates and self-efficacy levels was found to be noticeably weaker than the equivalents at T1, suggesting that students' progression with developing appropriate mental models and improving their levels of self-efficacy does not necessarily progress at the same rate.

Table 5.13

Spearman's Rank Correlation Tests Between Self-Efficacy Factors and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2

Mental Model	Self-Efficacy Factor 1 (Independence and Persistence)				Self-Efficacy Factor 3 (Self-Regulation)				Self-Efficacy Factor 4 (Simple Programming Tasks)			
	T1		T2		T1		T2		T1		T2	
	r_s	*p*	r_s	*p*	r_s	*p*	r_s	*p*	r_s	*p*	r_s	*p*
AND	.12	.188	.09	.342	-.05	.553	.05	.584	.21	.025	.06	.540
Conditional Statements	.35	<.001	.27	.004	.23	.013	.14	.125	.56	<.001	.28	.002
IF	.32	<.001	.18	.047	.25	.006	.10	.298	.48	<.001	.18	.060
Iteration	.42	<.001	.35	<.001	.23	.013	.27	.004	.71	<.001	.46	<.001
NOT	.10	.263	.29	.002	.02	.847	.10	.272	.25	.007	.26	.005
Output	.37	<.001	.20	.031	.20	.027	.11	.240	.62	<.001	.25	.008
OR	.14	.118	.23	.015	.03	.719	.04	.649	.37	<.001	.24	.011
Parallelism	.28	.002	.23	.014	.19	.040	.18	.047	.55	<.001	.32	<.001
Variable Assignment	.35	<.001	.04	.641	.15	.112	.05	.607	.60	<.001	.05	.583
Variable Naming	.07	0.443	.14	.144	.01	.889	.13	.154	.31	<.001	.17	.065

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20 – the number of tests for each Self-Efficacy factor) were deemed to be reliable for purposes of interpretation.

*n = 119

Students' average confidence in their answers to all programming questions demonstrates significant relationships with all mental model estimates at T1, with the majority of correlations being of moderate strength, as shown in Table 5.14. Again, the strength of these correlations decreases at T2, except for AND, NOT and Variable Naming, which experience a marginal increase in strength.

Table 5.14

Spearman's Rank Correlation Tests Between Average Confidence in Answers for All Programming Questions and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2

Mental Model	**T1**		**T2**	
	r_s	p	r_s	p
AND	.32	<.001	.33	<.001
Conditional Statements	.63	<.001	.46	<.001
IF	.54	<.001	.40	<.001
Iteration	.65	<.001	.56	<.001
NOT	.35	<.001	.37	<.001
Output	.65	<.001	.48	<.001
OR	.43	<.001	.39	<.001
Parallelism	.52	<.001	.48	<.001
Variable Assignment	.65	<.001	.35	<.001
Variable Naming	.39	<.001	.42	<.001

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 119

The average confidence in students' answers being correct across all questions within the Programming Diagnostic section of the Programming Checkup has been used thus far, as multiple models are being assessed in each question. However, the questions can be divided based on the main concept being examined within the question (Variable Assignment, Conditional Statements or Iteration) to allow for further analysis of students' confidence in their answers for these questions.

Interestingly, the correlations between students' confidence specifically relating to answering questions on Variable Assignment and their estimates of having an appropriate mental model (T1 r_s = .63, p < .001; T2 r_s = .32, p < .001) are slightly weaker at both T1 and T2 when compared to their average confidence for all questions. Similarly, students' confidence when answering questions on Conditional Statements exhibits a slightly weaker correlation at T1 and an equal strength correlation at T2, when compared with their equivalents across all questions (T1 r_s = .59, p < .001; T2 r_s = .46 p < .001).

However, when answering questions on Iteration, the correlations between students' confidence in their answers and their mental model estimates are stronger at both T1 (r_s = .70, p < .001) and T2 (r_s = .63, p < .001) when compared to their average confidence for all questions. This could be due to the fact that students may find Iteration a much harder topic, as discussed previously, which is reflected in a lower average confidence level at both T1 and T2 (T1, *M* = 54.61, *SD* = 32.11; T2, *M* = 69.40, *SD* = 32.26) when compared to when students are answering questions on Variable Assignment (T1, *M* = 63.68, *SD* = 30.91; T2, *M* = 79.90, *SD* = 29.33), Conditional Statements (T1, *M* = 77.06, *SD* = 23.13; T2, *M* = 79.47, *SD* = 29.85) and for all questions combined (T1, *M* = 67.68, *SD* = 25.52; T2, *M* = 77.62, *SD* = 28.97). Furthermore, this may also be an indication that the other models being examined in the questions (i.e., Variable Naming, Parallelism or Output), may be impacting students' confidence in their answers more for questions focusing on Variable Assignment and Conditional Statements, than questions on Iteration, where the concept of Iteration itself is the main factor affecting students' confidence in their answers.

Given the results presented above, it is possible to conclude that students who are considered to be more likely to hold appropriate mental models of concepts such as Conditional Statements, Iteration, Output and Parallelism, are likely to have greater confidence in their answers and also have higher levels of self-efficacy in relation to completing simple

programming tasks at T1, although the strength of these relationships does decrease by T2. Model estimates for Iteration exhibit stronger relationships with students' confidence and self-efficacy, suggesting that developing an understanding of this key concept significantly influences students' overall confidence and self-efficacy levels.

Upon completion of the questions within the Programming Diagnostic portion of the Programming Checkup, students were required to record how much mental effort they felt was required when answering questions. As students provided ratings of mental effort for each of the main question categories (Variable Assignment, Conditional Statements and Iteration), this allows for an examination of the relationship between students' mental effort ratings and their confidence in their answers for each of the categories. As such, students' mental effort ratings at T1 were found to be significantly negatively correlated (significance threshold adjusted to $p < .008$, 0.05/6, due to the Bonferroni correction), with their confidence in their answers for questions focusing on the corresponding concepts, with Variable Assignment exhibiting the strongest correlation ($r_s = -.45$, $p < .001$), followed by Iteration ($r_s = -.42$, $p < .001$) and then Conditional Statements ($r_s = -.41$, $p < .001$). This, therefore, provides some evidence to suggest that students' who are less confident in their answers require more mental effort to answer questions. The correlations remain significant at T2 (Variable Assignment, $r_s = -.33$, $p < .001$; Conditional Statements, $r_s = -.27$, $p = .006$; Iteration, $r_s = -.30$, $p = .002$). However, their strength has now decreased, which makes it difficult to establish any firm conclusions about the relationship between students' confidence in their answers and the mental effort they exert.

A similar trend is also evident when comparing students' mental effort ratings to the estimates of having an appropriate mental model of the main concepts being examined within the question. Significant correlations (significance threshold adjusted to $p < .008$, 0.05/6 due to the Bonferroni correction) were identified between the corresponding mental effort and model estimates for Variable Assignment ($r_s = -.36$, $p <.001$), Conditional Statements ($r_s = -.27$, $p = .003$) and Iteration ($r_s = -.27$, $p = .003$), albeit being much weaker than the equivalent correlations between the mental model estimates and students' confidence in their answers, as discussed previously. There were no significant correlations at T2 (Variable Assignment, $r_s = -.15$ $p = .127$; Conditional Statements, $r_s = -.08$, $p = .446$; Iteration, $r_s = -0.12$, $p = .234$).

Given that students were only asked to provide ratings of their mental effort at the end of the Programming Checkup, it may be appropriate for a future study to include a more comprehensive measurement of students' levels of mental effort, as has been done with their levels of confidence in their answers, in order to allow for a more robust investigation into how mental effort levels relate to students' prior experiences, their levels of confidence and their likelihood of holding appropriate mental models of the concepts being examined.

Curzon and Rix (1998), previously reported that one of the main reasons students want to learn to program at the beginning of their course is so that they can pursue a career as a professional programmer. However, the number of students still wanting to pursue this career path decreases as the course progresses, which is likely due to the difficulties students encounter when learning to program.

At T1, 64 of the 119 students who took part in both rounds of data collection stated they wished to pursue a career in software engineering, whilst 48 stated they were undecided and 7 stated they did not wish to pursue this career path. However, by T2 the number of students stating that they did wish to work in software engineering had decreased slightly to 61, with the number of students who were undecided also decreasing to 41 and those not wanting to work in software engineering increasing to 17. A Wilcoxon Signed Rank test confirmed the significance of the change in students' intentions to work in software engineering between T1 and T2, $z = -2.047$, $p = .041$, $r = 0.19$, which provides a degree of support for Curzon and Rix's (1998) claims. As the data collection for T2 was carried out at the end of the first semester it would be interesting to see if as large a drop as reported by Curzon and Rix (1998) occurs by the time students had completed their first full year.

Students' intentions for their intended career path upon graduation speaks to their motivation and determination for learning to program, as Bergin and Reilly (2005a) go on to state how students who were intrinsically motivated perform better than those who were extrinsically motivatied. Students who do wish to pursue a career in software engineering at T1, as opposed to those who are either undecided or do not want to work in software engineering, were significantly more likely, as confirmed by the Mann Whitney U tests presented in Table 5.15, to have appropriate models at T1 for Conditional Statements (Yes $M = 0.60$, $SD = 0.44$; No/Undecided $M = 0.34$, $SD = 0.45$), If statements (Yes $M = 0.93$, $SD = 0.19$; No/Undecided $M = 0.81$, $SD = 0.32$), Iteration (Yes $M = 0.32$, $SD = 0.45$; No/Undecided $M = 0.06$, $SD =$

0.23), Output (Yes *M* = 0.87 *SD* = 0.32; No/Undecided *M* = 0.78, *SD* = 0.40), Variable Assignemnt (Yes *M* = 0.62, *SD* = 0.46; No/Undecided *M* = 0.33, *SD* = 0.43) and also Parallelism (Yes *M* = 0.59, *SD* = 0.40; No/Undecided *M* = 0.40, *SD* = 0.36) which is approaching the corrected significance threshold. However, by T2, there are signs of the gap reducing between those who do want to work in software eingeering and those who do not, or are undecided. This is reflected in only estimates for Iteration (Yes, *M* = 0.47, *SD* = 0.48; No/Undecided, *M* = 0.21, *SD* = 0.38) and Conditional Statements (Yes, *M* = 0.72, *SD* = 0.40; No/Undecided, *M* = 0.48, *SD* = 0.48) continuing to demonstrate significant differences. Interestingly, the differences in students' estimates of having an appropriate model for NOT has become significant at T2 (Yes, *M* = 0.83, *SD* = 0.31; No/Undecided, *M* = 0.67 *SD* = 0.35), although it is difficult to suggest any conclusive reasons for this increase.

Table 5.15

Mann Whitney U Tests Between Students' Intentions to Pursue a Career in Software Engineering (Yes or Undecided/No) and Mental Model Estimates Established Using Bayesian Knowledge Tracing at T1 and T2

Mental Model	T1				T2			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
AND	1419.50	-2.03	.042	0.19	1430.50	-2.02	.044	0.19
Conditional Statements	1207.50	-2.95	.003	0.27	1190.00	-3.08	.002	0.28
IF	1167.50	-3.19	.001	0.29	1458.50	-1.69	.092	0.15
Iteration	1084.00	-3.86	<.001	0.35	1141.50	-3.41	<.001	0.31
NOT	1517.50	-1.32	.188	0.12	1047.50	-3.92	<.001	0.36
Output	1150.00	-3.26	.001	0.30	1335.50	-2.33	.020	0.21
OR	1556.50	-1.09	.275	0.10	1303.50	-2.49	.012	0.23
Parallelism	1243.00	-2.77	.006	0.25	1464.50	-1.63	.102	0.15
Variable Assignment	1088.00	-3.67	<.001	0.34	1564.00	-1.18	.237	0.11
Variable Naming	1351.50	-2.28	.023	0.21	1503.00	-1.57	.117	0.14

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 119

Furthermore, students who do wish to pursue a career in software engineering are generally more confident, as shown in Table 5.16, with significant differences being identifited at T1 in how difficult students believe learning to program to be (Yes, $M = 5.30$ $SD = 2.034$; No/Undecided, $M = 6.73$, $SD = 1.65$), how much they fear learning to program (Yes, $M = 2.52$, $SD = 2.085$; No/Undecided, $M = 4.47$, $SD = 2.57$), their self-efficacy levels for all three factors (Factor 1, Yes, $M = 4.70$, $SD = 1.50$; No/Undecided, $M = 3.53$, $SD = 1.40$; Factor 3, Yes, $M = 4.46$, $SD = 1.13$; No/Undecided, $M = 3.42$, $SD = 1.32$; Factor 4, Yes, $M = 4.68$, $SD = 1.73$; No/Undecided, $M = 3.54$, $SD = 1.71$), and their average confidence in answering questions on Variable Assignment (Yes, $M = 72.07$, $SD = 28.51$; No/Undecided, $M = 53.92$, $SD = 30.96$), Conditional Statements (Yes, $M = 82.57$, $SD = 20.56$; No/Undecided, $M = 70.64$, $SD = 24.45$), Iteration (Yes, $M = 64.37$, $SD = 30.72$; No/Undecided, $M = 43.25$, $SD = 30.13$), and for all questions combined (Yes, $M = 75.09$, $SD = 22.77$; No/Undecided, $M = 59.07$, $SD = 26.03$).

The differences remain significant at T2 between those who do wish to pursue a career in software engineering and those who are unsure or do not wish to for factors includinng how much students fear learning to program, self-efficacy Factors 1 (Yes, $M = 5.46$, $SD = 0.90$; No/Undecided, $M = 4.60$, $SD = 1.18$) and Factor 4 (Yes, $M = 5.85$, $SD = 0.95$; No/Undecided, $M = 5.16$, $SD = 1.32$), and students confidence in their answers (Variable Assignment, Yes, $M = 87.14$, $SD = 25.19$; No/Undecided, $M = 72.29$, $SD = 31.58$; Conditional Statement, Yes, $M = 85.39$, SD 26.95; No/Undecided, $M = 73.234$, $SD = 31.70$; Iteration, Yes, $M = 79.39$, $SD = 27.023$; No/Undecided, $M = 58.88$, $SD = 34.14$; All, Yes, $M = 85.46$, $SD = 24.59$; No/Undecided, $M = 69.37$, $SD = 31.07$).

The results therefore indicate that students who want to pursue a career in software engineering are generally more confident in their programming abilities. There is also evidence to suggest that students who do initially want to work in software engineering are also more likely to be holding appropriate mental models of a variety of conceptions at the commencement of their introductory programming module. However, there appeared to be less of a difference in terms of the liklihood of holding appropriate mental models by T2. This suggests that students' intentions for wanting to work in a software engineering role upon graduation may be influenced, to a greater extent, by how confident they are in their abilties, rather than how likely they are to actually be holding appropriate mental models.

Table 5.16

Mann Whitney U Tests Between Students' Intentions to Pursue a Career in Software Engineering (Yes or Undecided/No) And Confidence Factors at T1 and T2

Confidence Factor	**T1**				**T2**			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
Estimation of how difficult learning to program is	1096.00	-3.59	<.001	0.33	1322.00	-2.40	.016	0.22
Estimation of how difficult they find mathematics	1453.50	-1.66	.096	0.15	1509.50	-1.40	.163	0.13
Estimation of how difficult their degree is	1716.50	-0.24	.811	0.02	1524.00	-1.35	.178	0.12
How much they fear learning to program	986.00	-4.17	<.001	0.38	833.50	-5.02	<.001	0.46
Self-Efficacy Factor 1 (Independence and Persistence)	958.50	-4.27	<.001	0.39	959.00	-4.10	<.001	0.38
Self-Efficacy Factor 3 (Self-Regulation)	982.50	-4.15	<.001	0.38	1186.50	-2.86	.004	0.26
Self-Efficacy Factor 4 (Simple Programming Tasks)	1116.50	-3.43	<.001	0.31	1151.50	-3.05	.002	0.28
Confidence – Variable Assignment	1141.50	-3.30	<.001	0.30	1051.00	-3.92	<.001	0.36
Confidence – Conditional Statements	1163.00	-3.19	.001	0.29	1200.50	-3.04	.002	0.28
Confidence – Iteration	1085.50	-3.60	<.001	0.33	1088.50	-3.63	<.001	0.33
Confidence – All Questions	1111.50	-3.46	<.001	0.32	1016.50	-4.01	<.001	0.37
Mental Effort – Variable Assignment	1489.00	-1.46	.145	0.13	1121.50	-1.36	.175	0.12
Mental Effort – Conditional Statements	1633.00	-0.68	.494	0.06	1089.00	-1.57	.117	0.14
Mental Effort – Iteration	1552.50	-1.12	.264	0.10	1207.00	-0.79	.432	0.07

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) were deemed to be reliable for purposes of interpretation.

*n = 119

5.3 Examination of Relationships with Assessment 1 Results

The analyses conducted so far have focused on the changes in results between T1 and T2. Unfortunately, however, a large number of students decided not to take part at T2, which therefore reduces the amount of available data. Although the data collected at T2 are important for tracking students' progress across the first semester, the purpose of this investigation was the development of a predictive model that can be used when students first start their courses. This means that the relationships between the data collected at T1 and students' assessment results are the most important in terms of contributions towards development of the model.

Section 4.3 presents the feature selection process used to select which features to include in the model. However, this was carried out with 30% of the data being withheld from the analysis in order for it to be used later as the test dataset. Although this is a standard practice in machine learning to allow for the real-world performance of the model to be estimated (Kuhn & Johnson, 2013; Raschka, 2018; Russell & Norvig, 2020) by fully isolating the test set from all stages of model development, it does limit the amount of data available for analysis. Therefore, the following analysis includes all available data at T1 (n = 285) and was carried out after the model development was completed in order to allow for more robust conclusions to be drawn, whilst maintaining the integrity of the model development process as described in Section 4.3.

Although many of the tests below were also carried out during the feature selection process, they now include all available data, which allows for a much deeper analysis to be carried out than was done previously. Furthermore, the Bonferroni correction was not applied during the feature selection process due to the fact that its over-conservative nature would likely lead to a large number of features being excluded. However, the correction will be applied in this section in order to reduce the chance of Type 1 errors when drawing conclusions about the significance of the relationships within the data.

As stated previously, students' results from their first assessment within their Introductory Programming module were chosen as the dependent variable of the predictive model as these results focus on evaluating students' core programming skills, which ties in closely with the Programming Diagnostic questions within the Programming Checkup.

Table 5.17 presents a series of Mann Whitney U tests, which repeats the analysis conducted in Table 4.3, whereby the dichotomous features relating to students' backgrounds are examined as to whether they have a significant influence on students' Assessment 1 results. These analyses include the one-hot encoded responses to "Work in Software Engineering", where students' responses to "Yes" surpassed the Bonferroni corrected significance threshold of $p < .007$. Furthermore a Kruskal Wallis test also confirmed the existence of a significant difference between students' assessment results and the three possible responses – Yes (M = 72.98 SD = 18.26), Undecided (M = 65.73 SD = 21.95) and No (M = 64.12 SD = 18.61); $H(2)$ = 9.270, p = .010, $\eta^2 = 0.032$. Additionally, students' level of agreement with how strongly they considered themselves to be "self-taught programmers" was confirmed also to be significantly correlated, albeit relatively weakly, with their Assessment 1 results, r_s = .285, p = <.001, when tests were conducted on all available data at T1.

Table 5.17

Mann Whitney U Test Between Assessment 1 Results and Dichotomous Background Factors, Conducted on All Available Data at T1

Background Factor	***U***	***z***	***p***	***r***
Prior programming experience	7168.50	-2.26	.024	0.13
Previously Studied computer science	7270.00	-1.68	.094	0.09
Previously Studied mathematics-based subject	8083.50	-2.71	.007	0.16
Intend to work in software engineering - No	1762.50	-1.57	.117	0.09
Intend to work in software engineering – Undecided	8007.00	-2.25	.024	0.13
Intend to work in software engineering - Yes	7950.50	-2.95	.003	0.17
English is student's first language	5108.50	-0.94	.347	0.05

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .007$ (i.e., .05 divided by 7) were deemed to be reliable for purposes of interpretation.

**n = 285*

The Background Features, which were retained following the analysis conducted in Section 4.3 were: whether students have studied a mathematics-based subject after leaving school, whether students intended to pursue a career in software engineering and how strongly students consider themselves to be self-taught programmers. The results of the analysis

conducted on all available data at T1 support the conclusions drawn in Section 4.3 as to which background features should be included in the model. Two features which were not included in the models were whether students have previously studied computer science and whether students have prior programming experience. However, given that the previously discussed literature has suggested that prior experience – particularly prior programming experience – can aid students' performance, it was felt that it would be appropriate to examine these factors further.

Previously, when reviewing the results of students who completed the Programming Checkup at both T1 and T2, having prior programming experience was found to benefit students at T1 in terms of their likelihood of holding appropriate mental models for each of the concepts and their confidence in general. However, by T2 the differences between those with and without programming experience decreases for the majority of the factors examined within the Programming Checkup.

Tables 5.18 and 5.19 repeat the Mann Whitney U tests performed in Tables 5.3 and 5.9 in order to confirm any significant differences in students' mental model estimates or confidence levels at T1 between those who did and did not have prior programming experience. However, these analyses now focus on all available data at T1.

Table 5.18

Mann Whitney U Tests Between Prior Programming Experience (Yes/No) And Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1

Mental Model	***U***	***z***	***p***	***r***
AND	6910.00	-2.91	.004	0.17
Conditional Statements	4980.50	-5.67	<.001	0.34
IF	5732.00	-4.52	<.001	0.27
Iteration	4857.00	-6.33	<.001	0.38
NOT	6510.00	-3.33	<.001	0.20
Output	4132.50	-7.02	<.001	0.42
OR	6211.00	-3.77	<.001	0.22
Parallelism	4675.50	-6.17	<.001	0.37
Variable Assignment	4397.00	-6.71	<.001	0.40
Variable Naming	6421.00	-3.61	<.001	0.21

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .005$ (i.e., .05 divided by 10) were deemed to be reliable for purposes of interpretation.

*n = 285

Table 5.19

Mann Whitney U Tests Between Prior Programming Experience (Yes/No) and Confidence Factors, Conducted on All Available Data at T1

Confidence Factor	***U***	***z***	***p***	***r***
Estimation of how difficult learning to program is	5618.00	-4.73	<.001	0.28
Estimation of how difficult they find mathematics	8521.50	-0.15	.885	0.01
Estimation of how difficult their degree is	8439.50	-0.28	.780	0.02
How much they fear learning to program	6200.00	-3.80	<.001	0.23
Self-Efficacy Factor 1 (Independence and Persistence)	4241.00	-6.83	<.001	0.40
Self-Efficacy Factor 3 (Self-Regulation)	6238.50	-3.71	<.001	0.22
Self-Efficacy Factor 4 (Simple Programming Tasks)	3041.00	-8.70	<.001	0.52
Confidence – Variable Assignment	4664.50	-6.17	<.001	0.37
Confidence – Conditional Statements	5539.50	-4.80	<.001	0.28
Confidence – Iteration	4134.00	-6.99	<.001	0.41
Confidence – All Questions	4429.50	-6.53	<.001	0.39
Mental Effort – Variable Assignment	7422.00	-0.89	.375	0.05
Mental Effort – Conditional Statements	7286.00	-1.11	.266	0.07
Mental Effort – Iteration	6362.50	-2.65	.008	0.16

Note. Given the number of tests being conducted a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .004$ (i.e., .05 divided by 14) were deemed to be reliable for purposes of interpretation.

*n = 285

The Mann Whitney U test in Table 5.17 indicated that having prior programming experience does exert some level of influence on students' Assessment 1 results. However, although it surpasses the standard alpha significance threshold of .05, it fails to surpass the Bonferroni corrected threshold of $p < .006$. Similarly, the Mann Whitney U test performed during the feature selection process in Table 4.3, fails to surpass the standard threshold of .05, which contributed to prior programming experience not being included in the model.

The likelihood of there being a direct link between having prior programming experience and success within Assessment 1 appears tentative at best, with students with prior experience obtaining a slightly higher result, M = 71.45, SD = 19.39, than those without, M = 65.80, SD = 21.00. However, it has been previously stated that there are a number of factors examined

within the Programming Checkup where having prior programming experience has proven to cause a significant difference in results.

Prior programming experience may therefore, potentially be indirectly affecting students' Assessment 1 results by acting as a moderator variable, which according to Baron and Kenny (1986) "affects the direction and/or strength of the relation between an independent or predictor variable and a dependent or criterion variable" (p. 1174). Tables 5.20 and 5.21 present a Moderation Analysis conducted between having prior programming experience and students' mental model estimates and confidence factors, respectively, using the PROCESS Macro (Hayes, 2022). However, only mental model estimates and confidence factor ratings which were deemed to have significant differences between students with and without prior programming experience (as shown in Tables 5.18 and 5.19), were included in the moderation analysis. The moderation analysis tables include the full regression results with students' Assessment 1 results being the dependent variable. However, it is necessary to review the "Ind. x Mod. Interaction" row in order to examine whether prior programming experience does in fact affect students' assessment results through its influence on each of the independent variables.

Prior programming experience was revealed to only have one potentially significant interaction with a mental model (i.e., AND). However, this failed to reach the adjusted significance threshold. Additionally, a Johnson-Neyman analysis revealed that amongst students who did not have prior programming experience, there was a significant relationship between students' estimates of having an appropriate model for AND and their Assessment 1 results, $b = 27.58$, $t = 3.41$, $p = < .001$. However, the relationship was not significant amongst those with prior programming experience, $b = 4.58$, $t = 0.64$, $p = .525$. Given this, and the rest of the evidence presented in Table 5.20, prior programming experience should be generally viewed as not acting as a moderator on students' mental model estimates when attempting to predict their assessment performance.

Table 5.20

Moderation Analysis Between Prior Programming Experience (Yes/No) and Mental Model Estimates When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1

Mental Model	Regression Model	*b*	*SE*	*t*	*p*
	Constant	41.77	7.36	5.68	<.001
AND	Independent Variable	27.58	8.10	3.41	.001
R^2 = .06	Moderator	25.37	10.10	2.51	.013
	Ind. x Mod. Interaction	-23.00	10.84	-2.12	.034
	Constant	63.06	2.46	25.60	<.001
Conditional Statements	Independent Variable	9.34	4.87	1.92	.056
R^2 = .14	Moderator	-2.28	3.3297	-0.68	.495
	Ind. x Mod. Interaction	8.50	5.73	1.48	.138
	Constant	56.39	5.68	9.92	<.001
IF	Independent Variable	11.96	6.71	1.78	.076
R^2 = .05	Moderator	2.65	7.51	0.35	.724
	Ind. x Mod. Interaction	2.11	8.57	0.25	.806
	Constant	65.00	2.11	30.68	<.001
Iteration	Independent Variable	16.67	9.90	1.68	.093
R^2 = .09	Moderator	2.79	2.66	1.04	.295
	Ind. x Mod. Interaction	-2.49	10.44	-0.24	.812
	Constant	67.37	5.36	12.57	<.001
NOT	Independent Variable	-2.14	6.76	-0.32	.751
R^2 = .04	Moderator	-6.09	7.01	-0.87	.386
	Ind. x Mod. Interaction	14.62	8.57	1.71	.089
	Constant	54.19	3.78	14.35	<.001
Output	Independent Variable	17.34	4.71	3.68	<.001
R^2 = .07	Moderator	6.20	6.90	0.90	.370
	Ind. x Mod. Interaction	-5.48	7.64	-0.72	.474
	Constant	62.66	2.68	23.35	<.001
OR	Independent Variable	9.48	5.19	1.83	.069
R^2 = .09	Moderator	1.79	3.41	0.52	.601
	Ind. x Mod. Interaction	4.67	6.11	0.76	.446

	Constant	58.49	2.80	20.93	<.001
Parallelism	Independent Variable	24.33	6.44	3.78	<.001
$R^2 = .13$	Moderator	3.32	3.75	0.89	.376
	Ind. x Mod. Interaction	-8.13	7.34	-1.11	.269
	Constant	61.12	2.37	25.81	<.001
Variable Assignment	Independent Variable	18.92	5.22	3.63	<.001
$R^2 = .16$	Moderator	0.34	3.23	0.10	.917
	Ind. x Mod. Interaction	-2.62	5.95	-0.44	.659
	Constant	55.28	3.72	14.85	<.001
Variable Naming	Independent Variable	16.69	4.90	3.41	<.001
$R^2 = .08$	Moderator	6.38	5.52	1.16	.249
	Ind. x Mod. Interaction	-5.11	6.68	-0.76	.445

Note. Independent Variable is the Mental Model being tested.

Moderator is Prior Programming Experience.

Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .005$ (i.e., .05 divided by 10) were deemed to be reliable for purposes of interpretation.

Table 5.21

Moderation Analysis Between Prior Programming Experience (Yes/No) and Confidence Factors When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1

Confidence Factor	Regression Model	*b*	*SE*	*t*	*p*
Estimation of how difficult learning to program is R^2 = .04	Constant	60.91	8.25	7.38	<.001
	Independent Variable	0.73	1.19	0.61	.540
	Moderator	20.97	9.19	2.28	.023
	Ind. x Mod. Interaction	-2.60	1.37	-1.90	.058
How much they fear learning to program R^2 = .09	Constant	67.87	4.13	16.45	<.001
	Independent Variable	-0.49	0.84	-0.58	.564
	Moderator	11.92	4.68	2.55	.011
	Ind. x Mod. Interaction	-2.24	1.02	-2.21	.028
Self-Efficacy Factor 1 (Independence and Persistence) R^2 = .07	Constant	66.47	4.75	14.00	<.001
	Independent Variable	-0.21	1.33	-0.16	.875
	Moderator	-14.57	6.98	-2.09	.038
	Ind. x Mod. Interaction	4.40	1.70	2.59	.010
Self-Efficacy Factor 3 (Self-Regulation) R^2 = .04	Constant	68.41	5.86	11.67	<.001
	Independent Variable	-0.72	1.51	-0.48	.634
	Moderator	-9.66	7.78	-1.24	.215
	Ind. x Mod. Interaction	3.68	1.89	1.94	.053
Self-Efficacy Factor 4 (Simple Programming Tasks) R^2 = .14	Constant	58.60	4.05	14.46	<.001
	Independent Variable	2.58	1.26	2.05	.042
	Moderator	-14.00	6.16	-2.27	.024
	Ind. x Mod. Interaction	2.94	1.56	1.89	.060
Confidence – Variable Assignment R^2 = .10	Constant	57.65	3.79	15.20	<.001
	Independent Variable	0.17	.068	2.56	.011
	Moderator	-1.18	5.37	-0.22	.826
	Ind. x Mod. Interaction	0.04	0.08	0.42	.672
Confidence – Conditional Statements R^2 = .08	Constant	54.01	6.43	8.40	<.001
	Independent Variable	0.17	0.09	1.94	.054
	Moderator	-5.90	8.76	-0.67	.501
	Ind. x Mod. Interaction	0.11	0.11	1.01	.314

	Constant	60.27	3.18	18.95	<.001
Confidence – Iteration	Independent Variable	0.16	0.07	2.24	.026
$R^2 = .13$	Moderator	-4.87	4.47	-1.09	.277
	Ind. x Mod. Interaction	0.09	0.09	1.09	.277
	Constant	53.28	5.07	10.51	<.001
Confidence – All questions	Independent Variable	0.23	0.09	2.69	.008
$R^2 = .12$	Moderator	-4.52	6.83	-0.66	.509
	Ind. x Mod. Interaction	0.07	0.11	0.68	.495

Note. Independent Variable is the Confidence Factor being tested.
Moderator is Prior Programming Experience.
Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .006$ (i.e., .05 divided by 9) were deemed to be reliable for purposes of interpretation.
*n = 285

Similarly, only two interactions between prior programming experience and confidence factors (how much students fear learning to program and self-efficacy Factor 1), surpassed the standard significance threshold, but again failed to reach the adjusted threshold. The Johnson-Neyman analysis revealed that the relationship was significant between students' fear of learning to program and their Assessment 1 results amongst those with prior programming experience, b = -2.73, t = -4.78, p = <.001, but not amongst those without prior programming experience, b = -0.486, t = -0.58, p = .564. Additionally, it was also found that the relationship between self-efficacy Factor 1 and Assessment 1 results was significant amongst those with prior programming experience, b = 4.20, t = 3.97, p = <.001, but not amongst those without prior experience, b = -0.21, t = -0.16, p = .875.

These results suggest that prior programming experience should also not be considered as acting as a moderator variable between students' confidence factors and their Assessment 1 results. When taking into account that prior programming experience also does not appear to be a reliable moderator between students' mental model estimates and their Assessment 1 results, this supports the decision to not include prior programming experience within the predictive model.

A similar analysis was also undertaken in order to establish whether previously studying computer science indirectly affects students' Assessment 1 results indirectly, given that like prior programming experience, the likelihood of a direct link does not appear to be very strong as the Mann Whitney U test performed in Table 5.17 surpasses the standard significance threshold, but not the adjusted one. Additionally, there was only a small increase in students' assessment results between those who did previously study computer science, $M = 71.12$, $SD = 19.25$, and those who did not, $M = 66.29$, $SD = 21.56$. It should be noted that previously studying computer science was not included in the predictive model due to the Mann Whitney U test performed in Table 4.3, which did not indicate a significant relationship with students' Assessment 1 results within the training dataset.

As before, mental model estimates and confidence factor ratings that showed significant differences between those who did and did not previously study computer science (as shown in Tables 5.22 and 5.23), were included in the moderation analysis, which is presented in Tables 5.24 and 5.25.

Table 5.22

Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) And Mental Model Estimates, Conducted on All Available Data at T1

Mental Model	***U***	***z***	***p***	***r***
AND	7057.00	-2.20	.028	0.13
Conditional Statements	5537.00	-4.42	<.001	0.26
IF	6485.50	-2.93	.003	0.17
Iteration	5297.00	-5.19	<.001	0.31
NOT	6131.50	-3.53	<.001	0.21
Output	5524.50	-4.46	<.001	0.26
OR	6537.00	-2.85	.004	0.17
Parallelism	5321.00	-4.78	<.001	0.28
Variable Assignment	4982.00	-5.41	<.001	0.32
Variable Naming	7591.50	-1.23	.221	0.07

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .005$ (i.e., .05 divided by 10) were deemed to be reliable for purposes of interpretation.

*$n = 285$

Table 5.23

Mann Whitney U Tests Between Previously Studying Computer Science (Yes/No) and Confidence Factors Conducted on All Available Data at T1

Confidence Factor	***U***	***z***	***p***	***r***
Estimation of how difficult learning to program is	5874.00	-3.93	<.001	0.23
Estimation of how difficult they find mathematics	8316.00	-0.01	.991	0.00
Estimation of how difficult their degree is	7992.50	-0.54	.588	0.03
How much they fear learning to program	6447.00	-3.01	.003	0.18
Self-Efficacy Factor 1 (Independence and Persistence)	5171.50	-5.01	<.001	0.30
Self-Efficacy Factor 3 (Self-Regulation)	6933.50	-2.21	.027	0.13
Self-Efficacy Factor 4 (Simple Programming Tasks)	3817.50	-7.16	<.001	0.42
Confidence – Variable Assignment	5018.50	-5.25	<.001	0.31
Confidence – Conditional Statements	5655.50	-4.24	<.001	0.25
Confidence – Iteration	4622.00	-5.88	<.001	0.35
Confidence – All Questions	4756.00	-5.66	<.001	0.34
Mental Effort – Variable Assignment	7114.50	-1.03	.304	0.06
Mental Effort – Conditional Statements	7467.00	-0.43	.666	0.03
Mental Effort – Iteration	7611.00	-0.19	.851	0.02

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .004$ (i.e., .05 divided by 14) were deemed to be reliable for purposes of interpretation.

*$n = 285$

Table 5.24

Moderation Analysis Between Previously Studying Computer Science (Yes/No) and Mental Model Estimates When Predicting Students' Assessment 1 Results, Conducted on All Available Data at T1

Mental Model	Regression Model	*b*	*SE*	*t*	*p*
	Constant	62.30	2.58	24.13	<.001
Conditional Statements	Independent Variable	12.95	4.98	2.60	.010
$R^2 = .13$	Moderator	-0.87	3.38	-0.26	.797
	Ind. x Mod. Interaction	3.64	5.80	0.63	.530
	Constant	60.28	6.55	9.21	<.001
IF	Independent Variable	7.27	7.48	0.97	.332
$R^2 = .05$	Moderator	-3.74	7.94	-0.47	.638
	Ind. x Mod. Interaction	9.61	8.96	1.07	.285
	Constant	65.14	2.00	29.60	<.001
Iteration	Independent Variable	18.32	8.96	2.05	.042
$R^2 = .08$	Moderator	2.47	2.71	0.91	.362
	Ind. x Mod. Interaction	-4.13	9.559	-0.43	-.666
	Constant	60.88	5.30	11.49	<.001
NOT	Independent Variable	7.73	6.88	1.12	.262
$R^2 = .02$	Moderator	5.00	7.038	0.71	.477
	Ind. x Mod. Interaction	-1.39	8.72	-0.16	.873
	Constant	53.32	4.21	12.67	<.001
Output	Independent Variable	18.08	5.05	3.58	<.001
$R^2 =.08$	Moderator	6.22	6.37	0.98	.329
	Ind. x Mod. Interaction	-5.33	7.15	-0.75	.456
	Constant	62.06	2.76	22.47	<.001
OR	Independent Variable	12.56	5.23	2.40	.017
$R^2 = .09$	Moderator	2.60	3.45	0.75	.452
	Ind. x Mod. Interaction	0.67	6.14	0.11	.913

Parallelism	Constant	59.80	2.96	20.18	<.001
$R^2 = .12$	Independent Variable	18.93	6.15	3.08	.002
	Moderator	1.22	3.80	0.32	0.75
	Ind. x Mod. Interaction	-1.24	7.05	-0.18	.860
Variable Assignment	Constant	61.64	2.51	24.58	<.001
$R^2 = .15$	Independent Variable	15.95	4.97	3.21	.002
	Moderator	-0.50	3.27	-0.15	.879
	Ind. x Mod. Interaction	1.05	5.71	0.18	.854

Note. Ind. (Independent Variable) is the Confidence Factor being tested.

Mod. (Moderator) is previously studying computer science.

Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .006$ (i.e., .05 divided by 8) were deemed to be reliable for purposes of interpretation.

*$n = 285$

Table 5.25

Moderation Analysis Between P*reviously* S*tudying Computer Science (*Y*es/*N*o) and* Co*nfidence* F*actors* W*hen Predicting* S*tudents' Assessment 1 Results, Conducted on All Available Data at T1*

Confidence Factor	Regression Model	*b*	*SE*	*t*	*p*
Estimation of how difficult learning to program is R^2 = .05	Constant	53.723	9.50	5.65	<.001
	Independent Variable	1.88	1.39	1.36	.176
	Moderator	28.54	10.26	2.78	.006
	Ind. x Mod. Interaction	-3.87	1.53	-2.53	.012
How much they fear learning to program R^2 = .10	Constant	66.5	4.16	15.99	<.001
	Independent Variable	-0.05	0.88	-0.06	.950
	Moderator	13.59	4.70	2.89	.004
	Ind. x Mod. Interaction	-2.79	1.03	-2.70	.007
Self-Efficacy Factor 1 (Independence and Persistence) R^2 =.06	Constant	63.27	4.82	13.13	<.001
	Independent Variable	0.89	1.26	0.70	.483
	Moderator	-8.83	6.85	-1.29	.198
	Ind. x Mod. Interaction	2.78	1.63	1.71	.089
Self-Efficacy Factor 4 (Simple Programming Tasks) R^2 =.13	Constant	57.01	4.23	13.49	<.001
	Independent Variable	3.09	1.23	2.52	.012
	Moderator	-8.85	5.98	-1.48	.140
	Ind. x Mod. Interaction	1.76	1.49	1.18	.240
Confidence – Variable Assignment R^2 =.10	Constant	58.83	4.22	13.94	<.001
	Independent Variable	0.15	0.07	2.04	.042
	Moderator	-2.84	5.47	-0.52	.604
	Ind. x Mod. Interaction	0.07	0.087	0.746	.456
Confidence – Conditional Statements R^2 =.08	Constant	53.28	7.15	7.45	<.001
	Independent Variable	0.185	0.10	1.91	.058
	Moderator	-4.27	9.03	-0.47	.636
	Ind. x Mod. Interaction	0.09	0.12	0.74	.458
Confidence – Iteration R^2 =.13	Constant	60.15	3.40	17.71	<.001
	Independent Variable	0.17	0.07	2.29	.023
	Moderator	-4.17	4.50	-0.93	.354
	Ind. x Mod. Interaction	0.08	0.09	0.93	.353

	Constant	53.60	5.59	9.59	<.001
Confidence – All Questions	Independent Variable	0.23	0.09	2.45	.015
R^2 =.12	Moderator	-4.42	7.01	-0.63	.529
	Ind. x Mod. Interaction	0.07	0.11	0.66	.509

Note. Independent Variable is the Confidence Factor being tested.
Moderator is previously studying computer science.
Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .006$ (i.e., .05 divided by 8) were deemed to be reliable for purposes of interpretation.
*n = 285

The moderation analysis revealed no significant interactions were found between students' mental model estimates and previously studying computer science when attempting to predict students' Assessment 1 results. However, two confidence factors (how difficult students believe learning to program to be and how much students fear learning to program), were identified as having potentially significant interactions with previously studying computer science.

The interaction between previously studying computer science and students' ratings for how difficult students believe learning to program to be surpassed the standard significance threshold of .05, but not the adjusted significance threshold. However, a Johnson-Neyman analysis revealed that a significant relationship existed between those who did previously study computer science and their rating for how difficult they believe learning to program to be, b = -1.984, t = -3.081, p = .002. However, this was not significant amongst students who did not previously study Computer Science, b = 1.884, t = 1.358, p = .176.

The interaction between students' ratings of how much they fear learning to program and whether they had previously studied computer science almost reached the adjusted significance threshold of $p < .006$. Follow-up analyses using the Johnson-Neyman test revealed a significant relationship between students' level of fear of learning to program and their Assessment 1 results amongst those who did previously study computer science, b = -2.852, t = -5.191, p = <.001, but not those who did not previously study it, b = -0.055, t = -0.062, p = .950.

Although previously studying computer science appears to indirectly exert a degree of influence on students' Assessment 1 results through the levels of fear associated with learning to program, the fact that no significant interactions take place with all mental model estimates and the majority of the confidence factors, supports the decision not to include whether students have previously studied computer science within the predictive model. Both prior programming experience and previously studying computer science cannot generally be considered to be moderator variables when attempting to predict students' Assessment 1 results. Although both variables have been shown to significantly influence students' responses to the Programming Checkup at T1, the influence of both variables has been seen to decrease by T2 and subsequently, their influence over students' Assessment 1 is also limited given that this takes place within a few weeks of the T2 data collection.

Students who exhibit high levels of anxiety towards learning to program and low levels of self-confidence in their abilities have been shown to encounter an "almost physical barrier" to learning to program (Rogerson & Scott, 2010, p.167). The results presented in Table 5.26 depict the relationships between each of the confidence factors examined within the Programming Checkup at T1 and students' Assessment 1 results. This repeats the analysis conducted during the feature selection process, as shown in Table 4.5, where the following features were retained within the predictive model:

- Estimation of how difficult learning to program is
- Estimation of how difficult they find mathematics
- How much they fear learning to program
- Self-efficacy Factor 1 (Independence and Persistence)
- Self-efficacy Factor 3 (Self-Regulation)
- Self-efficacy Factor 4 (Simple Programming Tasks)
- Confidence (all questions)

Table 5.26

Spearman's Rank Correlation Tests Between Assessment 1 Results and Confidence Features, Conducted on All Available Data at T1

Confidence Factor	r_s	***p***
Estimation of how difficult learning to program is	-.15	.012
Estimation of how difficult they find mathematics	-.09	.133
Estimation of how difficult their degree is	.04	.493
How much they fear learning to program	-.29	<.001
Self-Efficacy Factor 1 (Independence and Persistence)	.22	<.001
Self-Efficacy Factor 3 (Self-Regulation)	.13	.025
Self-Efficacy Factor 4 (Simple Programming Tasks)	.39	<.001
Confidence – Variable Assignment	.34	<.001
Confidence – Conditional Statements	.29	<.001
Confidence – Iteration	.39	<.001
Confidence – All Questions	.38	<.001
Mental Effort – Variable Assignment	-.12	.046
Mental Effort – Conditional Statements	-.09	.119
Mental Effort – Iteration	-.10	.113

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .004$ (i.e., .05 divided by 14) were deemed to be reliable for purposes of interpretation.

*$n = 285$

The results of the tests conducted on the full dataset are analogous to those of the tests conducted during Feature Selection, whereby students' ratings for how difficult they believe their degrees to be as well as their ratings of mental effort when answering questions on Conditional Statements and Iteration, showed no significant relationship with their Assessment 1 results and, as such, were not included in the predictive model. Although students' mental effort ratings for answering questions on Variable Assignment surpassed the standard significance threshold of $p < .05$ (but not the adjusted threshold), the original decision not to include any mental effort ratings in the predictive model is felt to still be justified, given that the correlations between students' mental effort ratings for Conditional Statements and Iteration did not surpass the standard significance threshold.

Students' estimations of how difficult they find mathematics was retained in the model due to a strong relationship being identified with the dichotomised assessment results used for classification, as shown in Table 4.4, despite a non-significant relationship being identified with the continuous assessment result used for the regression models. In order to ensure consistency, the same features were used for both classification and regression models.

The link between mathematics and programming is evident in the literature (i.e. Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Kaufmann & Stenseth, 2021; Wilson & Shrock, 2001), and a near-significant difference (when applying the Bonferroni correction) in Assessment 1 results between students who did or did not previously study a mathematics-based subject, has already been identified within the full T1 dataset. However, the results in Table 5.26 confirmed that the relationship between students' estimations of how difficult they find mathematics and their Assessment 1 results was not significant. It should be noted that students' estimations as to how difficult it is to learn to program was included in the model was as it surpassed the standard significance threshold of
$p < .05$ in Table 4.5. However, it fails to reach the adjusted significance threshold when evaluated on all data available at T1, with the correlation being too weak to reliably state that there is a substantial relationship between students' views of how difficult learning to program will be and their actual performance.

Students' confidence in their answers for each of the question topics, and for all questions combined, were revealed to have some of the strongest correlations with the Assessment 1 results. Although the assessment takes place nearer to T2 than T1, these results do indeed support the link between students' initial self-confidence and their performance given that significant relationships between students' confidence in their answers and their estimates of having appropriate mental models have previously been identified. Additionally, as per Rogerson and Scott's (2010) claims, a significant relationship between students' level of fear of learning to program and their Assessment 1 results was reported.

Students' levels of self-efficacy relating to completing simple programming tasks (Factor 4) also demonstrates a comparatively strong correlation with their Assessment 1 results. Furthermore, students' levels of self-efficacy relating to Independence and Persistence also exhibit a significant correlation with their Assessment 1 results, albeit weaker than that with

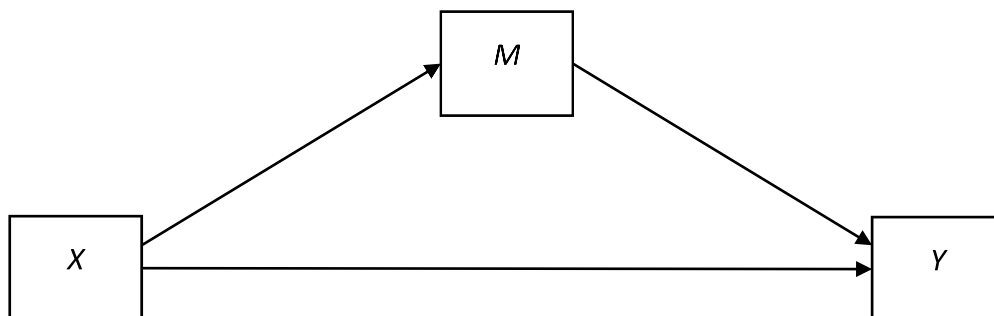
Factor 4. However, their self-efficacy levels relating to Self-Regulation (Factor 3) failed to produce a correlation that surpasses the adjusted significance threshold.

The results in Table 5.26 do therefore, generally support the decisions being made during the feature selection process, although if only a regression model was being considered it would be appropriate to not include how difficult students find mathematics within the model. The strongest correlations with students' Assessment 1 grades were found to be with features that directly mapped to students' confidence in the core content of the module and the assessment itself, that is, completing simple programming tasks and their understanding of the key concepts being taught, which are essential for students to be able to do in order to pass the assignment. It is evident that how much students fear learning to program also links to their assessment results, suggesting that some students' learning may indeed be being blocked due to a lack of confidence by, for example, not feeling confident enough to ask questions when they are struggling (Bergin & Reilly, 2005a; Rogerson & Scott, 2010).

Wiedenbeck et al. (2004) stated that students "pre-self-efficacy" within their study does not directly affect their performance; rather, "post-self-efficacy" acts as a mediator variable whereby it allows performance to be affected by pre-self-efficacy by passing through post-self-efficacy (Baron & Kenny, 1986; Hayes, 2022; Wiedenbeck et al., 2004). Hayes (2022) provides a conceptual diagram of the "simple mediation model", as shown in Figure 5.3, where *X* represents an independent variable (i.e., pre-self-efficacy), *M* represents the mediator variable (i.e., post-self-efficacy) and *Y* represents the outcome variable (i.e., performance).

Figure 5.3

Conceptual Diagram of a Simple Mediation Model



Note. From "Introduction to Mediation, Moderation and Conditional Process Analysis: A Regression Based Approach, Third Edition," by A. F. Hayes, Guilford Publications, 2022.

As the diagram shows, there are two paths from *X* to *Y* – either the direct path which passes from *X* to *Y* without passing through *M,* thus showing pre-self-efficacy's *direct effect* on performance, or through the second, indirect, path from *X* to *Y*, which passes through *M*. This second path demonstrates the *indirect effect* where the influence of *X* on *Y* is indirect and instead is a result of *X*'s influence on *M*, which in turn influences *Y* (Hayes, 2022). This captures how Wiedenbeck et al.'s (2004) pre-self-efficacy levels influence post-self-efficacy, which consequently influences performance. Wiedenbeck et al.'s (2004) pre- and post-self-efficacy readings are akin to the T1 and T2 Programming Checkup data collections. As such, the subset of responses which were involved in both rounds of data collection was used to carry out a Mediator analysis using the PROCESS Macro, in order to evaluate whether self-efficacy, or any other confidence factor, at T2 is acting as a mediator variable.

The results from the Mediation analysis are presented in Table 5.27, with there being three possible conclusions for each factor in the analysis: No significant mediation, Partial meditation (whereby the relationships between mediator and outcome variables are significant as well as the relationships between the independent and outcome variables), and Complete Mediation (where the direct relationship between independent and outcome variables is insignificant). However, Hayes (2022) recommends that the results of mediation analyses should not be reviewed using these terms given their sensitivity to sample size. Therefore, any factor that has been concluded as having either partial or complete mediation will be considered to be significant.

Table 5.27

Mediator Analysis Conducted on Confidence Factors When Predicting Assessment 1 Results

Confidence Factor	Total Effect	Direct Effect	Indirect Effect	Indirect Effect CI LB	Indirect Effect CI UB	t	Conclusion
Estimation of how difficult learning to program is	-2.05	-0.47	-1.59	-2.95	-0.332	-2.33	Partial Mediation
Estimation of how difficult they find mathematics	-1.65	-1.75	0.10	-1.118	1.505	0.15	No Significant Mediation
Estimation of how difficult their degree is	0.81	1.70	-0.89	-2.412	0.159	-1.35	No Significant Mediation
How much they fear learning to program	-2.26	-1.35	-0.91	-2.624	-0.075	-2.50	Complete Mediation
Self-Efficacy Factor 1 (Independence and Persistence)	2.83	0.88	1.95	0.785	3.364	3.01	Complete Mediation
Self-Efficacy Factor 3 (Self-Regulation)	2.79	1.81	0.98	-0.004	2.170	1.79	No Significant Mediation
Self-Efficacy Factor 4 (Simple Programming Tasks)	3.84	2.08	1.76	0.383	3.433	2.28	Partial Mediation
Confidence – Variable Assignment	0.21	0.16	0.05	0.003	0.101	1.81	Partial Mediation
Confidence – Conditional Statements	0.20	0.16	0.04	-0.016	0.119	1.24	No significant mediation
Confidence – Iteration	0.23	0.16	0.07	0.016	0.143	2.21	Partial Mediation
Confidence – All Questions	0.26	0.21	0.06	-0.011	0.136	1.50	No Significant Mediation
Mental Effort – Variable Assignment	-0.98	-0.67	-0.31	-0.816	0.018	-1.44	No Significant Mediation
Mental Effort – Conditional Statements	-1.00	-0.83	-0.17	-0.664	0.130	-0.86	No Significant Mediation
Mental Effort – Iteration	-0.53	-0.34	-0.20	-0.701	0.083	-3.40	No Significant Mediation

Note. Mediation analysis conducted on subset of dataset where participants had taken part in both T1 and T2 data collections

*n = 119

Two of the three self-efficacy factors (Factors 1 and 4) were revealed to be mediated by students' responses at T2. Although Factor 3 does not appear to be mediated, this generally does support Wiedenbeck et al.'s (2004) claims as they did not break down the self-efficacy ratings into the individual factors. Furthermore, a number of other factors appear to be mediated by students' responses at T2 including how difficult students feel learning to program is, how much students fear learning to program and students' confidence in their answers for questions on Variable Assignment and Iteration. Although not every factor included in the predictive model can be considered to be mediated by students' responses at T2, clear relationships between a number of different variables that are related to students' confidence levels at T1 and success within their first assessment have been established.

All mental model estimates were included within the predictive model, with the exception of AND, OR, NOT and IF, given that the Conditional Statements model was found to have a stronger relationship with students' Assessment 1 results whilst also accounting for each of the individual concepts within a single model. Table 5.28 presents the Spearman's Rank analysis between students' mental model estimates and their Assessment 1 results using all available data at T1. Although the strength of the correlations range from weak to moderate at best, the results do support the original decision to retain the Conditional Statements model instead of the models associated with each of the individual concepts, given that Conditional Statements again showed the strongest relationship when compared to the individual concepts.

A mediation analysis was also carried out in order to examine the causal paths between students' estimates of having appropriate mental models at T1 and T2, and their influence on students' assessment results, as shown in Table 5.29. Consequently, it was necessary to also conduct this analysis on the subset of students who completed the Programming Checkup at both T1 and T2.

Table 5.28

Spearman's Rank Correlation Tests Between Assessment 1 Grades and Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1

Mental Model	r_s	p
AND	.23	<.001
Conditional Statements	.39	<.001
IF	.33	<.001
Iteration	.47	<.001
NOT	.25	<.001
Output	.40	<.001
OR	.31	<.001
Parallelism	.38	<.001
Variable Assignment	.43	<.001
Variable Naming	.23	<.001

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .005$ (i.e., .05 divided by 10) were deemed to be reliable for purposes of interpretation.

*$n = 285$

Table 5.29

Mediator Analysis Conducted on Mental Model Estimates When Predicting Assessment 1 Results

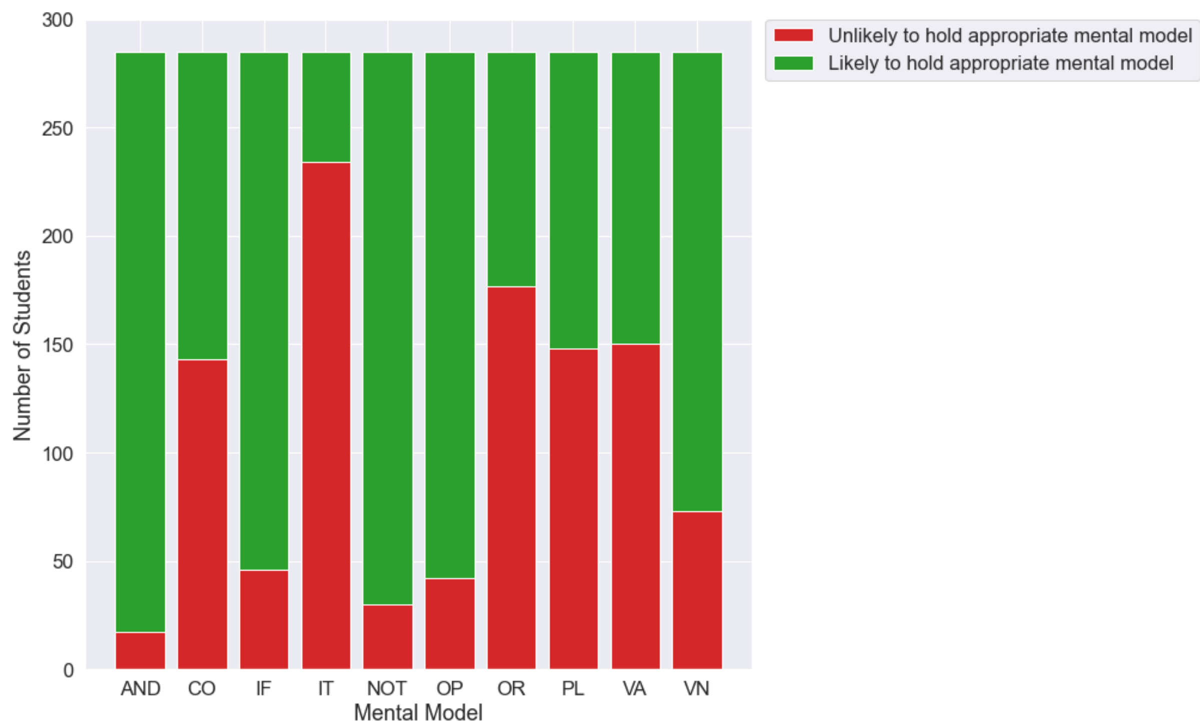
Mental Model	Total Effect	Direct Effect	Indirect Effect	Indirect Effect CI LB	Indirect Effect CI UB	t	Conclusion
AND	1.04	0.67	0.37	-1.78	2.60	0.33	No Significant Mediation
Conditional Statements	12.07	8.86	3.21	0.47	6.66	2.03	Partial Mediation
IF	22.67	22.66	0.01	-1.65	0.57	0.01	No Significant Mediation
Iteration	14.65	4.51	10.15	6.15	15.01	4.51	Complete Mediation
NOT	2.86	-0.29	3.15	0.41	8.47	1.48	Complete Mediation
Output	8.27	8.63	-0.36	-1.67	0.26	-0.71	No Significant Mediation
OR	6.46	2.31	4.15	-1.71	10.42	1.36	No Significant Mediation
Parallelism	14.57	11.92	2.65	0.26	5.97	1.81	Partial Mediation
Variable Assignment	11.24	9.79	1.44	0.09	3.68	1.56	Partial Mediation
Variable Naming	14.92	14.71	0.21	-1.33	1.89	0.28	No Significant Mediation

Note. Mediation analysis conducted on subset of dataset where participants had taken part in both T1 and T2 data collections.

*n = 119

Figure 5.4

Estimates of Whether Students Hold Appropriate Mental Models at T1 (Using All Available Data), Established Using Bayesian Knowledge Tracing with a Threshold of 0.5



The strongest correlation presented within Table 5.28 was between students' Assessment 1 results and their estimates of holding an appropriate model of Iteration (r_s = .47). Figure 5.4 incorporates the previously isolated holdout-test set, and subsequently presents similar results to that of Figure 5.2. Both Figures 5.2 and 5.4 show the vast majority of students were deemed unlikely to hold an appropriate model for Iteration, which has previously been shown to be a topic of difficulty for students. Despite this, students' estimates of having an appropriate mental model of Iteration at T2 were shown to be a significant mediator between their T1 estimates and their Assessment 1 results, thereby supporting the notion that development of an appropriate model of Iteration is an important step in students' learning.

The next strongest correlation is with Variable Assignment, arguably the most fundamental of concepts which students are required to develop a model of. Again, a significant portion of students were considered to be unlikely to hold an appropriate mental model at T1, mostly due to the MA misconception. However, among students who completed the Programming Checkup at both opportunities, the vast majority were shown to develop an appropriate mental model by T2, which is reflected in their T2 estimates also being considered a significant mediator.

Two further model estimates which showed correlations of reasonable strength (relative to other mental models) with students' Assessment 1 results were Conditional Statements and Parallelism. Both had significant proportions of students who were not considered likely to have appropriate models at T1, although they did not have quite as large a drop in the number of students being considered unlikely to hold an appropriate model as Variable Assignment. However, both were identified as being mediated by students' responses at T2.
The only other model estimate which was identified as being mediated by students' estimates at T2 was NOT. However, it had one of the weakest correlations with students' Assessment 1 results and was also not included in the predictive model.

Although estimates for other models were not considered to be significant mediators, it is expected that students' models will improve over time if students are able to overcome any misconceptions they are holding. Nevertheless, the estimates of how likely that a student is to hold an appropriate model for each of the core concepts has previously been shown to aid in the prediction of students' assessment results. All the mental models assessed within the Programming Checkup are required to be used within the assessment students undertake. However, it is interesting to note that the strongest correlations with students' Assessment 1 results were with several of the models that significant proportions of students were considered unlikely to hold at T1.

In addition to the mediation analysis conducted between students' responses to the Programming Checkup at T1 and T2, and their Assessment 1 results, an additional mediation analysis was carried out in order to examine the casual effect between students' levels of confidence, their mental model estimates and their Assessment 1 results. Only confidence factors that were significantly correlated with students' Assessment 1 results (see Table 5.26) were included in this mediation analysis. However, as this mediation analysis only requires students' responses at T1, all available data were used. From the results presented in Table 5.30 it can be determined that students' ratings for self-efficacy Factor 3 did not act as a mediating variable between their estimates of holding appropriate mental models for each of the concepts being examined and their Assessment 1 results. Furthermore, students' estimates of how difficult they believe learning to program to be cannot be considered to be a mediating variable given that no significant mediation was determined for the majority of the mental model estimates.

However, the remaining factors, including how much students fear learning to program, self-efficacy Factors 1 and 4, and students' confidence in answering questions focusing on Variable Assignment, Conditional Statements, Iteration, and for all questions combined, can be considered to be reliable mediator variables. This is because a significant mediation effect was established between the majority (or all) of the mental model estimates and students' Assessment 1 results when each of the factors were included as mediators. This, therefore, provides evidence of a causal link whereby the likelihood of holding appropriate mental models directly influences these confidence factors, which in turn influence students' performance. It should, however, be noted that the direct influence of students' mental model estimates on their Assessment 1 results remains significant.

Table 5.30

Mediator Analysis Conducted on Confidence Factors and Mental Model Estimates When Predicting Assessment 1 Results

Confidence Factor (Mediator)	Mental Model	Total Effect	Direct Effect	Indirect Effect	Indirect Effect CI LB	Indirect Effect CI UB	t	Conclusion
Estimation of how difficult learning to program is	AND	16.19	15.90	0.29	-1.12	1.88	0.40	No Significant Mediation
	Conditional Statements	15.78	15.07	0.72	-0.47	1.98	1.17	No Significant Mediation
	IF	14.37	13.29	1.08	0.05	2.74	1.56	Partial Mediation
	Iteration	15.27	14.11	1.15	-0.39	2.98	1.15	No Significant Mediation
	NOT	8.12	7.15	0.96	-0.12	2.62	1.33	No Significant Mediation
	Output	16.09	15.03	1.05	0.08	2.49	1.65	Partial Mediation
	OR	13.54	12.64	0.09	-0.02	2.17	0.17	No Significant Mediation
	Parallelism	18.21	17.45	0.76	-1.02	2.53	0.86	No Significant Mediation
	Variable Assignment	16.71	16.54	0.17	-1.46	1.85	0.21	No Significant Mediation
	Variable Naming	14.88	13.99	0.89	0.04	2.18	1.59	Partial Mediation

How much they fear learning to program	AND	16.19	13.15	3.05	0.64	6.47	2.05	Partial Mediation
	Conditional Statements	15.78	13.35	2.43	0.68	4.52	2.47	Partial Mediation
	IF	14.37	11.73	2.65	0.71	5.14	2.30	Partial Mediation
	Iteration	15.27	12.58	2.68	1.01	4.74	2.80	Partial Mediation
	NOT	8.16	6.28	1.84	-0.13	4.57	1.56	No Significant Mediation
	Output	16.09	13.26	2.82	1.04	5.31	2.60	Partial Mediation
	OR	13.54	10.97	2.57	0.96	4.66	2.69	Partial Mediation
	Parallelism	18.21	15.28	2.94	0.73	5.34	2.53	Partial Mediation
	Variable Assignment	16.71	14.50	2.22	0.18	4.41	2.04	Partial Mediation
	Variable Naming	14.88	12.43	2.45	0.84	4.62	2.54	Partial Mediation
Self-Efficacy Factor 1 (Independence and Persistence)	AND	16.19	14.48	1.71	-0.45	4.96	1.27	No Significant Mediation
	Conditional Statements	15.78	14.39	1.39	0.15	2.98	1.92	Partial Mediation
	IF	14.37	12.52	1.85	0.13	4.27	1.74	Partial Mediation
	Iteration	15.27	13.29	1.98	0.37	3.98	2.17	Partial Mediation
	NOT	8.12	6.06	2.05	0.41	4.51	1.95	Complete Mediation
	Output	16.09	14.19	1.90	0.32	4.22	1.87	Partial Mediation
	OR	13.54	12.62	0.92	-0.06	2.34	1.48	No Significant Mediation
	Parallelism	18.21	16.48	1.74	0.16	3.79	1.87	Partial Mediation
	Variable Assignment	16.71	15.44	1.27	-0.66	3.47	1.23	Partial Mediation
	Variable Naming	14.88	13.20	1.68	0.28	3.61	1.94	Partial Mediation

Self-Efficacy Factor 3 (Self-Regulation)	AND	16.19	16.36	-0.17	-2.17	1.81	-0.18	No Significant Mediation
	Conditional Statements	15.78	15.26	0.52	-0.27	1.61	1.09	No Significant Mediation
	IF	14.37	13.36	1.01	-0.21	2.92	1.27	No Significant Mediation
	Iteration	15.27	14.56	0.71	-0.25	2.14	1.16	No Significant Mediation
	NOT	8.12	7.84	0.28	-0.74	1.90	0.43	No Significant Mediation
	Output	16.09	15.68	0.40	-0.64	1.79	0.69	No Significant Mediation
	OR	13.54	13.56	-0.02	-0.83	0.81	-0.05	No Significant Mediation
	Parallelism	18.21	17.68	0.53	-0.20	1.76	1.05	No Significant Mediation
	Variable Assignment	16.71	16.25	0.46	-0.25	1.50	1.04	No Significant Mediation
	Variable Naming	14.88	14.40	0.48	-0.27	1.73	0.94	No Significant Mediation
Self-Efficacy Factor 4 (Simple Programming Tasks)	AND	16.19	11.02	5.17	1.37	10.19	2.31	Partial Mediation
	Conditional Statements	15.78	11.25	4.53	2.21	7.27	3.46	Partial Mediation
	IF	14.37	9.84	4.53	1.85	7.97	2.88	Partial Mediation
	Iteration	15.27	8.53	6.74	3.66	10.08	4.15	Partial Mediation
	NOT	8.12	3.84	4.28	1.38	7.92	2.57	Complete Mediation
	Output	16.09	9.78	6.31	3.33	10.11	3.67	Partial Mediation
	OR	13.54	9.78	3.76	1.82	6.20	3.35	Partial Mediation
	Parallelism	18.21	12.23	5.98	2.56	9.91	3.22	Partial Mediation
	Variable Assignment	16.71	12.04	4.68	1.29	8.35	2.63	Partial Mediation
	Variable Naming	14.88	9.73	5.16	2.66	8.28	3.60	Partial Mediation

Confidence – Variable Assignment	Variable Assignment	16.71	13.56	3.16	0.22	6.18	2.05	Partial Mediation
Confidence – Conditional Statements	Conditional Statements	15.78	13.01	2.77	0.44	5.48	2.18	Partial Mediation
Confidence – Iteration	Iteration	15.27	7.50	7.77	4.06	11.72	4.02	Partial Mediation
Confidence – All Questions	AND	16.19	8.70	7.49	3.44	12.72	3.17	Complete Mediation
	Conditional Statements	15.78	10.90	4.89	1.97	7.99	3.18	Partial Mediation
	IF	14.37	7.81	6.56	3.38	10.50	3.60	Complete Mediation
	Iteration	15.27	8.53	6.74	3.41	10.33	3.86	Partial Mediation
	NOT	8.12	3.78	4.34	1.56	8.12	0.94	Complete Mediation
	Output	16.09	10.02	6.07	3.01	9.84	1.73	Partial Mediation
	OR	13.54	9.00	4.54	2.30	7.28	1.77	Partial Mediation
	Parallelism	18.21	12.34	5.88	2.47	9.65	1.35	Partial Mediation
	Variable Assignment	16.71	12.22	4.49	1.07	8.21	0.59	Partial Mediation
	Variable Naming	14.88	8.83	6.05	3.20	9.69	1.91	Partial Mediation

Note. Mediation analysis conducted on all available data at T1

*n = 285

5.4 Comparison with Assessment 2 Results

The first assignment that students complete as part of their introductory programming module was chosen as the outcome variable given the fact that it primarily focuses on assessing concepts which are included within the Programming Checkup. However, given that it is the first programming-based assessment students complete then it is relatively simplistic in nature. Students are also required to complete a second assessment at the end of the second semester (approximately 12 weeks after Assessment 1), which is more complex in nature and requires the use of more advanced concepts, such as object-orientation, for higher marks. Of the 285 students who took part in the Programming Checkup at T1 and completed the first assessment, 244 students also completed their second assessment. Unfortunately, the reasons as to why 41 students across the three years of data collection did not take part in their second assessment are not available.

Nevertheless, a significant correlation of moderate strength exists between students' results in their first and second assessments, $r_s = .509$ $p = < .001$. Tables 5.31 – 5.33 utilise the subset of students who completed both assessments to examine how students' results for both assessments relate to their responses to the Programming Checkup at T1. The correlations between students' mental model estimates at T1 and their Assessment 1 results, as shown in Table 5.31, are very similar in strength and significance when compared to their correlations with students' Assessment 2 results. This gives credence to the notion of assessing students' mental models at the beginning of the course, as whilst there is a significant variability in assessment results, they do appear to be an appropriate indicator of how a student is likely to progress throughout their course. It is interesting to note how strongly students' estimates of having an appropriate model for Iteration correlates with both their assessment results, as this is a seemingly difficult concept and yet is key to writing effective programs.

Additionally, the correlations between students' confidence factors in Table 5.32 and their results in their assessment appear to be relatively consistent between both assessments. However, students' estimations of how difficult they find mathematics has a stronger correlation with Assessment 2, which almost reaches the adjusted significance threshold. This corresponds to a slight increase in effect size between students who did or did not previously study a mathematics-based subject, as shown by the Mann Whitney U tests performed in Table 5.33, between Assessments 1 and 2, although despite the increase,

previously studying mathematics can only be considered to have a small effect on Assessment 2 results. However, it should be noted that the Mann Whitney U tests that were performed on the full T1 dataset (as shown in Table 5.17), demonstrated a stronger effect between previously studying mathematics-based subjects and students' Assessment 1 results, than was observed within the tests conducted in Table 5.33, suggesting the need for further investigation into the relationships between students' prior mathematics experience and their performance throughout the course of an introductory programming module.

Wiedenbeck et al. (2004) claimed that students' experiences prior to them starting their introductory courses lose their predictive value over time, as students' more recent experiences within the course become the more dominant factor in their level of confidence, which can also be extended to include the likelihood of students' possessing appropriate mental models of core concepts that aids their confidence levels. The Mann Whitney U tests within Table 5.33 revealed a drop in effect sizes in relation to having either prior programming experience or previously studying computer science between Assessment 1 and Assessment 2, which does provide some support for Wiedenbeck et al.'s (2004) belief in terms of students' computer science and programming related experiences. An additional data collection point closer to Assessment 2 would help to further verify these claims.

Students' intentions for wanting to work in software engineering at the start of their course exhibit a stronger relationship with their Assessment 2 results when analysed with a Kruskal Wallis test, $H(2) = 27.87$, $p = <.001$, $\eta^2 = 0.115$, compared to Assessment 1 $H(2) = 13.56$, $p = .001$, $\eta^2 = 0.056$. Students who wish to pursue a career in software engineering may potentially be more motivated to succussed within their programming module, as previously discussed, which is evident in the results for both assessments as students indicating they do wish to work in a software engineering role after graduating had higher results on average (Ass. 1, $M = 76.11$, $SD = 15.91$, Ass. 2, $M = 68.28$, $SD = 24.17$) than those indicating that they were undecided (Ass. 1, $M = 68.03$, $SD = 19.60$, Ass. 2, $M = 51.73$, $SD = 24.89$) or that they did not want to pursue this career path (Ass. 1, $M = 64.06$, $SD = 19.22$, Ass. 2, $M = 57.25$, $SD = 33.27$). Additionally, students' level of agreement in considering themselves self-taught programmers has a slightly stronger correlation with students' Assessment 2 results, $r_s = .29$ $p = <.001$, than with their Assessment 1 results, $r_s = .27$ $p = <.001$. However, the difference is not substantial enough to draw any firm conclusions.

Table 5.31

Spearman's Rank Correlation Tests Between Assessment Grades and Mental Model Estimates Established Using Bayesian Knowledge Tracing, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments

Mental Model	**Assessment 1**		**Assessment 2**	
	r_s	p	r_s	p
AND	.22	<.001	.23	<.001
Conditional Statements	.35	<.001	.32	<.001
IF	.31	<.001	.31	<.001
Iteration	.45	<.001	.45	<.001
NOT	.23	<.001	.24	<.001
Output	.37	<.001	.30	<.001
OR	.26	<.001	.27	<.001
Parallelism	.34	<.001	.26	<.001
Variable Assignment	.38	<.001	.42	<.001
Variable Naming	.24	<.001	.19	<.001

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 20) were deemed to be reliable for purposes of interpretation.

*n = 244

Table 5.32

Spearman's Rank Correlation Tests Between Assessment Results and Confidence Factors, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments

Confidence Factor	Assessment 1		Assessment 2	
	r_s	p	r_s	p
Estimation of how difficult learning to program is	-.20	.002	-.22	<.001
Estimation of how difficult they find mathematics	-.07	.249	-.19	.003
Estimation of how difficult their degree is	.02	.807	-.04	.520
How much they fear learning to program	-.28	<.001	-.32	<.001
Self-Efficacy Factor 1 (Independence and Persistence)	.24	<.001	.22	<.001
Self-Efficacy Factor 3 (Self-Regulation)	.14	.028	.15	.022
Self-Efficacy Factor 4 (Simple Programming Tasks)	.37	<.001	.37	<.001
Confidence – Variable Assignment	.32	<.001	.34	<.001
Confidence – Conditional Statements	.30	<.001	.28	<.001
Confidence – Iteration	.39	<.001	.38	<.001
Confidence – all questions	.37	<.001	.37	<.001
Mental Effort – Variable Assignment	-.14	.034	-.13	.047
Mental Effort – Conditional Statements	-.10	.132	-.12	.073
Mental Effort – Iteration	-.16	.011	-.15	.021

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .002$ (i.e., .05 divided by 28) were deemed to be reliable for purposes of interpretation.

*n = 244

Table 5.33

Mann Whitney U Tests Between Assessment Grades and Dichotomous Background Features, Conducted on All Available Data at T1 Where Students Had Completed Both Assessments

Background Factor	**Assessment 1**				**Assessment 2**			
	U	***z***	***p***	***r***	***U***	***z***	***p***	***r***
Prior programming experience	4929.50	-2.24	.025	0.14	5388.00	-1.31	.190	0.08
Previously Studied computer science	5179.50	-1.20	.230	0.08	5454.50	-0.63	.528	0.04
Previously Studied mathematics-based subject	6289.00	-1.85	.064	0.12	5609.00	-3.10	.002	0.20
Intend to work in software engineering – No	1285.00	-1.98	.048	0.13	1727.00	-0.36	.722	0.02
Intend to work in software engineering – Undecided	5619.00	-2.63	.009	0.17	4307.00	-5.08	<.001	0.33
Intend to work in software engineering – Yes	5416.00	-3.55	<.001	0.23	4546.00	-5.14	<.001	0.33
English is student's first language	3947.50	-0.33	.745	0.02	3930.50	-0.37	.714	0.02

Note. Given the number of tests being conducted, a Bonferroni correction factor was applied to the standard alpha value of .05 to reduce the chance of Type 1 errors. Only significant differences at $p < .003$ (i.e., .05 divided by 14) are deemed to be reliable for purposes of interpretation.

*n = 244

5.5 Summary

This chapter has presented an in-depth statistical analysis of students' responses to the Programming Checkup, through which relationships between factors relating to their backgrounds, confidence and estimates of holding appropriate mental models have been explored. Furthermore, the decisions taken during the model development process have been reinforced through examinations of relationships with students' Assessment 1 results. Additionally, students' results to Assessment 2 have been used to scrutinize how the factors examined within the Programming Checkup at T1 relate to students' performance over a much longer period of time. The subsequent chapter will draw on the results of these analysis in order to directly answer the research questions at the heart of this investigation.

6. General Discussion and Reflections of Research Outcomes and Future Work

6.1 Scope of Discussion

This final chapter revisits the research questions that guided the path of this work and considers the findings of this research programme in relation to these questions. In addition, this chapter discusses the limitations of the present research that constrain the conclusions that can be drawn and that also inform future work.

6.2 Responses to Research Questions

This design of this investigation, and the later analysis, was guided by three key research questions. As such, this section draws on the literature and results presented throughout this thesis in order to directly address each question.

RQ 1 How do students' mental models of core programming concepts develop during a university introductory programming module?

The use of Bayesian Knowledge Tracing (BKT) to estimate the likelihood of students holding appropriate mental models of core programming concepts has proved to be a useful approach. It accounts for students making mistakes or guessing answers correctly and produces a tangible result, which aids in both the analysis of students' conceptual understanding of programming and the development of predictive models of student performance.

The estimates produced by BKT have revealed that a significant number of students are initially not likely to hold an appropriate mental model for Variable Assignment at T1. Variable Assignment is arguably one of the most fundamental concepts that students must master in order to be successful within their programming course. Students who are deemed to be unlikely to hold an appropriate model of Variable Assignment are often demonstrating the Multiple Assignment (MA) misconception, whereby they refer to the original values of variables, rather than recognising that the original value is overridden when performing an assignment operation. This is evident from the observation that MA is the most prevalent misconception that is associated with Variable Assignment and that the vast majority of students who demonstrate MA do so three or more times.

However, many of the students who are initially estimated as being unlikely to hold an appropriate mental model for Variable Assignment do in fact go on to develop appropriate mental models by the end of the first semester (T2). Again, it appears that students who are considered to be unlikely to hold an appropriate model of Variable Assignment are often exhibiting the MA misconception. Although it is clear that the vast majority of students are able to establish an appropriate model for Variable Assignment by the time they reach T2, it is evident that it can be a stumbling block for some. If a student is not able to establish an appropriate mental model for Variable Assignment, they may face greater difficulties when completing programming tasks, as logical errors could easily be introduced if they do not have a clear understanding of how variables and their values are handled within a program.

As previously discussed, the Conditional Statements mental model encompasses the models associated with individual Boolean Logic concepts including AND, OR and NOT, as well as If Statements. This broader approach to assessing related concepts within a single model has shown to have a stronger relationship with students' assessment results as opposed to the individual concepts. At T1, half of students were estimated as not having an appropriate Conditional Statements model. It is apparent from the results that the largest area of difficulty is with students demonstrating an appropriate understanding of the OR operator, which is reflected in a substantial number of students demonstrating it as a misconception three or more times, subsequently resulting in more than half of the students being estimated to not hold an appropriate mental model of OR at T1. A substantial number of students also demonstrated misconceptions relating to If statements, AND and NOT at T1, although for the most part, students only demonstrated these misconceptions once or twice, meaning a greater proportion of students were considered to have appropriate mental models for these concepts. However, as each of the concepts is accounted for within the Conditional Statements model, the estimates associated with it are indicative of a more general view of students' logical abilities. None of the individual model estimates change significantly between T1 and T2, but Conditional Statements shows a decrease that is approaching significance, although 39% of students are still considered to be unlikely to hold an appropriate model at T2. As Grover and Basu (2017) suggest, Boolean Logic is a difficult topic for students to grasp, with OR appearing to be one of the main points of confusion for students.

It appears that the concept that students struggle to comprehend the most is Iteration, as 81% of students were estimated to not be holding an appropriate model at T1, which only reduces

to 67% at T2. Students exhibit a wide range of misconceptions associated with Iteration, the frequencies of which do not significantly alter between T1 and T2, with the exception of Summation (SM). The Summation misconception occurs when a student views an iterative loop as a single element, that is, instead of outputting a series of numbers the student indicates that only the final number would be outputted. There is a substantial drop in the number of occurrences of SM between T1 and T2 and although many of the students who demonstrated SM at T1 did so only once, this may be indicative of the fact that students are beginning to progress towards more appropriate models by recognising that all lines within the loop are being repeated.

However, as mentioned previously, this latter finding could also potentially be attributed to students developing more accurate models of Program Output. Most students were estimated to be holding an appropriate mental model for program output at T1 and T2, despite the seemingly high number of students demonstrating misconceptions relating to output, which can be accounted for by the fact that almost all of the questions included print statements. There was, however, a significant change in the estimates of how likely students were to be holding an appropriate model for Output, which could have supported the reduction in the SM misconception. Further work is required to validate this view. Nevertheless, what is clear from the evidence is that students generally lack an appropriate mental model for Iteration prior to beginning their programming course, and although some are able to develop their models independently, the T2 mental model estimates indicate that a significant proportion of students are still in need of additional support in order to develop fully accurate models.

An additional area of difficulty for students appears to be with their understanding of flow of control within a program, which is represented by the Parallelism model. Around half of students were considered to be unlikely to hold an appropriate model at T1, which reduces to 41% at T2. There is, therefore, still a significant number of students who do not appear to have a complete understanding of how the ordering of statements within a program affects the output, which could consequently lead to the student experiencing difficulties when writing their own programs. These results provide evidence of Pea's (1986) "parallelism bug" being encountered by a substantial number of students, and by extension, can be viewed as also giving evidence of the "Superbug", which Pea describes as being associated with the idea that students believe that the programming language is, in some way, intelligent. For the

Parallelism bug, this relates to students' belief that "different lines in a program can be somehow known by the computer at the same time, or in parallel" (Pea, 1986, p. 5).

Furthermore, students who may mistakenly believe that the name of a variable affects what it can hold, could be viewed as holding Pea's (1986) "Intentionality bug", which is also associated with the "Superbug". However, the majority of students demonstrated that they held an appropriate mental model associated with Variable Naming and although some students were still considered to be unlikely to hold an appropriate model at T2, this can generally be considered not to be a widespread issue. Without conducting detailed walkthroughs and interviews, it is difficult to ascertain whether students in higher education have an implicit belief in the intelligence of the programming language, as proposed by Pea's (1986) notion of the Superbug in the context of younger students. Following up on this possibility represents an important avenue for future research.

Students' mental model estimates have shown to be significantly correlated with the results they achieve in their assessments. For the most part, these correlations have been relatively weak. However, students' estimates of having an appropriate model of Iteration have shown to exhibit the strongest relationship with students' assessment results. Iteration could potentially therefore be seen as a Threshold Concept, which Meyer and Land (2005) define as being *transformative*, whereby they change how a student looks at the subject, *irreversible*, as they will be difficult for students to forget once they have been mastered, *integrative*, through the way that multiple concepts are drawn together, *troublesome*, as they can be difficult for a student to grasp and can often act as *boundary markers* which represent the limits of a student's understanding (Boustedt et al., 2007). Previous work has made claims that potential programming-related threshold concepts could consist of more advanced concepts such as pointers, object-orientation (Boustedt et al., 2007), or more abstract principles such as program dynamics, information hiding and object interaction (Sorva, 2010). However, I believe it is also appropriate to consider more fundamental concepts as threshold concepts. For example, Iteration as a concept clearly fits within Boustedt et al.'s (2007) criteria and is subsequently one of the core concepts students must master in order to be successful within their introductory programming module, although, as the results of this investigation suggest, it is a concept that students struggle with.

Variable Assignment could also be considered to be a Threshold Concept given that it has a comparatively strong correlation with students' assessment results, although weaker than Iteration, and has shown to be potentially troublesome at the beginning of the course. Indeed, all of the mental models assessed within the Programming Checkup fall into the category of Threshold Concepts. However, Variable Assignment can be viewed as one of the first Threshold Concepts students must master in order to progress in their course, whereas Iteration is one of the main Threshold Concepts students must master in order to write effective programs and be successful within their assessments. Functions and object-orientation are not covered within the Programming Checkup, but it is likely they would also be areas of difficulty for students. Equally, learning to program requires more than simply developing an understanding of how each of the concepts works, as highlighted by the five areas of difficulty with learning to program identified by Du Boulay (1986). However, the findings from this work indicate that Variable Assignment and Iteration are two key concepts which can be viewed as milestones in students' development, and which, are essential for students to develop appropriate mental models for in order to be successful within their introductory programming module.

RQ 2 Is students' perception of confidence and their previous experience positively related to their mental model development as well as their performance within their first introductory programming assessment?

Although students with previous programming experience have been found to be more likely to hold appropriate mental models of the concepts examined within the Programming Checkup, by the time students have reached the end of the first semester the differences between those with and without prior experience have decreased. Most noticeably, students with prior programming experience remain significantly more likely to hold appropriate models for Conditional Statements, Iteration and for the flow of control within a program (Parallelism). It is not unexpected for the difference between students with and without prior programming to decrease over time, as all students have the opportunity to develop their understanding of the concepts. This can be seen most significantly in the improvements students who do not have prior programming experience make with their models of Variable Assignment, although some students, particularly those with no prior programming experience, would benefit from additional support in developing their mental models surrounding Conditional Statements and Parallelism. Furthermore, it is also clear that many

students require support to develop appropriate models of Iteration – regardless of whether they have prior programming experience or not.

By having prior programming experience, students were in general initially found to be more confident, which is reflected in significantly higher self-efficacy levels (for all three factors) and also in higher levels of confidence in their answers. Students with prior programming experience also are generally less anxious about learning to program than those without prior experience, which can be seen in students' responses to how difficult they believe learning to program will be, and how much they fear learning to program. However, like the differences in students' mental model estimates, the difference between students with or without prior programming experience generally reduces by the end of the first semester, although a slight widening of the gap was observed between those with and without prior programming experience in how much they feared learning to program, and how difficult they believed learning to program to be. This may indicate that struggling students may be feeling that they are falling behind, and as such, they become more anxious.

As has been seen with mental model development, by the end of the first semester the difference between students with or without prior programming experience has begun to decrease. However, a significant difference remains within students' levels of self-efficacy relating to completing simple programming tasks (Factor 4) despite students' levels of self-efficacy for all three factors having increased between T1 and T2, regardless of whether they had previous programming experience or not. The additional experience that students have gained from programming previously is likely to be the main contributing factor to the significant difference in students' levels of self-efficacy associated with completing simple programming tasks. Although a sizeable gap between students with and without prior experience remains, there has been a substantial increase in the levels of students with no prior programming experience from T1, when they had no prior experience of completing programming tasks to draw upon, whereas students who had programmed before are able to build on their prior experiences, which helps their confidence to grow further.

Like Factor 4, students' levels of self-efficacy relating to Independence and Persistence (Factor 1) and Self-Regulation (Factor 3) also increase by T2. However, whether students have prior programming experience or not no longer results in a significant difference to their levels of self-efficacy, as these factors are generally more concerned with students'

approached to working than how confident they feels about performing programming tasks. Furthermore, a substantial difference still exists in how confident students are in their answers between those with and without prior programming experience, although it has begun to reduce by T2 as all students will be becoming more familiar with the different concepts and as such, building confidence. Nevertheless, it is evident that having experience of programming prior to starting their university course does still have a positive effect on confidence levels up until the end of the first semester. The decreasing difference between students with and without prior programming experience appears to support Wiedenbeck et al.'s (2004) claims that prior programming experience will eventually lose its predictive value.

Additionally, the levels of anxiety associated with learning to program, as measured by how much students fear learning to program and also how difficult they believe learning to program to be, were found to be substantially higher amongst students who do not have experience with programming prior to starting their course. Given that the difference in anxiety levels between students with or without prior programming experience was observed to increase at T2, it could suggest that struggling students, who have no prior programming experience, may be becoming more anxious as the module progresses to more difficult topics. This could ultimately lead to a student becoming disengaged with the module, which potentially could be mitigated through early interventions to support students in overcoming their difficulties.

Previously studying computer science has been shown to support students' initial confidence levels, although the difference between those who have or have not previously studied computer science is not as substantial as between those with or without prior programming experience. By T2, the influence of previously studying computer science does reduce somewhat, although it can still be seen to be having a positive effect on students' confidence in their answers. Furthermore, previously studying computer science has also been shown to benefit students with the development of appropriate mental models, although it may be beneficial to establish exactly what students have studied previously, rather than relying on the dichotomous option currently presented in the Programming Checkup, as there is the potential that some students may be considering IT-focused courses to be the same as computer science, when, in fact, they are not, which may also explain the differences between prior programming experience and previously studying computer science.

Given that programming is contained, either explicitly or implicitly, within all computer science curricula in the UK and Ireland, with the exception of the Irish Primary Level Curriculum (Sentance et al., 2022), it is hoped that with the resurgence of computer science in schools and the additional subject knowledge support now available for teachers (Brown et al., 2014; Sentance et al., 2022), that students will be able to build a solid foundation for their mental models of core programming concepts prior to starting university. As such, it would be useful for a future investigation to obtain as full a picture as possible about each student's prior learning, by recording whether they studied computer science at GCSE and/or A Level prior to their degree, as well as how they have gone about their learning if they consider themselves to be a self-taught programmer. This is relevant as a significant positive relationship exists between how strongly students consider themselves to be a self-taught programmer and how likely they are to hold an appropriate mental model of many of the core concepts examined within the Programming Checkup at T1. However, this relationship became non-significant for all models, apart from Iteration, by T2. Furthermore, as how strongly a student considers themselves to be a self-taught programmer has been shown to be significantly correlated with a number of confidence factors at T1 (particularly strongly with self-efficacy Factors 1 and 4), it can be concluded that students who previously taught themselves to program are likely to be initially more confident in their abilities, although the strength of the correlations does typically reduce by T2 given that all students are gaining familiarity with programming.

A similar trend is also evident amongst students who wish to pursue a career in software engineering, whereby students who state they do wish to pursue a career in software engineering are typically those who are more confident in their programming abilities. Additionally, students intending to work in software engineering are found to be more likely to hold mental models of a number of key concepts, although, the differences between those wishing to pursue a career in software engineering and those who do not, or are unsure, decreases by T2. However, it is interesting to note that a significant difference remains in students' estimates for holding an appropriate model of iteration.

Both students considering themselves to be self-taught programmers or indicating that they wish to pursue a career in software engineering speak to the motivations of students. Their motivations are likely to be intrinsic in nature given that they are indicating programming is a subject that they wish to engage in, rather than what they are forced to be doing as part of

their course. As Bergin and Reilly (2005a) previously mentioned, intrinsically motivated students were seen to perform better than students who are extrinsically motivated. This is reflected in students who wish to pursue a career in software engineering performing better in their introductory programming assessments than those who did not want to or were unsure about working in software engineering. Furthermore, how strongly a student considers themselves to be a self-taught programmer is indicative of their assessment results, as evidenced by a significant positive correlation, although it is of relatively weak strength. Of course, it is possible for a student to be intrinsically motivated and neither consider themselves to be self-taught or to wish to work in a software engineering role. However, the results presented provide support for the link between students' motivations and their performance in their introductory programming module.

There is evidence to suggest that having experience of studying a mathematics-based subject prior to starting their degree, but after leaving school, can lead to students achieving higher results within their assessments, thus supporting previous claims that mathematics experience aids students when learning to program (Bergin & Reilly, 2005b; Byrne & Lyons, 2001; Gomes et al., 2006; Wilson & Shrock, 2001). However, the Programming Checkup results revealed that having previous experience of studying a mathematics-based subject after finishing school, does not appear to significantly aid students in terms of being more likely to hold appropriate mental models, nor does it substantially aid their levels of confidence when compared to other background factors.

The limited direct impact of having previously studied a mathematics-based course on the factors examined within the Programming Checkup, raises questions as to how this previous experience is actually benefiting students. In particular, there is a lack of any significant influence on the likelihood of students holding appropriate mental models of core concepts, so it does not appear to be directly influencing students' understanding of programming concepts. Instead, studying mathematics-based subjects allows for the development of skills such as logical thinking, abstraction and attention to detail, which are precursor to students' computational thinking abilities (Curzon et al., 2019; Wing, 2008) that subsequently aid students when completing large, independent tasks such as their assessments within their introductory programming module (Gomes et al., 2006; Lister et al., 2004).

Furthermore, the effects of previously studying a mathematics-based subject on students' assessment results within their introductory programming module continue to aid students much later into their course (i.e., within their second assessment at the end of the academic year), which suggests that the problem-solving skills gained through the study of mathematics and mathematics-based subjects such as Engineering and Physics continue to be of benefit to students. However, having previous experience of programming or studying computer science exhibit a limited relationships with students' assessment results, likely due to the fact that all students are gaining experience with programming throughout the course of their introductory programming module. In sum, it is likely that previously studying a mathematics-based course does not directly support students with their understanding of programming, rather it is likely supporting them solving problems and completing tasks using programming within their assessments.

As mentioned previously, students' estimates for holding appropriate mental models of the concepts examined within the Programming Checkup are significantly correlated with their assessment results. Furthermore, it is evident that a relationship exists between the likelihood of students holding appropriate mental models for some of the key concepts examined within the Programming Checkup, and the confidence students show in their answers, as well as their levels of self-efficacy relating to completing simple programming tasks (Factor 4). Notably, Iteration demonstrated the strongest correlations with both students' average confidence in their answers, and their self-efficacy levels for Factor 4, thus giving further credence to Iteration being considered a Threshold Concept for students learning to program.

A number of variables which measure students' level of confidence in their own abilities, and their levels of anxiety surrounding learning to program, have consistently been identified as having a significant positive relationship with students' assessments results. These included how much students fear learning to program, self-efficacy Factors 1 (Independence and Persistence) and 4 (Simple Programming Tasks) and students' confidence in answering questions focusing on Variable Assignment, Conditional Statements, Iteration, and for all questions combined. Each of these variables have been confirmed through a mediation analysis to be influenced by how likely students are to be holding appropriate mental models and subsequently, influence students' performance in their assessments. An overview is, therefore, provided, of how confident a student is in applying their knowledge of programming concepts to solve simple tasks, which is key in order to progress within their

assessments. Additionally, the relationship between students' assessment results and how much they fear learning to program, suggests that their levels of anxiety surrounding programming could, as Rogerson and Scott (2010) indicate, be a barrier to their learning, as students may, for example, not feel confident to ask questions when they are struggling (Bergin & Reilly, 2005a), which, if a student is struggling to aquire appopriate mental models of key concepts, will result in the student facing greater difficulties and ultimatly impact upon their performance in assessments.

RQ 3 Can students' initial responses to the Programming Checkup be used to make predictions of students' introductory programming assessment results?

From the outset of this investigation, the aptitude test, which would ultimately become the Programming Checkup, was designed with the intention for it to be used to aid in the prediction of students' assessment results, which would enable future support interventions to be developed. Although during the research investigation the Programming Checkup was issued to students twice, in order to evaluate their progress during their first semester, it was always envisaged that students' T1 responses would be used to predict students' assessment results as identifying students who would benefit from additional support at the earliest possible opportunity, allows for interventions to be put in place in order to address misconceptions and aid students in their mental model development as they progress through their course (Romero & Ventura, 2019).

As has been discussed previously, a number of the factors examined within the Programming Checkup revealed significant relationships with the results students achieve within their first assessment as part of the introductory programming module. Given this is the first piece of assessment students undertake, it naturally ties in well with the concepts examined within the Programming Checkup, as well as providing an indication of students' performance later in the module. As such, students' results for their first assessment were chosen to be an indication of whether they would benefit from additional support.

Chapter 4 described the process through which the responses to the Programming Checkup at T1 were used to develop methods of predicting students' assessment results. This culminated in two potential approaches being explored, a regression-based approach to predicting the result a student obtains, and a binary classification approach which predicts whether a

student’s result will surpass a threshold of 50% or not. After trialling a number of different machine learning algorithms, along with different combinations of input variables, the regression model which was chosen to be evaluated using the hold-out test set utilised the Random Forest Regressor, with variables pertaining to students’ confidence levels and their estimates of holding appropriate mental models, which had been established using Bayesian Knowledge Tracing being inputted into the model. Although this was not the lowest overall RMSE obtained during tests on the training data, the fact that Random Forests are less susceptible to overfitting made it an appropriate choice for evaluation on the hold-out test set when compared to other algorithms of similar performance. This appears to have been an appropriate decision given that when trained on the whole training dataset, the model achieved an average RMSE of 0.1686, and an average of 0.1687 when evaluated on the hold-out test set. These results therefore indicate that the model does not appear to be overfitting the training set, and when scaled up, represents a predictive error of approximately 17 marks (17%).

Although 17 marks is a sizeable margin of error, it is important to note that the aim of this investigation is not to predict the exact mark a student would achieve in their assessment, rather, it is to provide an indication of whether the student is likely to require additional support. Given that making predictions about students’ performance at such an early stage can be difficult due to the wide variety of factors that can potentially influence their results (López-Zambrano et al., 2021), the margin of error is at a level to be generally acceptable to be used as a guide for identifying students who are likely to struggle in the assessment, and as such, would benefit from additional support.

The high level of granularity in the predictions made by the regression model provides educators with a more nuanced estimation as to whether a student is likely to require support. However, it does require a degree of interpretation on the part of the educator. Alternatively, the binary classification approach requires very little interpretation; students who are predicted a 1 are deemed likely to achieve a mark of 50% or greater, whereas those who are predicted a 0 are likely to achieve a mark of less than 50%.

The Random Forest classifier, with variables pertaining to students’ background factors, confidence levels and mental model estimates being utilised as input for the model, was

deemed to be the most appropriate choice to be evaluated using the hold-out test given that it had the highest AUC out of all model and input combinations being trialled.

The classification model achieved an average AUC of 0.7400 when trained on the entire training set, which indicates a drop in performance when compared to the average estimate of performance obtained through cross-validation during the model evaluation process, which produced an AUC of 0.7783. This reduction in performance despite the increased amount of training data being available, potentially indicates that the model has high levels of variance and as such, may be overfitting the training data. The model achieved an average AUC of 0.6595 when evaluated on the hold-out test set, which given the difference between this result and the average performance on the training set, does indicate a substantial level of overfitting within the model.

The results from trialling the model on the hold-out test set can be interpreted as there being approximately a 66% chance of the model correctly predicting two examples that are of different classes. Although there is still a significant margin for error, the results indicate that the classification model does perform significantly better than chance at making correct predictions and could still serve as a useful tool for the identification of struggling students.

It is possible that despite the implementation of both over-sampling and under-sampling within the training set, the distribution of the Assessment 1 results may be a significant contributing factor to the classification model overfitting the training data. Indeed, the threshold of 50% was selected due to the distribution of the results making 40% (the standard undergraduate pass level) an inappropriate choice. Further work into refining the classification model would benefit from investigating whether a different threshold would be more appropriate for classifying students who are likely to require support, while also allowing for more balanced classes, although this would likely to be both assessment specific and institution specific.

Work could also be carried out to further improve both the classification and regression models to enhance their performance by performing additional hyperparameter tuning and further refining the variables being inputted into the model with the aid of the feature importance plots obtained from the regression model, as presented in Figure 4.6, which indicated features, including students' mental model estimates for Conditional Statements

and Iteration, as well as their levels of Self-Efficacy pertaining to completing simple programming tasks (Factor 4) as consistently being important contributors to the performance of the regression model. Unfortunately, the degree of overfitting observed within classification model limits the usefulness of the associated feature importance plots presented in Figure 4.7. However, the statistical analysis of the Programming Checkup results presented within Chapter 5 is, perhaps, a more useful source of information for future refinement of the models, as feature importance plots are specific to a given model.

Ultimately, more data are needed to support any further substantial gains in performance. It is, however, interesting to note that the best performing input combinations for all of the regression models being trialled (with the exception of Regression Trees) as well as all classification models being trialled, included students' mental model estimates within their input combinations. This provides support for the use of Bayesian Knowledge Tracing as a means of assessing students' mental models of core concepts and their usefulness in predicting students' assessment results.

In sum, both the classification and regression models produced as part of this investigation have demonstrated that it is possible to make predictions of students' Assessment 1 results by using their responses to the Programming Checkup at T1. At present, the regression model has been found to be the more robust technique, with predictions being made with an acceptable margin of error for educators to gain an indication as to whether a student is likely to require additional support or not. The classification model has also been shown to be capable of making predictions of students' assessment results. However, the degree of overfitting which has been observed does indicate that the generalisability of the classification model in its current state is limited. Nevertheless, this investigation has shown that there is merit to this approach, which with further refinement and additional data, could yield a model with improved generalisability.

By demonstrating that it is possible to make predictions of the results students are likely to achieve in their first assessment based on their responses to the Programming Checkup, the ability to identify students who are likely to require additional support within their introductory programming module has been evidenced, with students who are predicted to achieve low grades likely benefiting the most from the additional support. Although this

capability has been demonstrated at a technical level, consideration must be shown as to how this could be implemented within a higher education setting.

One proposed method would be to produce an individual report for each student, which summarises their estimates of holding appropriate mental models for each concept, confidence levels, and the like, alongside the prediction of their Assessment 1 result. Such a report would encourage a constructive dialogue between educators and students; however, a future investigation should explore whether educators have a preference for the binary output of the classification algorithm, which requires very little interpretation as to whether the student requires support or not, or the numeric output from the regression model. From a personal perspective, I would prefer to utilise the output from the regression model, as it would allow for a more nuanced discussion than would be possible with a simpler binary output. Discussions around a student's abilities could also be complemented with other factors included within the report, particularly the mental model estimates, in order to determine an individualised support plan for the student.

Although the evaluation of different pedagogic interventions falls outside the scope of the current investigation, it should be acknowledged that the predictions produced by the models could be used to direct students towards targeted interventions. For example, at UCLan, the introductory programming module is complemented by an additional, optional, support lecture, where concepts are explained in more detail. Attendance of this support lecture could be made compulsory for students who have been identified as requiring additional support. Furthermore, techniques such as PRIMM (Sentance et al., 2019) could be integrated into the support lecture to aid in the development of appropriate mental models. PRIMM stands for Predict, Run, Investigate, Modify and Make (Sentance et al., 2019) and aims to address the issue of students writing programs before they are able to read and comprehend them (Parry, 2020; Sentance et al., 2019). As Sentance et al. (2019) state, "PRIMM draws on existing research in computer science education, particularly four areas of programming research: Use-Modify-Create (Lee et al., 2011), tracing and reading code before writing (Lister et al., 2004), the Abstraction Transition Taxonomy (Cutts et al., 2012) and the Block Model (Schulte, 2008)" (p. 148).

Sentance et al. (2019) describe a typical PRIMM class, which in a higher-education context would be a practical laboratory session that begins with the Predict and Run phase, where students are given a short piece of code for which they must predict and write down the expected output, in much a similar way to many of the questions within the Programming Diagnostic portion of the Programming Checkup. The predictions students make are discussed within the class and then, they are subsequently required to download (not copy) the code and run it to check their answers. During the following Investigate phase, students undertake scaffolded exercises and answer questions in order to further develop their understanding of the topic being taught before moving on to the next phase, Modify and Make where students complete structured tasks to modify the existing program (i.e., expanding its functionality or fixing bugs), and subsequently make new programs based on problem descriptions, thus giving them the chance to apply what they have learnt.

It is believed that PRIMM has not previously been applied as an intervention technique within higher education. Given the wide range of abilities students have, it would likely not be appropriate to include PRIMM as a main part of the introductory programming syllabus. However, it could form a useful part of an intervention to which students are directed to based on the predictions made from their responses to the Programming Checkup.

6.3 Limitations of this Investigation

Although the three research questions at the heart of this investigation have been successfully explored, there are nevertheless several factors that limit the conclusions that can be drawn from this work. Perhaps the most significant limiting factor is the fact that all of the participating students studied at the same institution, which consequently limits the generalisability of the results. Although many of the trends observed support those previously identified in the literature, further work involving different institutions is required in order to validate the results. A second, key limiting factor relates to the relatively small sample size with respect to the number of students who successfully completed the Programming Checkup. This hampers the performance of the predictive models and is most evident in the observed overfitting of the classification model. An expanded dataset, along with a potential exploration of differing threshold values for assessment results, as discussed previously, would help to reduce the likelihood of the classification model overfitting the training set.

Furthermore, additional data would also help to reduce the error seen in the predictions made using the regression model.

At the outset of this investigation, it was the intention to carry out data collection using the Programming Checkup at multiple institutions, but with the outbreak of the Covid-19 pandemic, it was ultimately decided to focus on students studying at UCLan. This decision ultimately impacts on the generalisability of the overall findings, as although the first administration of the Programming Checkup takes place prior to any teaching, the teaching and assessment materials are specific to the course students are studying. In particular, the choice of C++ as an introductory programming language is not common amongst universities in the UK, with Java being seen as the generally more popular language (Simon et al., 2018). Although the introductory programming module at UCLan focuses on core concepts, rather than specific language details, there is the potential for students' progression to be influenced by the language being taught. For example, whilst Python is viewed as being an easier language to learn than the likes of C++ or Java (Simon et al., 2018), there have been suggestions that the mental models that students construct are insufficient when Python is taught without explicit instruction pertaining to the notional machine (Dickson et al., 2020; Johnson et al., 2020). By only carrying out this investigation at a single institution, it is not possible to identify whether the use of C++ directly impacts on students' mental model development, nor their levels of confidence when compared to other languages. Future studies taking place across different institutions should therefore examine the influence of the programming language being on taught on students' development.

Even though data were collected from only a single institution, no two years can be considered the same due to changes in the teaching and assessment being carried out within the introductory programming module, which is most notable in the change of assessment from an exam to an assignment during the second year of data collection. Teaching on the module has also differed each year, which ranged from standard changes in teaching staff and updating to content, to classes being carried out online during the pandemic. Although these yearly changes were unavoidable, the learning outcomes of the module have remained the same, and no significant differences were identified between the results students achieved on the exam as opposed to the practical assignment that replaced it. However, any future investigations involving multiple institutions must consider the differences in the teaching and assessment of their introductory programming modules, particularly when selecting an

appropriate outcome variable to predict. An ideal solution would be to have an independent, standardised measure of a students' programming performance, although this would first need to be developed and validated. This would also require students to complete extra work, which may not be practical, hence why using students' assessment results was deemed to be an appropriate choice for the outcome variable for this investigation.
Students' participation within this investigation was voluntary, with a significant number of students choosing not to take part in the second round of data collection at T2. Although data were not collected as to why they chose not to take part, it is likely that the proximity of the second data-collection round to a number of assessment deadlines for other modules contributed to the lower uptake. Additionally, as there was no requirement to take part, students who had very low levels of confidence may have chosen not to engage with the Programming Checkup. Attempts were made to describe the Programming Checkup as an opportunity for support, rather than a test; however, the Programming Checkup would need to be made compulsory if it were to be implemented formally within the curriculum, rather than as part of a research investigation. It was also necessary to constrain what was assessed within the Programming Checkup, as making it too long to complete would have led to incomplete responses.

Being a part-time PhD student, whilst working full-time, posed restrictions on the time available for conducting research. As such, a quantitative methodology was adopted in order to maximise the amount of data being collected. However, this has meant that whilst derived from literature, the interpretation of students' misconceptions is based on my own epistemological viewpoint, as there was not enough time available to conduct interviews and walkthroughs with a sufficient number of students in order to make generalisable conclusions. As such, it would be beneficial to include interviews and walkthroughs with students as part of a future investigation in order to develop a deeper understanding of the misconceptions that students possess.

6.4 Future Work

This investigation has always been situated as a starting point for a continuing line of research. The initial focus has been on examining whether it is at all possible to make predictions about students' assessment results from the data collected using the Programming Checkup, that can be used to help identify students who are likely to require support. Although the proposed approach to identifying students has been shown to be valid, as mentioned previously, it is necessary to collect additional data from different institutions in order to improve the generalisability of the findings and allow for the performance of the models to be improved.

Following on from this investigation, the natural next step would be to explore how the Programming Checkup and the predictions made by the models can be formally integrated into an Introductory Programming module. This could include studies relating to how the data from the Programming Checkup can be used to inform teaching, which could be conducted alongside an evaluation to determine whether educators find the outputs from the regression or classification models more useful. Furthermore, future studies could focus on evaluating different forms of interventions that could be put in place to aid students who are identified as being likely to require additional support. As described previously, one potential example is the use of PRIMM within dedicated classes to aid students in developing appropriate mental models.

Future areas of research could also focus on further refinement of the Programming Checkup and of the predictive models. For example, interviews and walkthroughs could be conducted to examine the processes students go through when answering questions. This approach would be beneficial as it would allow for students to explain their thought processes in their own words, and as such, provide a deeper insight into their mental models and any misconceptions that they hold, without being driven by my own views and interpretations (Holloway, 2005). Additionally, questions within the Programming Diagnostic section of the Programming Checkup could be redeveloped, or expanded upon, to focus on individual parts of a program in an approach similar to Block Analysis (Schulte, 2008). This would, therefore, allow for more nuanced questions to be used to directly assess students' mental models of concepts embedded within larger programs. For example, questions could focus on assessing

concepts contained within specific blocks of code, which could be complemented by additional questions that would require students to read and comprehend the entire program.

It would also be useful to collect more detailed information about exactly what computer science course (if any) students have previously studied and how they have gone about learning to program if they consider themselves to be self-taught, as well as more information regarding their previous experience of mathematics. It may also be useful to include a number of questions which assess students' mathematical problem-solving abilities, similar to that of Gomes et al. (2006), to allow for a more comprehensive evaluation of how they relate to their programming abilities. It would also be beneficial for future investigations to include a third data collection point during the second semester, thus providing an opportunity to examine how students' mental models and confidence levels develop over a longer period of time. Future studies could also focus on factors such as working memory capacity, which was ultimately dropped from the Programming Checkup and estimating students' level of cognitive load, given that the placement of the mental effort questions at the end of the Programming Checkup failed to allow for any firm conclusions to be drawn.

The emphasis of this investigation was placed on exploring the different types of machine learning algorithm which could be utilised to predict students' assessment results, which culminated in Random Forests being selected for both classification and regression models, However, with the addition of an expanded dataset, future work could examine methods for how the performance of these models could be improved. This could include utilising automated feature selection techniques, such as Recursive Feature Elimination (Scikit-Learn, n.d.-aa) to identify the optimal combination of features to include in the models, as well as further hyperparameter optimisation.

As mentioned previously, the issues surrounding the class imbalance when training the classification model could be tackled by exploring alternate thresholds. Although this threshold would likely be assessment and institution specific, the process of determining what the threshold should be could constitute a significant body of work from both a machine learning and a pedagogic perspective. Furthermore, multinomial classification methods could also be explored as they would offer a middle ground between binary classification and regression, which may appeal to educators, although it would be necessary again to determine appropriate levels in students' results. It would also be beneficial to explore how newer

methods for establishing the Bayesian Knowledge Tracing parameters, such as pyBKT (Badrinath et al., 2021), could be used to automate what is currently a manual process. Furthermore, extensions to Bayesian Knowledge Tracing, such as the Forget parameter (Qiu et al., 2011), could also be explored. This would be particularly interesting in order to examine students' progression between their first and second years, where students may initially struggle if they had not practiced using their programming skills over the summer break.

There is also scope for integrating the Programming Checkup into an intelligent tutoring system, whereby it can be utilised to provide an initial assessment of students' abilities prior to them starting the course. The design of such a system and the content within it would likely constitute a significant amount of work. However, it would be beneficial in the short to medium term, to develop a web-application version of the Programming Checkup. This would, therefore, remove the dependency on survey platforms such as Qualtrics, and also allow for more direct control over the output files. This would help to streamline the process of identifying students who are likely to require support and the generation of individual reports for each student that summarise their Programming Checkup responses and include the estimates of holding appropriate mental models for each concept.

6.5 Self-Reflection and Concluding Remarks

The motivation of this investigation has always been to help struggling students overcome their difficulties with learning to program. As a lecturer in charge of the introductory programming module studied by 200+ students, it is a near-impossible task to know the issues each individual student is facing. Therefore, this investigation set out to explore whether it was possible to identify students who are likely to require support with learning to program at the beginning of their course, so that they could be provided with the guidance they need at the earliest possible opportunity.

As such, the Programming Checkup was developed to allow for an examination of how students' background factors, confidence levels and likelihood of holding appropriate mental models of core programming concepts, related to their assessment results within their introductory programming module. The data collected by the Programming Checkup were then used to develop a regression model and a classification model. These models were used

as a means of confirming that predicting students' first introductory programming assessment result from their responses to the Programming Checkup at the start of the course is a viable approach, and as such, could be utilised to identify students who would likely benefit from additional support.

The investigation provides an original contribution to knowledge through how students' responses to the factors examined within the Programming Checkup, particularly prior to the commencement of their course, relate to their performance in their introductory programming module. Furthermore, the approach of assessing students' mental models of core concepts through the use of Bayesian Knowledge Tracing also represents an original contribution to knowledge, as it is the first time it has been used within the context of an aptitude test, such as the Programming Checkup. Consequently, this is the first-time students' mental models of programming concepts have been assessed in this way.

This investigation has laid the foundations for a significant body of future work, including how the outputs from the Programming Checkup and the model predictions can be integrated into the teaching of an introductory programming module, as well as how the predictive models can be refined to further improve their performance. There is, however, a clear need for future work to focus on improving the generalisability of the findings through the inclusion of multiple institutions. Nevertheless, this investigation can be seen as a first step of many, toward a more individualised learning environment for introductory programming classes within higher education, where students' misconceptions can be addressed directly before they become engrained and hamper progress.

From a personal perspective, I am pleased with the outcomes of this investigation, although at times it has been challenging to undertake a longitudinal study such as this, whilst balancing the demands of a full-time academic role, particularly during the Covid-19 pandemic. I do believe that the findings of this investigation highlight the potential in the proposed approach for identifying students who are likely to require support with learning to program and give clear directions for future research. As an educator, I believe that all students, regardless of their previous experience, have the ability to learn to program. However, the experience I have recently gained as the module leader for introductory programming at the University of Central Lancashire, has shown me that there is a very difficult balancing act required within the teaching in order to meet the needs of students who

are completely new to programming, whilst also keeping students who have more experience engaged. Therefore, any information that can be used to better understand students and guide them towards the support they need would be beneficial to me as a module leader. It has always been my intention to utilise the findings from this investigation within the introductory programming module, both in terms of using the predictions from the Programming Checkup to identify students who are likely going to struggle, and also ensuring my teaching directly addresses the misconceptions that may be preventing students from obtaining appropriate mental models. I am, therefore, hopeful that the implementation of the findings from this work, and future investigations stemming from it, will help students to overcome the difficulties they face when learning to program.

References

Al-Radaideh, Q. A., Al-Shawakfa, E. M., & Al-Najjar, M. I. (2006). Mining student data using decision trees. *International Arab Conference on Information Technology (ACIT'2006), Yarmouk University, Jordan*.

Alexander, P. A. (2017). Reflection and reflexivity in practice versus in theory: Challenges of conceptualization, complexity, and competence. *Educational Psychologist*. https://doi.org/10.1080/00461520.2017.1350181

Amineh, R., & Asl, H. (2015). Review of constructivism and social constructivism. *Journal of Social Sciences, Literature and Languages*, *1*(1), 9–16.

Anderson, J. R. (2015). *Cognitive psychology and its implications*. Macmillan.

Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from Working Memory. *Human-Computer Interaction*. https://doi.org/10.1207/s15327051hci0102_2

AQA. (2020). *GCSE Computer Science (8525)*. https://filestore.aqa.org.uk/resources/computing/specifications/AQA-8525-SP-2020.PDF

Archer, M. S. (2012). *The reflexive imperative in late modernity*. https://doi.org/10.1017/CBO9781139108058

Avella, J. T., Kebritchi, M., Nunn, S. G., & Kanai, T. (2016). Learning analytics methods, benefits, and challenges in higher education: A systematic literature review. *Journal of Asynchronous Learning Network*. https://doi.org/10.24059/olj.v20i2.790

Awad, M., & Khanna, R. (2015). Efficient learning machines: Theories, concepts, and applications for engineers and system designers. In *Springer Nature*. https://doi.org/10.1007/978-1-4302-5990-9

Baddeley, A. (1992). Working memory. *Science*, *255*(5044). https://doi.org/10.1126/science.1736359

Badrinath, A., Wang, F., & Pardos, Z. (2021). *pyBKT: An accessible Python library of Bayesian Knowledge Tracing models*. http://arxiv.org/abs/2105.00385

Bahari, S. F. (2012). Qualitative versus quantitative research strategies: Contrasting epistemological and ontological assumptions. *Jurnal Teknologi*. https://doi.org/10.11113/jt.v52.134

Bakar, M., Mukhtar, M., & Khalid, F. (2019). The development of a visual output approach for programming via the application of cognitive load theory and constructivism. *International Journal of Advanced Computer Science and Applications*, *10*(11).

Baker, R. (2010). Data mining for education. *International Encyclopedia of Education*. https://doi.org/10.4018/978-1-59140-557-3

Baker, R. (2020). *Big data and education* (6th ed.). University of Pennsylvania.

Baker, R., Corbett, A., & Aleven, V. (2008). More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-540-69132-7-44

Baker, R., Corbett, A., Gowda, S., Wagner, A., MacLaren, B., Kauffman, L., Mitchell, A., & Giguere, S. (2010). Contextual slip and prediction of student performance after use of an intelligent tutor. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6075 LNCS*, 52–63. https://doi.org/10.1007/978-3-642-13470-8_7

Baker, R., & Siemens, G. (2014). Educational data mining and learning analytics. In *The Cambridge Handbook of the Learning Sciences* (2nd ed,). Cambridge University Press. https://doi.org/10.1017/CBO9781139519526.016

Baker, R., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *Journal of Educational Data Mining*. https://doi.org/http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240314

Bandura, A. (1977). Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review*. https://doi.org/10.1037/0033-295X.84.2.191

Bandura, A. (2006). Guide for constructing self-efficacy scales. *Self-Efficacy Beliefs of Adolescents*. https://doi.org/10.1017/CBO9781107415324.004

Baron, R. M., & Kenny, D. A. (1986). The moderator-mediator variable distinction in social psychological research: Conceptual, strategic, and statistical considerations. *Journal of Personality and Social Psychology*, *51*(6), 1173–1182. https://doi.org/10.1037/0022-3514.51.6.1173

Bartlett, F. (1933). *Remembering: A study in experimental and social psychology.* Cambridge University Press.

Batista, G., & Silva, D. F. (2009). How k-Nearest Neighbor parameters affect its performance. *Argentine Symposium on Artificial Intelligence*, *2009*, 95–106.

Batista, Gustavo, & Ronaldo. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, *6*(1), 20–29.

Beck, J. E., & Chang, K. M. (2007). Identifiability: A fundamental problem of student modeling. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-540-73078-1_17

Ben-Ari, M. (1998). Constructivism in computer science education. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*. https://doi.org/10.1145/274790.274308

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, *20.1*, 45–73. https://doi.org/10.1145/274790.274308

Bengio, Y., & Grandvalet, Y. (2004). No unbiased estimator of the variance of k-Fold Cross-Validation. *Journal of Machine Learning Research*, *5*, 1089–1105.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39(2)*, 32–36. https://doi.org/10.1145/3324888

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming: 12 years later. *ACM Inroads*. https://doi.org/10.1145/3324888

Berch, D. B., Krikorian, R., & Huha, E. M. (1998). The Corsi block-tapping task: Methodological and theoretical considerations. *Brain and Cognition*. https://doi.org/10.1006/brcg.1998.1039

Bergin, S., & Reilly, R. (2005a). The influence of motivation and comfort-level on learning to program. *PPIG 17*.

Bergin, S., & Reilly, R. (2006). Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*. https://doi.org/10.1080/08993400600997096

Bergin, S., & Reilly, R. (2005b). Programming: factors that influence success. *36th SIGCSE Technical Symposium on Computer Science Education*, 411–415.

Berglund, A., & Lister, R. (2010). Introductory programming and the didactic triangle. *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*, 35–44.

Berry, M., & Kölling, M. (2013). The design and implementation of a notional machine for teaching introductory programming. *Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSE '13)*. https://doi.org/10.1145/2532748.2532765

Berssanette, J., & De Francisco, A. (2022). Cognitive load theory in the context of teaching

and learning computer programming: A systematic literature review. *IEEE Transactions on Education*, *65*(3), 440–449. https://doi.org/10.1109/TE.2021.3127215

Bienkowski, M., Feng, M., & Means, B. (2014). Enhancing teaching and learning through educational data mining and learning analytics: An issue brief. In *Educational Improvement Through Data Mining and Analytics*.

Biggs, J. (1996). Enhancing teaching through constructive alignment. *Higher Education*, *32*(3), 347–364.

Biggs, J. (1999). What the student does: Teaching for enhanced learning. *Higher Education Research & Development*, *18*(1), 57–75.

Biggs, J. (2014). Constructive alignment in university teaching. *HERDSA Review of Higher Education*.

Bishop, C. (2006). Pattern recognition and machine learning. In *IEEE Transactions on Information Theory* (Vol. 9, Issue 4). Spinger.

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, *23*(4), 561–599.

Bonar, J., & Soloway, E. (1983). Uncovering princlples of novice programming. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. https://doi.org/10.1145/567067.567069

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*. https://doi.org/10.1207/s15327051hci0102_3

Bornat, R. (2014). *Camels and humps: A retraction*.

Bornat, R., Dehnadi, S., & Simon. (2008). Mental models, consistency and programming aptitude. *Conferences in Research and Practice in Information Technology Series*.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: Do they exist and are they useful? *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education*, 504–508. https://doi.org/10.1145/1227310.1227482

Boyer, N., Langevin, S., & Gaspar, A. (2008). Self direction & constructivism in programming education. *9th ACM SIGITE Conference on Information Technology Education* , 89–94.

Brehm, L., Guenzel, H., Hinz, O., Humpe, A., & Martius, H. (2019). Collaborative learning with COZMO to teach programming in scratch and python. *IEEE Global Engineering*

Education Conference, EDUCON, *April-2019*, 448–452. https://doi.org/10.1109/EDUCON.2019.8725037

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 Annual Meeting Of*.

Britos, P., Rey, E. J., Rodriguez, D., & Garcia-Martinez, R. (2008). Work in progress - Programming misunderstandings discovering process based on intelligent data mining tools. *Proceedings - Frontiers in Education Conference, FIE*. https://doi.org/10.1109/FIE.2008.4720499

Brown, N. C. C., Sentance, S., Crick, T., & Humphreys, S. (2014). Restart: The resurgence of computer science in UK schools. *ACM Transactions on Computing Education*, *14*(2). https://doi.org/10.1145/2602484

Bruce, & Bruce. (2017). *Practical statistics for data scientists: 50+ essential concepts using R and Python*. O'Reilly Media, Inc.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education: Research*, *3*(1), 145–160. https://www.learntechlib.org/p/111446/

Bruce, K. B. (2005). Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *ACM SIGCSE Bulletin*, *36*(4).

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 49–52. https://doi.org/10.1145/377435.377467

Cabin, R. J., & Mitchell, R. J. (2000). To Bonferroni or not to Bonferroni: when and how are the questions. *Bulletin of the Ecological Society of America*, *81*(3), 246–248. http://www.jstor.org/stable/20168454

Caceffo, R., Frank-Bolton, P., Souza, R., & Azevedo, R. (2019). Identifying and validating Java misconceptions toward a CS1 concept inventory. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, *19*, 23–29. https://doi.org/10.1145/3304221.3319771

Caceffo, R., Wolfman, S., Booth, K., & Azevedo, R. (2016). Developing a computer science concept inventory for introductory programming. *SIGCSE 2016 - Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 364–369. https://doi.org/10.1145/2839509.2844559

Cacioppo, J. T., & Petty, R. E. (1982). The need for cognition. *Journal of Personality and*

Social Psychology. https://doi.org/10.1037/0022-3514.42.1.116

Cacioppo, J. T., Petty, R. E., & Kao, C. F. (1984). The efficient assessment of need for cognition. *Journal of Personality Assessment*. https://doi.org/10.1207/s15327752jpa4803_13

Capstick, C. K., Gordon, J. D., & Salvadori, A. (1975). Predicting performance by university students in introductory computing courses. *ACM SIGCSE Bulletin*, *7*(3), 21–29.

Castro-Wunsch, K., Ahadi, A., & Petersen, A. (2017). Evaluating neural networks as a method for identifying students in need of assistance. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 111–116. https://doi.org/10.1145/3017680.3017792

Chawla, Bowyer, Hall, & Kegelmeyer. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, *16*, 321–357.

Cheah, C. (2020). Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology*, *12*(2).

Chen, T., He, T., & Benesty, M. (2018). XGBoost: Extreme gradient boosting. *R Package Version 0.71-2*, 1–4.

Cheney, P. (1980). Cognitive style and student programming ability: An investigation. *AEDS Journal*, *13*(4), 285–291.

Chiodini, L., Moreno Santos, I., Gallidabino, A., Tafliovich, A., Santos, A., & Hauswirth, M. (2021). A curated inventory of programming language misconceptions. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 380–386. https://doi.org/10.1145/3430665.3456343

Chollet. (2015). *Keras*.

Chomboon, K., Chujai, P., Teerarassammee, P., Kerdprasop, K., & Kerdprasop, N. (2015). An empirical study of distance metrics for k-nearest neighbor algorithm. *Proceedings of the 3rd International Conference on Industrial Application*, 280–285. https://doi.org/10.12792/iciae2015.051

Claesen, M., & De Moor, B. (2015). Hyperparameter search in machine learning. *MIC 2015: The XI Metaheuristics International Conference*. http://arxiv.org/abs/1502.02127

Claessen, M. H. G., Van Der Ham, I. J. M., & Van Zandvoort, M. J. E. (2015). Computerization of the standard corsi block-tapping task affects its underlying cognitive concepts: A pilot study. *Applied Neuropsychology: Adult*. https://doi.org/10.1080/23279095.2014.892488

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program.

Computer Science Education Research.

Corbett, A. T., & Anderson, J. R. (1994). Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*. https://doi.org/10.1007/BF01099821

Corney, M., Lister, R., & Teague, D. (2011). Early relational reasoning and the novice programmer : swapping as the "hello world" of relational reasoning. *Proceedings of the Thirteenth Australiasian Computing Education Conference*. http://www.computing.edu.au/acsw2011/

Corsi, P. M. (1973). *Human memory and the medial temporal region of the brain.*

Costa, E. B., Fonseca, B., Santana, M. A., de Araújo, F. F., & Rego, J. (2017). Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. *Computers in Human Behavior*, *73*, 247–256. https://doi.org/10.1016/j.chb.2017.01.047

Cox, A., Fisher, M., & O'Brien, P. (2005). Theoretical considerations on navigating codespace with spatial cognition. *PPIG 17*.

Curzon, P., Bell, T., Waite, J., & Dorling, M. (2019). Computational thinking. In *The Cambridge handbook of computing education research* (pp. 513–546).

Curzon, P., & Rix, J. (1998). Why do students take programming modules? *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, *Part F1292*, 59–63. https://doi.org/10.1145/282991.283022

Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., & Saffrey, P. (2010). Manipulating mindset to positively influence introductory programming performance. *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 431–435. https://doi.org/10.1145/1734263.1734409

Cutts, Q., Esper, S., Fecho, M., Foster, S., & Simon, B. (2012). The abstraction transition taxonomy: Developing desired learning outcomes through the lens of situated cognition. *ICER 2012 - Proceedings of the 9th Annual International Conference on International Computing Education Research*, 63–70. https://doi.org/10.1145/2361276.2361290

Danielsiek, H., Toma, L., College, B., & Vahrenhold, J. (2018). An instrument to assess self-efficacy in introductory algorithms courses. *ACM Inroads*, *9*(1), 56–65. https://doi.org/10.1145/3183510

Dehnadi, S. (2006). Testing programming aptitude. *PPIG 18*, 23–37.

Dehnadi, S., & Bornat, R. (2006). The camel has two humps (working title). *Middlesex University, UK*.

Dehnadi, S., Bornat, R., & Adams, R. (2009). Meta-analysis of the effect of consistency on success in early learning of programming. *Proceedings of 21st Annual Psychology of Programming Interest Group Conference*.

Dickson, P., Brown, N., & Becker, B. (2020). Engage against the machine: Rise of the notional machines as effective pedagogical devices. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, *7*(20), 159–165. https://doi.org/10.1145/3341525.3387404

Dietrich, D., Heller, R., & Yang, B. (2015). *Data science and big data analytics: Discovering, analyzing, visualizing and presenting data*. Wiley.

Dietterich, T. (2000). Ensemble methods in machine learning. *International Journal on Multiple Classifier Systems*, 1–15.

Doyle, E., Stamouli, I., & Huggard, M. (2005). Computer anxiety, self-efficacy, computer experience: An investigation throughout a computer science degree. *Proceedings - Frontiers in Education Conference, FIE*. https://doi.org/10.1109/fie.2005.1612246

Dreyfus, H., & Dreyfus, S. (1986). *Mind Over Machine*. Simon and Schuster.

Dreyfus, S. E. (2004). The five-stage model of adult skill acquisition. *Bulletin of Science, Technology and Society*. https://doi.org/10.1177/0270467604264992

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Du Boulay, B., O'Shea, T., & Monk, J. (1999). Black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human Computer Studies*. https://doi.org/10.1006/ijhc.1981.0309

Dudani, S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, *4*, 325–327.

Duran, R., Rybicki, J., Sorva, J., & Hellas, A. (2019). Exploring the value of student self-evaluation in introductory programming. *ICER 2019 - Proceedings of the 2019 ACM Conference on International Computing Education Research*, *10*, 121–130. https://doi.org/10.1145/3291279.3339407

Duran, R., Sorva, J., & Leite, S. (2018). Towards an analysis of program complexity from a cognitive perspective. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*. https://doi.org/10.1145/3230977.3230986

Duran, R., Zavgorodniaia, A., & Sorva, J. (2022). Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education*, *22*(4), 1–27.

Dweck, C. (2000). *Self-theories: Their role in motivation, personality, and development*.

Psychology press.

Eckert, D., Timmermann, D., & Kautz, C. (2022). Student misconceptions about loops in introductory programming courses and the influence of representations. *Proceedings - Frontiers in Education Conference, FIE*, *2022-October*. https://doi.org/10.1109/FIE56618.2022.9962545

Efron, B., & Tibshirani, R. (1986). Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, *1*(1), 54–75. https://doi.org/10.1214/ss/1177013815

El Aissaoui, O., El Alami El Madani, Y., Oughdir, L., Dakkak, A., & El allioui, Y. (2020). A multiple linear regression-based approach to predict student performance. *Advanced Intelligent Systems for Sustainable Development (AI2SD'2019)*, *1*, 9–23. https://doi.org/10.1007/978-3-030-36653-7_2

ElGamal, A. F. (2013). An educational data mining model for predicting student performance in programming course. *International Journal of Computer Applications*, *70*(17), 22–28.

Engle, R. W., Laughlin, J. E., Tuholski, S. W., & Conway, A. R. A. (1999). Working memory, short-term memory, and general fluid intelligence: A latent-variable approach. *Journal of Experimental Psychology: General*. https://doi.org/10.1037/0096-3445.128.3.309

Esposito, D., & Esposito, F. (2020). *Introducing machine learning*. Microsoft Press.

Etherington, K. (2007). Ethical research in reflexive relationships. *Qualitative Inquiry*. https://doi.org/10.1177/1077800407301175

Evans, A., Wang, Z., Liu, J., & Zheng, M. (2023). SIDE-lib: A library for detecting symptoms of python programming misconceptions. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, *1*, 159–165. https://doi.org/10.1145/3587102.3588838

Evans, J. (2003). In two minds: Dual-process accounts of reasoning. *Trends in Cognitive Sciences*. https://doi.org/10.1016/j.tics.2003.08.012

Fernandez-Medina, C., Pérez-Pérez, J., Álvarez-García, V., & Del Puerto Paule-Ruiz, M. (2013). Assistance in computer programming learning using educational data mining and learning analytics. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*. https://doi.org/10.1145/2462476.2462496

Fernández, A., García, S., Herrera, F., & Chawla, N. (2018). SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. *Journal of Artificial Intelligence Research*, *61*, 863–905. https://doi.org/10.1613/jair.1.11192

Feucht, F. C., Lunn Brownlee, J., & Schraw, G. (2017). Moving beyond reflection: reflexivity and epistemic cognition in teaching and teacher education. In *Educational Psychologist*. https://doi.org/10.1080/00461520.2017.1350180

Fincher, S., Jeuring, J., Miller, C., Donaldson, P., Du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J., & Petersen, A. (2020). Notional machines in computing education: The education of attention. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, *20*, 21–50. https://doi.org/10.1145/3437800.3439202

Flowers. (2009). Research philosophies – importance and relevance. *European Journal of Information Systems*, *3*(2), 112–126.

Ganis, G., & Kievit, R. (2015). A new set of three-dimensional shapes for investigating mental rotation processes: Validation data and stimulus set. *Journal of Open Psychology Data*. https://doi.org/10.5334/jopd.ai

García, S., Luengo, J., & Herrera, F. (2015). Data preprocessing in data mining. In *Intelligent Systems Reference Library*. Springer.

George, C. (2000). Experiences with novices: The importance of graphical representations in supporting mental models. *PPIG 12*, 33–44.

Géron, A. (2022). *Hands-on machine learning with Scikit-learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc.

Gomes, A., Carmo, L., Bigotte, E., & Mendes, A. (2006). Mathematics and programming problem solving. *3rd E-Learning Conference–Computer Science Education*, 1–5.

Gonzalez, G. (2004). Constructivism in an introduction to programming course. *Journal of Computing Sciences in Colleges*, *19.4*, 299–305.

Götschi, T., Sanders, I., & Galpin, V. (2003). Mental models of recursion. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education.* https://doi.org/10.1145/792548.612004

Griffiths, M. (1998). *Educational research for social justice: Getting off the fence*. McGraw-Hill Education (UK).

Grover, S., & Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean logic. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*. https://doi.org/10.1145/3017680.3017723

Guo, P. (2018). Non-native English speakers learning computer programming: Barriers, desires, and design opportunities. *Conference on Human Factors in Computing Systems*

- *Proceedings*, *2018-April*. https://doi.org/10.1145/3173574.3173970

Guzdial, M. (2010). Why is it so hard to learn to program. In *Making Software: What Really Works, and Why We Believe It* (pp. 111–124). O'Reilly Media.

Hambrusch, S., Hoffmann, C., Korb, J. T., Haugan, M., & Hosking, A. L. (2009). A multidisciplinary approach towards computational thinking for science majors. *ACM SIGCSE Bulletin*. https://doi.org/10.1145/1539024.1508931

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference and prediction* (2nd ed.). Springer New York. https://doi.org/10.1007/978-0-387-84858-7

Hayes, A. F. (2022). Introduction to mediation, moderation, and conditional process analysis. In *the Guilford Press* (3rd ed.). Guilford Publications.

Hill, R., & Guzdial, M. (2019). Pondering variables and direct instruction. *Communications of the ACM*, *62*(4).

Holloway, I. (2005). *Qualitative research in health care.* McGraw-Hill Education (UK).

Hsu, C., Chang, C., & Lin, C. (2008). A practical guide to support vector classification. *BJU International*, *101*(1), 1396–1400. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

Imbalanced-learn. (n.d.-a). *Pipeline — Imblearn 0.10.1 documentation*. Retrieved May 3, 2023, from https://imbalanced-learn.org/stable/references/generated/imblearn.pipeline.Pipeline.html

Imbalanced-learn. (n.d.-b). *SMOTETomek — Imblearn 0.10.1 documentation*. Retrieved May 3, 2023, from https://imbalanced-learn.org/stable/references/generated/imblearn.combine.SMOTETomek.html

Jacob, J., Jha, K., Kotak, P., & Puthran, S. (2016). Educational data mining techniques and their applications. *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015*, 1344–1348. https://doi.org/10.1109/ICGCIoT.2015.7380675

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. *ICER 2006 - Proceedings of the 2nd International Computing Education Research Workshop*. https://doi.org/10.1145/1151588.1151600

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112). Springer.

Johnson-Laird, Philip N. (2010). Mental models and human reasoning. *Proceedings of the National Academy of Sciences of the United States of America*.

https://doi.org/10.1073/pnas.1012933107

Johnson-Laird, Philip Nicholas. (1983). *Mental models: Towards a cognitive science of language, inference, and consciousness* (6th ed.). Harvard University Press.

Johnson, F., McQuistin, S., & O'Donnell, J. (2020). Analysis of student misconceptions using python as an introductory programming language. *Proceedings of the 4th Conference on Computing Education Practice (CEP '20)*. https://doi.org/10.1145/3372356.3372360

Jones, & Burnett. (2008). Spatial ability and learning to program. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*. https://doi.org/10.17011/ht/urn.200804151352

Jones, M. G., & Brader-Araje, L. (2002). The impact of constructivism on education: Language, discourse, and meaning. *American Communication Journal*.

Jovic, A., Brkic, K., & Bogunovic, N. (2015). A review of feature selection methods with applications. *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 1200–1205.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/1734263.1734299

Kahney, H. (1983). What do novice programmers know about recursion. *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/800045.801618

Kallia, M., & Sentance, S. (2019). Learning to use functions: The relationship between misconceptions and self-efficacy. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 752–758. https://doi.org/10.1145/3287324.3287377

Kamler, B., & Thomson, P. (2014). Helping doctoral students write: Pedagogies for supervision, second edition. In *Routledge*. https://doi.org/10.4324/9781315813639

Kanaparan, G., Cullen, R., & Mason, D. (2019). Effect of self-efficacy and emotional engagement on introductory programming students. *Australasian Journal of Information Systems*, *23*.

Kansanen, P., & Meri, M. (1999). Didactic relation in the teaching-studying-learning process. *Didaktik/Fachdidaktik as Science (-s) of the Teaching Profession, 2*(1), 107–116. https://doi.org/10.13140/RG.2.1.2646.4726

Karsoliya, S. (2012). Approximating number of hidden layer neurons in multiple hidden layer

BPNN architecture. *International Journal of Engineering Trends and Technology*, *3*(6), 714–717.

Kaufmann, O., & Stenseth, B. (2021). Programming in mathematics education. *International Journal of Mathematical Education in Science and Technology*, *52*(7), 1029–1048. https://doi.org/10.1080/0020739X.2020.1736349

Kessels, R., Van Zandvoort, M., Postma, A., Kappelle, L., & De Haan, E. (2000). The Corsi block-tapping task: Standardization and normative data. *Applied Neuropsychology*. https://doi.org/10.1207/S15324826AN0704_8

Kessler, C., & Anderson, J. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, *2*(2), 135–166.

Khalife, J. (2006). Threshold for the introduction of programming: Providing learners with a simple computer model. *28th International Conference on Information Technology Interfaces, 2006.*, 71–76.

Kiran, B., & Serra, J. (2017). Cost-complexity pruning of random forests. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10225 LNCS*, 222–232. https://doi.org/10.1007/978-3-319-57240-6_18

Konecki, M., & Petrlic, M. (2014). Main problems of programming novices and the right course of action. *Central European Conference on Information and Intelligent Systems*, 116.

Kotsiantis, S. (2007). Supervised machine learning: A review of classification techniques. *Emerging Artificial Intelligence Applications in Computer Engineering*, *160*(1), 3–24.

Kotsiantis, S., Kanellopoulos, D., & Pintelas, P. (2006). Data preprocessing for supervised learning. *International Journal of Computer Scince*, *1*(2), 111–117.

Kuhn, M., & Johnson, K. (2013). *Applied predictive modeling*. Spinger. https://doi.org/10.1007/978-1-4614-6849-3

Kuhn, M., & Johnson, K. (2019). *Feature engineering and selection: A practical approach for predictive models*. Chapman and Hall/CRC. https://doi.org/10.1201/9781315108230

Kunkle, W. (2010). *The impact of different teaching approaches and languages on student learning of introductory programming concepts*. Doctoral Dissertation, Drexel University.

Kunkle, W., & Allen, R. (2016). The impact of different teaching approaches and languages on student learning of introductory programming concepts. *ACM Transactions on Computinig Education*. https://doi.org/10.1145/2785807

Kurland, D., & Pea, R. (1985). Children's mental models of recursive Logo programs. *Journal of Educational Computing Research*. https://doi.org/10.2190/jv9y-5pd0-mx22-9j4y

Kwon, K. (2017). Novice programmer's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools*. https://doi.org/10.21585/ijcses.v1i4.19

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, *37*(3), 14–18. https://doi.org/10.1145/1151954.1067453

Landis, J., & Koch, G. (1977). The measurement of observer agreement for categorical data. *Biometrics*. https://doi.org/10.2307/2529310

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., & Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads*, *2*(1), 32–37. https://doi.org/10.1145/1929887.1929902

Leeds-Hurwitz, W. (2009). Social construction of reality. In *Encyclopedia of communication theory* (Vol. 2, pp. 891–894).

Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, *18*, 1–5.

Liao, S. N., Zingaro, D., Thai, K., Alvarado, C., Griswold, W. G., & Porter, L. (2019). A robust machine learning technique to predict low-performing students. *ACM Transactions on Computing Education*, *19*(3), 18. https://doi.org/10.1145/3277569

Linn, M. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*. https://doi.org/10.3102/0013189X014005014

Lishinski, A., Yadav, A., Good, J., & Enbody, R. (2016). Learning to program: Gender differences and interactive effects of students' motivation, goals, and self-efficacy on performance. *2016 ACM Conference on International Computing Education Research*, 211–220.

Lister, R, Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J., Sanders, K., & Seppälä, O. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*, 119–150.

Lister, R. (2011). Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conferences in Research and Practice in Information Technology Series*, *114*, 9–18.

López-Zambrano, J., Torralbo, J., & Romero, C. (2021). Early prediction of student learning performance through data mining: A systematic review. *Psicothema*, *33*(3), 456–465. https://doi.org/10.7334/psicothema2021.62

Lowe, T. (2019). Explaining novice programmer's struggles, in two parts: Revisiting the ITiCSE 2004 working group's study using dual process theory. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 30–36.

Lu, J., & Fletcher, G. (2009). Thinking about computational thinking categories and subject descriptors. *40th ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/1539024.1508959

Lui, A. K., Kwan, R., Poon, M., & Cheung, Y. H. Y. (2004). Saving weak programming students: Applying constructivism in a first programming course. In *ACM SIGCSE Bulletin* (Vol. 36, Issue 2). https://doi.org/10.1145/1024338.1024376

Luxton-Reilly, A. (2016). Learning to program is easy. *2016 ACM Conference on Innovation and Technology in Computer Science Education*, 284–289.

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B., Giannakos, M., Kumar, A., Ott, L., Paterson, J., Scott, M., Sheard, J., & Szabo, C. (2018). Introductory programming: A systematic literature review. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 55–106. https://doi.org/10.1145/3293881.3295779

Ma, L., Ferguson, J., Roper, M., Ross, I., & Wood, M. (2008). Using cognitive conflict and visualisation to improve mental models held by novice programmers. *ACM SIGCSE Bulletin*. https://doi.org/10.1145/1352322.1352253

MacKay, D. (1992). Bayesian interpolation. *Neural Computation*, *4*(3), 415–447. https://doi.org/10.1162/neco.1992.4.3.415

Mantovani, R., Rossi, A., Vanschoren, J., Bischl, B., & De Carvalho, A. (2015). Effectiveness of random search in SVM hyper-parameter tuning. *Proceedings of the International Joint Conference on Neural Networks*. https://doi.org/10.1109/IJCNN.2015.7280664

Margulieux, L. (2020). Spatial encoding strategy theory: the relationship between spatial skill and STEM achievement. *ACM Inroads*, *11*(1), 65–75. https://doi.org/10.1145/3381891

Mason, R., & Cooper, G. (2012). Why the bottom 10% just can't do it: Mental effort measures and Implication for Introductory programming courses. *Proceedings of the Fourteenth Australasian Computing Education Conference-*, *123*, 187–196.

Maulud, D., & Abdulazeez, A. (2020). A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, *1*(4), 140–147. https://doi.org/10.38094/jastt1457

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 125–180.

Meyer, & Land. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, *49*(3), 373–388.

Miles, J., & Shevlin, M. (2001). *Applying regression and correlation: A guide for students and researchers*. Sage.

Miller, G. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*. https://doi.org/10.1037/h0043158

Mitchell, N., Danino, N., & May, L. (2013). Motivation and manipulation: A gamification approach to influencing undergraduate attitudes in computing. *7th European Conference on Games Based Learning, ECGBL 2013*.

Moons, J., & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, *60*(1), 368–384.

Morales-Navarro, L., Giang, M., Fields, D., & Kafai, Y. (2023). Connecting beliefs, mindsets, anxiety and self-efficacy in computer science learning: an instrument for capturing secondary school students' self-beliefs. *Computer Science Education*. https://doi.org/10.1080/08993408.2023.2201548

Morrison, B., Dorn, B., & Guzdial, M. (2014). Measuring cognitive load in introductory CS: adaptation of an instrument. *ICER 2014: Proceedings of the Tenth Annual Conference on International Computing Education Research*, 131–138. https://doi.org/10.1145/2632320.2632348

Murphy, L., & Thomas, L. (2008). Dangers of a fixed mindset: Implications of self-theories research for computer science education. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 271–275. https://doi.org/10.1145/1384271.1384344

Myles, A., Feudale, R., Liu, Y., Woody, N., & Brown, S. (2004). An introduction to decision tree modeling. *Journal of Chemometrics*, *18*(6), 275–285.

https://doi.org/10.1002/cem.873

Nasiri, M., Minaei, B., & Vafaei, F. (2012). Predicting GPA and academic dismissal in LMS using educational data mining: A case mining. *3rd International Conference on ELearning and ETeaching, ICeLeT 2012*, 53–58. https://doi.org/10.1109/ICELET.2012.6333365

Norman, D. A. (1983). Some observations on mental models. In *Mental Models*.

OCR. (2015). *OCR GCSE (9-1) Computer Science Pseudocode Guide*. https://www.ocr.org.uk/Images/202654-pseudocode-guide.pdf

OCR. (2020). *GCSE Computer Science (9 - 1) - J277*. https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf

Omer, U., & Farooq, M. S. (2020). Cognitive learning analytics using assessment data and concept map: A framework-based approach for sustainability of programming courses. *Sustainability*, *12*(17).

Omer, U., Farooq, M. S., & Abid, A. (2021). Introductory programming course: Review and future implications. *PeerJ Computer Science, 7*. https://doi.org/10.7717/peerj-cs.647

Ormerod, T. (2014). Human Cognition and Programming. In *Psychology of Programming*. https://doi.org/10.1016/b978-0-12-350772-3.50009-4

Paas, F., Tuovinen, J., Tabbers, H., & Van Gerven, P. (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist*. https://doi.org/10.1207/S15326985EP3801_8

Paas, F., & Van Merriënboer, J. (1994). Instructional control of cognitive load in the training of complex cognitive tasks. *Educational Psychology Review*. https://doi.org/10.1007/BF02213420

Paas, F., Van Merriënboer, J., & Adam, J. (1994). Measurement of cognitive load in instructional research. *Perceptual and Motor Skills*.

Palaganas, E., Sanchez, M., Molintas, M., & Caricativo, R. (2017). Reflexivity in qualitative research: A journey of learning. *Qualitative Report*.

Parahoo, K. (2014). *Nursing research: principles, process and issues*. Macmillan International Higher Education.

Pardos, Z., Bergner, Y., Seaton, D., & Pritchard, D. (2013). Adapting Bayesian knowledge tracing to a massive open online course in edX. *Proceedings of the 6th International Conference on Educational Data Mining, EDM 2013*.

Pardos, Z., & Heffernan, N. (2010). Navigating the parameter space of Bayesian knowledge tracing models: Visualizations of the convergence of the expectation maximization

algorithm. *Educational Data Mining 2010 - 3rd International Conference on Educational Data Mining*.

Parkinson, J. (2022). What does space look like in CS? Mapping out the relationship between spatial skills and CS aptitude. *ICER 2022 - Proceedings of the 2022 ACM Conference on International Computing Education Research*, *2*, 46–47. https://doi.org/10.1145/3501709.3544284

Parkinson, J., & Cutts, Q. (2018). Investigating the relationship between spatial skills and computer science. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, 106–114. https://doi.org/10.1145/3230977.3230990

Parry, A. (2020). Investigating the relationship between programming and natural languages within the primm framework. *Proceedings of the 15th Workshop on Primary and Secondary Computing Education (WiPSCE '20)*. https://doi.org/10.1145/3421590.3421592

Pea, R. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*. https://doi.org/10.2190/689t-1r2a-x4w4-29j2

Pea, R., & Kurland, D. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*. https://doi.org/10.1016/0732-118X(84)90018-7

Pearson. (2020). *GCSE (9-1) Computer Science*. https://qualifications.pearson.com/content/dam/pdf/GCSE/Computer Science/2020/specification-and-sample-assessments/GCSE_L1_L2_Computer_Science_2020_Specification.pdf

Pedregosa, F., Varoquaux, G., Gramfort, A., V, M., B, T., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830.

Perkins, D., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, *2*(1), 37–55.

Piaget, J. (1973). *To understand is to invent: The future of education.*

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. *SIGCSE'12 - Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/2157136.2157182

Pintrich, P. R. (2002). Future challenges and directions for theory and research on personal

epistemology. In *Personal Epistemology: The Psychology of Beliefs about Knowledge and Knowing*.

Przybylla, M., & Romeike, R. (2014). Physical computing and its scope - Towards a constructionist computer science curriculum with physical computing. *Informatics in Education*, *13*(2), 241–254.

Qian, Y., Hambrusch, S., Yadav, A., Gretter, S., & Li, Y. (2020). Teachers' perceptions of student misconceptions in introductory programming. *Journal of Educational Computing Research*, *58*(2), 364–397.

Qian, Y., & Lehman, J. (2016). Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning*, *5*(2), 73–83. https://doi.org/10.5539/jel.v5n2p73

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. In *ACM Transactions on Computing Education*. https://doi.org/10.1145/3077618

Qiu, Y., Ql, Y., Lu, H., Pardos, Z. A., & Heffernan, N. T. (2011). Does time matter? Modeling the effect of time in Bayesian knowledge tracing. *EDM 2011 - Proceedings of the 4th International Conference on Educational Data Mining*.

Qualtrics. (n.d.). *Qualtrics JavaScript Question API Class*. Retrieved April 25, 2020, from https://api.qualtrics.com/82bd4d5c331f1-qualtrics-java-script-question-api-class

Quille, K., & Bergin, S. (2018). Programming: Predicting student success early in CS1. A re-validation and replication study. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 15–20. https://doi.org/10.1145/3197091.3197101

Quille, K., & Bergin, S. (2020). Promoting a growth mindset in CS1: Does one size fit all? A pilot study. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, *20*, 12–18. https://doi.org/10.1145/3341525.3387361

Quille, K., Culligan, N., & Bergin, S. (2017). Insights on gender differences in CS1: A multi-institutional, multi-variate study. *2017 ACM Conference on Innovation and Technology in Computer Science Education*, 263–268.

Raj, A., Ketsuriyonk, K., Patel, J., & Halverson, R. (2017). What do students feel about learning programming using both English and their native language? *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, 1–8.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research*.

https://doi.org/10.2190/C670-Y3C8-LTJ1-CT3P

Ramentol, E., Caballero, Y., Bello, R., & Herrera, F. (2012). SMOTE-RSB*: A hybrid preprocessing approach based on oversampling and undersampling for high imbalanced data-sets using SMOTE and rough sets theory. *Knowledge and Information Systems*, *33*(2), 245–265. https://doi.org/10.1007/s10115-011-0465-6

Raschka, S. (2018). *Model evaluation, model selection, and algorithm selection in machine learning*. arXiv preprint. http://arxiv.org/abs/1811.12808

Rastrollo-Guerrero, J., Gómez-Pulido, J., & Durán-Domínguez, A. (2020). Analyzing and predicting students' performance by means of machine learning: A review. *Applied Sciences (Switzerland)*, *10*(3). https://doi.org/10.3390/app10031042

Redick, T., Broadway, J., Meier, M., Kuriakose, P., Unsworth, N., Kane, M., & Engle, R. (2012). Measuring working memory capacity with automated complex span tasks. *European Journal of Psychological Assessment*. https://doi.org/10.1027/1015-5759/a000123

Reges, S. (2008). The mystery of "b := (b = false)." *ACM SIGCSE Bulletin*, *40*(1). https://doi.org/10.1145/1352322.1352147

Robins, A. (2010). Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, *20*(1), 37–71. https://doi.org/10.1080/08993401003612167

Robins, A. (2019). Novice programmers and introductory programming. In *The Cambridge handbook of computing education research* (pp. 327–376).

Rodrigo, M., Tabanao, E., Lahoz, M., & Jadud, M. (2009). Analyzing online protocols to characterize novice java programmers. *Philippine Journal of Science*.

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In *Psychology of programming* (pp. 157–174). Elsevier.

Rogerson, C., & Scott, E. (2010). The fear factor: How it affects students learning to program in a tertiary environment. *Journal of Information Technology Education: Research*. https://doi.org/10.28945/1183

Romero, C., & Ventura, S. (2010). Educational data mining: A review of the state of the art. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, *40*(6), 601–618. https://doi.org/10.1109/TSMCC.2010.2053532

Romero, C., & Ventura, S. (2019). Guest editorial: Special issue on early prediction and supporting of learning Performance. *IEEE Transactions on Learning Technologies*, *12*(2), 145–147. https://doi.org/10.1109/TLT.2019.2908106

Rumelhart, D., & Ortony, A. (2017). The representation of knowledge in memory 1. In

Schooling and the Acquisition of Knowledge (pp. 99–135). Routledge. https://doi.org/10.4324/9781315271644-10

Russell, S., & Norvig, P. (2020). *Artificial intelligence: a modern approach* (4th ed.).

Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. *ITiCSE 2007: 12th Annual Conference on Innovation and Technology in Computer Science Education - Inclusive Education in Computer Science*. https://doi.org/10.1145/1268784.1268834

Sasse, A. (1997). *Eliciting and describing users' models of computer systems*. (Doctoral dissertation, University of Birmingham).

Schneider, W., & Shiffrin, R. (1977). Controlled and automatic human information processing: I. Detection, search, and attention. *Psychological Review*. https://doi.org/10.1037/0033-295X.84.1.1

Schulte, C. (2008). Block Model: An educational model of program comprehension as a tool for a scholarly approach to teaching. *ICER 2008 - Proceedings of the ACM Workshop on International Computing Education Research*, 149–160. https://doi.org/10.1145/1404520.1404535

Scikit-Learn. (n.d.-a). *BaggingClassifier— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

Scikit-Learn. (n.d.-b). *BaggingRegressor — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingRegressor.html

Scikit-Learn. (n.d.-c). *BayesianRidge— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html

Scikit-Learn. (n.d.-d). *Cross_Val_Score — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

Scikit-Learn. (n.d.-e). *Decision trees — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/tree.html#tree

Scikit-Learn. (n.d.-f). *DecisionTreeClassifier — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier

Scikit-Learn. (n.d.-g). *DecisionTreeRegressor — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

Scikit-Learn. (n.d.-h). *ElasticNet — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html

Scikit-Learn. (n.d.-i). *GradientBoostingClassifier— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Scikit-Learn. (n.d.-j). *GradientBoostingRegressor— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

Scikit-Learn. (n.d.-k). *GridSeachCV — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Scikit-Learn. (n.d.-l). *KNeighborsClassifier— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Scikit-Learn. (n.d.-m). *KNeighborsRegressor— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

Scikit-Learn. (n.d.-n). *Lasso — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

Scikit-Learn. (n.d.-o). *LinearRegression — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Scikit-Learn. (n.d.-p). *LinearSVC — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

Scikit-Learn. (n.d.-q). *LinearSVR – scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html

Scikit-Learn. (n.d.-r). *LogisticRegression — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-

learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Scikit-Learn. (n.d.-s). *MinMaxScaler— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Scikit-Learn. (n.d.-t). *MLPClassifier – scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

Scikit-Learn. (n.d.-u). *MLPRegressor — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_network.MLPRegressor

Scikit-Learn. (n.d.-v). *OneHotEncoder – scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

Scikit-Learn. (n.d.-w). *Pipeline — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Scikit-Learn. (n.d.-x). *RandomForestClassifier — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Scikit-Learn. (n.d.-y). *RandomForestRegressor — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

Scikit-Learn. (n.d.-z). *RBF SVM parameters – scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

Scikit-Learn. (n.d.-aa). *RFE — scikit-learn 1.2.2 Documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

Scikit-Learn. (n.d.-ab). *Ridge — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

Scikit-Learn. (n.d.-ac). *RidgeClassifier — scikit-learn 1.2.2 documentation*.

Scikit-Learn. (n.d.-ad). *Support vector machines — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/svm.html

Scikit-Learn. (n.d.-ae). *SVC — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

Scikit-Learn. (n.d.-af). *SVR — scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

Scikit-Learn. (n.d.-ag). *Train_test_split— scikit-learn 1.2.2 documentation*. Retrieved May 18, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Sentance, S., Kirby, D., Quille, K., Cole, E., Crick, T., & Looker, N. (2022). Computing in school in the UK & Ireland: A comparative study. *UKICER 2022: Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education Research*. https://doi.org/10.1145/3555009.3555015

Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education*, *29*(2–3), 136–176. https://doi.org/10.1080/08993408.2019.1608781

Shabani, K., Khatib, M., & Ebadi, S. (2010). Vygotsky's zone of proximal development: Instructional implications and teachers' professional development. *English Language Teaching*, *3*(4), 237–248.

Sharmin, S., Zingaro, D., Zhang, L., & Brett, C. (2019). Impact of open-ended assignments on student self-efficacy in CS1. *CompEd 2019 - Proceedings of the ACM Conference on Global Computing Education*, *19*, 215–221. https://doi.org/10.1145/3300115.3309532

Shepard, R., & Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science*. https://doi.org/10.1126/science.171.3972.701

Shephard, K. (2019). Higher Education Peadagogy. In *The Cambridge handbook of computing education research* (pp. 276–291).

Shiffrin, R., & Schneider, W. (1977). Controlled and automatic human information processing: II. Perceptual learning, automatic attending and a general theory. *Psychological Review*. https://doi.org/10.1037/0033-295X.84.2.127

Siemens, G. (2012). Learning analytics: Envisioning a research discipline and a domain of practice. *Proceedings of the 2nd International Conference on Learning Analytics and Knowledge (LAK '12)*. https://doi.org/10.1145/2330601.2330605

Siemens, G., & Baker, R. (2012). Learning analytics and educational data mining: Towards communication and collaboration. *Learning Analytics and Educational Data Mining: Towards Communication and Collaboration.* https://doi.org/10.1145/2330601.2330661

Simon. (2011). Assignment and sequence: Why some students can't recognise a simple swap. *Proceedings - 11th Koli Calling International Conference on Computing Education Research, Koli Calling'11*, 10–15. https://doi.org/10.1145/2094131.2094134

Simon, B., Hanks, B., Murphy, L., Fitzgerald, S., McCauley, R., Thomas, L., & Zander, C. (2008). Saying isn't necessarily believing: Influencing self-theories in computing. *ICER 2008 - Proceedings of the ACM Workshop on International Computing Education Research*, 173–184. https://doi.org/10.1145/1404520.1404537

Simon, Mason, R., Crick, T., Davenport, J., & Murphy, E. (2018). Language choice in introductory programming courses at Australasian and UK universities. *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium on Computer Science Education, 2018*, 852–857. https://doi.org/10.1145/3159450.3159547

Simon, S., Fincher, S., Robins, A., Baker, B., Cutts, Q., Haden, P., Hamilton, M., Petre, M., Tolhurst, D., Box, I., de Raadt, M., Hamer, J., Lister, R., Sutton, K., & Tutty, J. (2006). Predictors of success in a first programming course. *Conferences in Research and Practice in Information Technology Series*.

Sirkiä, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. *Proceedings - 12th Koli Calling International Conference on Computing Education Research, Koli Calling 2012*. https://doi.org/10.1145/2401796.2401799

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*. https://doi.org/10.2190/2xpp-ltyh-98nq-bu77

Smith, J., DiSessa, A., & Roschelle, J. (1994). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*. https://doi.org/10.1207/s15327809jls0302_1

Sorva, J. (2010). Reflections on threshold concepts in computer programming and beyond. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling'10*, 21–30. https://doi.org/10.1145/1930464.1930467

Sorva, J. (2012). *Visual program simulation in introductory programming education.* Aalto University.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*. https://doi.org/10.1145/2483710.2483713

Strecht, P., Cruz, L., Soares, C., Mendes-Moreira, J., & Abreu, R. (2015). A comparative study of classification and regression algorithms for modelling students' academic

performance. *International Educational Data Mining Society*.

Strnad, M., Šerbec, I. N., & Rugelj, J. (2009). Programming aptitude and learning success in the introductory course on programming. *12th International Conference on Interactive Computer Aided Learning*.

Strong, S. (2000). The development of a computerized version of Vandenberg's mental rotation test and the effect of visuo-spatial working memory loading. *Dissertation Abstracts International Section A: Humanities and Social Sciences*.

Sudol, L., & Jaspan, C. (2010). *Analyzing the strength of undergraduate misconceptions about software engineering*. 31. https://doi.org/10.1145/1839594.1839601

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*. https://doi.org/10.1016/0364-0213(88)90023-7

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*. https://doi.org/10.1016/0959-4752(94)90003-5

Swets, J. (1988). Measuring the accuracy of diagnostic systems. *Science Science*, *240*(4857), 1285–1293. https://doi.org/10.1126/science.3287615

Swidan, A., Hermans, F., & Smit, M. (2018). Programming misconceptions for school students. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, 151–159. https://doi.org/10.1145/3230977.3230995

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013). A qualitative think aloud study of the early neo-Piagetian stages of reasoning in novice programmers. *Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]*.

Teague, D., & Lister, R. (2014a). Longitudinal think aloud study of a novice programmer. *Conferences in Research and Practice in Information Technology Series*.

Teague, D., & Lister, R. (2014b). Manifestations of preoperational reasoning on similar programming tasks. *Proceedings of the Sixteenth Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 148]*. http://crpit.com/Vol148.html

Teague, D., & Lister, R. (2014c). Programming: Reading, writing and reversing. *ITICSE 2014 - Proceedings of the 2014 Innovation and Technology in Computer Science Education Conference*, 285–290. https://doi.org/10.1145/2591708.2591712

Tek, F., Benli, K., & Deveci, E. (2018). Implicit theories and self-efficacy in an introductory programming course. *IEEE Transactions on Education*, *61*(3), 218–225. https://doi.org/10.1109/TE.2017.2789183

Tew, A. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner*. Doctoral dissertation, Georgia Institute of Technology.

Tibshiranit, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, *58*(1), 267–288.

Tipping, M. (2001). Sparse Bayesian learning and the relevance bector machine. *Journal of Machine Learning Research*, *1*(3), 211–244. https://doi.org/10.1162/15324430152748236

Tomasevic, N., Gvozdenovic, N., & Vranes, S. (2020). An overview and comparison of supervised data mining techniques for student exam performance prediction. *Computers & Education*, *143*, 103676. https://doi.org/10.1016/J.COMPEDU.2019.103676

Vabalas, A., Gowen, E., Poliakoff, E., & Casson, A. (2019). Machine learning algorithm validation with a limited sample size. *PLoS ONE*, *14*(11), e0224365. https://doi.org/10.1371/JOURNAL.PONE.0224365

VanDeGrift, T., Bouvier, D., Chen, T., Lewandowski, G., McCartney, R., & Simon, B. (2010). Commonsense computing (episode 6) logic is harder than pie. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 76–85.

Vandenberg, S., & Kuse, A. (1978). Mental rotations, a group test of three-dimensional spatial visualization. *Perceptual and Motor Skills*. https://doi.org/10.2466/pms.1978.47.2.599

Vandierendonck, A., Kemps, E., Fastame, M., & Szmalec, A. (2004). Working memory components of the Corsi blocks task. *British Journal of Psychology*. https://doi.org/10.1348/000712604322779460

Vapnik, V. (2000). *The nature of statistical learning theory*. Springer. https://doi.org/10.1007/978-1-4757-3264-1

Veerasamy, A. K., & Shillabeer, A. (2014). Teaching English based programming courses to English language learners/non-native speakers of English. *International Proceedings of Economics Development and Research*, *70*(17).

Ventura, P., & Ramamurthy, B. (2004). Wanted: CS1 students. no experience required. *ACM SIGCSE Bulletin*, *36*(1), 240–244.

Vygotsky, L. (1962). *Thought and language*. MIT Press.

Vygotsky, L. (1978). *Mind in society: Development of higher psychological processes*. Harvard university press.

Wakefield, J. (2013). *Bayesian and frequentist regression methods*. Springer New York.

https://doi.org/10.1007/978-1-4419-0925-1

Wakelam, E., Jefferies, A., Davey, N., & Sun, Y. (2020). The potential for student performance prediction in small cohorts with minimal available attributes. *British Journal of Educational Technology*, *51*(2), 347–370. https://doi.org/10.1111/BJET.12836

Watson, C., & Li, F. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education - ITiCSE '14*. https://doi.org/10.1145/2591708.2591749

Watson, C., Li, F. W. B., & Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. *2013 IEEE 13th International Conference on Advanced Learning Technologies*, 319–323.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*. https://doi.org/10.1007/s10956-015-9581-5

Wiedenbeck, S. (1989). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, *30*(1), 1–22. https://doi.org/10.1016/S0020-7373(89)80018-5

Wiedenbeck, S., LaBelle, D., & Kain, V. (2004). Factors affecting course outcomes in introductory programming. *Proceedings of the 16th Workshop of the Psychology of Programming Interest Group*.

Wilson, B., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *ACM SIGCSE Bulletin* , *33*(1). https://doi.org/10.1145/366413.364581

Wing, J. (2006). Computational thinking. *Communications of the ACM*, *49(3)*.

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*. https://doi.org/10.1098/rsta.2008.0118

Winslow, L. E. (1996). Programming pedagogy---a psychological overview. *ACM SIGCSE Bulletin*. https://doi.org/10.1145/234867.234872

Wong, T., & Yeh, P. (2020). Reliable accuracy estimates from k-fold cross validation. *IEEE Transactions on Knowledge and Data Engineering*, *32*(8), 1586–1594. https://doi.org/10.1109/TKDE.2019.2912815

Wright, R., Thompson, W., Ganis, G., Newcombe, N., & Kosslyn, S. (2008). Training

generalized spatial skills. *Psychonomic Bulletin and Review*. https://doi.org/10.3758/PBR.15.4.763

XGBoost. (n.d.). *XGBoost parameters*. Retrieved May 18, 2023, from https://xgboost.readthedocs.io/en/stable/parameter.html

Yadin, A. (2012). Reducing the dropout rate in an introductory programming course. *ACM Inroads*. https://doi.org/10.1145/2038876.2038894

Ye, J., Chow, J., Chen, J., & Zheng, Z. (2009). Stochastic gradient boosted distributed decision trees. *International Conference on Information and Knowledge Management, Proceedings*, 2061–2064. https://doi.org/10.1145/1645953.1646301

Yousoof, M., Sapiyan, M., & Kamaluddin, K. (2007). Measuring cognitive load – A solution to ease learning of programming. *Proceedings of World Academy of Science, Engineering and Technology*, *1*(2), 216–217.

Yudelson, M. V., Koedinger, K. R., & Gordon, G. J. (2013). Individualized bayesian knowledge tracing models. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-39112-5-18

Žanko, Ž., Mladenović, M., & Boljat, I. (2019). Misconceptions about variables at the K-12 level. *Education and Information Technologies*, *24*(2), 1251–1268. https://doi.org/10.1007/S10639-018-9824-1/TABLES/18

Žanko, Ž., Mladenović, M., & Krpan, D. (2022). Analysis of school students' misconceptions about basic programming concepts. *Journal of Computer Assisted Learning*, *38*(3), 719–730. https://doi.org/10.1111/JCAL.12643

Zhang, H., Chen, L., Qu, Y., Zhao, G., & Guo, Z. (2014). Support vector regression based on grid-search method for short-term wind power forecasting. *Journal of Applied Mathematics*, *2014*. https://doi.org/10.1155/2014/835791

Zingaro, D. (2014). Peer Instruction contributes to self-efficacy in CS1. *SIGCSE 2014 - Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 373–378. https://doi.org/10.1145/2538862.2538878

Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B: (Methodological)*, *67*(2), 301–320. https://doi.org/10.1111/J.1467-9868.2005.00503.X

Appendices

Appendix A

Final Programming Checkup Questions

Section 1: Student Details

1. Please enter your student ID number (located on the back of your UCLan card - i.e. Reg no. 20627219 / CE).
2. Please enter your year of birth.
3. Please select your gender:
   - Male
   - Female
   - Other (please specify)
4. Have you studied Computer Science (or Computing) at any level prior to beginning this course?
   - Yes
   - No
5. Have you studied any Mathematics based subjects after leaving school but prior to beginning this course (I.e., Mathematics, Engineering, Physics, etc.)?
   - Yes
   - No
6. Is English your first language?
   - Yes
   - No
7. Did you have any programming experience prior to starting university?
   - Yes
   - No

8. Would you consider yourself a self-taught programmer?
    - Strongly agree
    - Agree
    - Somewhat agree
    - Neither agree nor disagree
    - Somewhat disagree
    - Disagree
    - Strongly Disagree
9. Do you intend to work in a software engineering/programming role after graduating university?
    - Yes
    - No
    - Undecided
10. Do you intend to work in a software engineering/programming role after graduating university?
11. On a scale of 1 to 10 (with 1 being very easy and 10 being very difficult), how difficult do you expect your degree to be?
12. On a scale of 1 to 10 (with 1 being very easy and 10 being very difficult), how difficult do you expect learning to program to be?
13. On a scale for 1 to 10 (with 1 being very easy and 10 being very difficult), how difficult do you find mathematics?
14. On a scale for 1 to 10 (with 1 being no fear and 10 being very fearful), how much do you fear learning to program?

Section 2: Modified Computer Programming Self-Efficacy Questions

Students were instructed to rate their current confidence using a scale of 1 (not at all confident) to 7 (absolutely confident).

1. I could write a syntactically correct statement (line of code).
2. I could understand the language structure (of any programming language) and usage of reserved words.
3. I could write logically correct blocks of code.
4. I could write a program that displays a greetings message.
5. I could write a program that computes the average of three numbers.
6. I could use built-in functions that are available in various libraries.
7. I could write a small program given a small problem that is familiar to me.
8. I could write a reasonably sized program that can solve a problem that is only vaguely familiar to me.
9. I could debug (correct all the errors) a long and complex program that I have written, and make it work.
10. I could complete a programming project if someone shows me how to solve the problem first.
11. I could complete a programming project if I had only the language reference manual for help.
12. I could complete a programming project if I can call someone for help if I got stuck.
13. I could complete a programming project once someone else helps me get started.
14. I could complete a programming project if I had a lot of time to complete the program.
15. I could complete a programming project if I had just the built-in facility for assistance.
16. I could find ways of overcoming the problem if I got stuck at a point whilst working on a programming project.
17. I could come up with a suitable strategy for a given programming project in a short time.
18. I could manage my time efficiently if I had a pressing deadline on a programming project.
19. I could find a way to concentrate on my program, even when there are many distractions around me.
20. I could find ways of motivating myself to program, even if the problem area is of no interest to me.

Section 3: Programming Diagnostic Questions (with Example Answers)

The example answers provided for each question include the correct answer, as well as a non-exhaustive list of incorrect answers which demonstrate how the misconceptions described in Appendix B can influence students' responses. If necessary, It is also possible to combine relevant misconceptions when coding students answers.

Only answers which match the specified correct answer can be coded as correct.

1.

The variables 'A' and 'B' are initialised in the lines of code below.

```
A = 10
B = 20
```

What are the values of 'A' and 'B' after carrying out the following operation?

```
A = B
```

Example Answers	**Code**
A = 20 B = 20	Correct
A = 10 B = 10	REV
A = 20 B = 0	EX
A = 30 B = 20	AD
A = 20 B = 10	SW
A = 10 B = 20	NC

2.

Examine the following code.

What would be outputted on the screen when it is run?

```
Height = 0
Width = 0

Area = Height * Width

print Area

Height = 5
Width = 3
```

Example Answers	**Code**
0	Correct
15	PL
Area	OP
Height*Width	OP
Print	OP

3.

Examine the following code.

What would be outputted on the screen when it is run?

```
Largest = 20
Smallest = 40

print Largest
```

Example Answers	Code
20	Correct
Largest	OP
Smallest	OP
40	VN

4.

The variables 'A' and 'B' are initialised in the lines of code below.

```
A = 10
B = 20
```

What are the values of 'A' and 'B' after carrying out the following operation?

```
A = B
B = A
```

Example Answers	Code
A = 20 B = 20	Correct
A = 20 B = 10	MA
A = 10 B = 10	REV
A = 0 B = 20	EX
A = 30 B = 50	AD
A = 10 B = 20	NC
A = A B = B	OP

5.

Examine the following code.

What would be outputted on the screen when it is run?

```
for i = 0; i <= 3 {
      print i
      i = i + 1
 }
```

Example Answers	Code
0123	Correct
012	ET
i	OP
123	SP
12	SP + ET
0	NI
1	NI
01234	LT
1234	SP + LT
3	SM
4	SM + LT
2	SM + ET

6.

The variables 'big' and 'small' are initialised in the lines of code below.

```
big = 30
small = 70
```

What are the values of 'big' and 'small' after carrying out the following operation?

```
big = small
small = big
```

Example Answers	Code
big = 70 small = 70	Correct
If answer is incorrect and different to Q4	Include VN alongside any other misconceptions
big = 70 small = 30	MA
big = 30 small = 30	REV
big = 0 small = 70	EX
big = 100 small = 170	AD
big = 30 small = 70	NC
big = big small = small	OP

7.

Examine the following code.

What would be outputted on the screen when it is run?

```
for i = 0; i <= 3 {
      print i
      i = i + 1
 }
print "10"
```

Example Answers	Code
012310	Correct
01210	ET
i	OP
12310	SP
1210	SP + ET
0123410	LT
123410	SP + LT
0	NI
1	NI
30	SM
0123	SE
10101010	PL

8.

The variables 'A' and 'B' are initialised in the lines of code below.

```
A = 10
B = 20
```

What are the values of 'A' and 'B' after carrying out the following operation?

```
B = A
```

Example Answers	Code
A = 10 B = 10	Correct
A = 20 B = 20	REV
A = 0 B = 10	EX
A = 10 B = 30	AD
A = 20 B = 10	SW
A = 10 B = 20	NC
A = A B = B	OP

9.

Which of the following words starts with a 'd' AND ends with an 'e'?

Select all words this applies to.

- dance
- delicious
- soccer
- share

Example Answers	**Code**
dance	Correct
delicious	AND
share	AND
soccer	AND

10.

Which of the following words starts with a 'd' OR ends with an 'e'?

Select all words this applies to.

- dance
- delicious
- soccer
- share

Example Answers	**Code**
dance, delicious, share	Correct
soccer	OR

11.

Which of the following words starts with a 'd' AND does NOT end with an 'e'?

Select all words this applies to.

- dance
- delicious
- soccer
- share

Example Answers	**Code**
delicious	Correct
dance	AND
soccer	AND
share	NOT

12.

Which of the following words starts with a 'd' OR does NOT end with an 'e'?

Select all words this applies to.

- dance
- delicious
- soccer
- share

Example Answers	**Code**
dance, delicious, soccer	Correct
dance	OR
delicious	OR
share	NOT

13.

What is the value of 'Total' after the execution of the following code?

```
Total = 0

for i = 1; i <= 4 {
      Total = Total + i
      i = i + 1
}
Total = Total + 2
```

Example Answers	Code
12	Correct
8	ET
17	LT
11	SP
0	NI
1	NI
10	SE
Total	OP
i	OP
1234	OP

14.

Examine the following code.

What would be outputted on the screen when it is run?

```
num = 0

for i = 0; i <= 4 {
      num = i * 2
      print num
      i = i + 1
}
num = 15
```

Example Answers	**Code**
02468	Correct
0246	ET
024681012	LT
0	NI
2	NI
15	PL
0246815	PL
2468	SP
20	SM

15.

Which of these words would result in the following program outputting **False**?

Select all words this applies to.

```
if word.firstLetter == c AND word.lastLetter == r then {
      print "True"
}
else {
      print "False"
}
```

- computer
- cycle
- tale
- tear

Example Answers	Code
cycle, tale, tear	Correct
tale	AND
tear	AND
cycle	AND
computer	IF

16.

The variables 'A', 'B' and 'C' are initialised in the lines of code below.

```
A = 5
B = 3
C = 7
```

What are the values of 'A', 'B' and 'C' after carrying out the following operation?

```
A = C
B = A
C = B
```

Example Answers	Code
A = 7 B = 7 C = 7	Correct
A = 7 B = 5 C = 3	MA
A = 7 B = 7 C = 3	MA
A = 3 B = 5 C = 5	REV
A = 0 B = 0 C = 7	EX
A = 12 B = 15 C = 22	AD

A = 3 B = 5 C = 7	SW
A = 5 B = 3 C = 7	NC
A = A B = B C = C	OP

17.

Examine the following code.

What would be outputted on the screen when it is run?

```
i = 0

while i <= 5 {
      print i
      i = i + 1
}
```

Example Answers	Code
012345	Correct
01234	ET
0123456	LT
12345	SP
5	SM
i	OP

18.

The variables 'A', 'B' and 'C' are initialised in the lines of code below.

```
A = 12
B = 4
C = 6
```

What are the values of 'A', 'B' and 'C' after carrying out the following operation?

```
C = B
A = C
B = A
```

Example Answers	**Code**
A = 4 B = 4 C = 4	Correct
A = 6 B = 12 C = 4	MA
A = 4 B = 12 C = 4	MA
A = 6 B = 6 C = 12	REV
A = 0 B = 4 C = 0	EX
A = 22 B = 26 C = 10	AD

A = 6 B = 4 C = 12	SW
A = 12 B = 4 C = 6	NC
A = A B = B C = C	OP

19.

Which of these words would result in the following program outputting **False**?

Select all words this applies to.

```
if word.firstLetter == r OR NOT word.lastLetter == e then {
      print "True"
}
else {
      print "False"
}
```

- rain
- rotate
- compute
- cold
- exceed
- space

Example Answers	Code
compute, space	Correct
rain, rotate, cold, exceed	IF
rain	OR
space	OR
compute	OR
rotate	OR
cold	OR + NOT
exceed	OR + NOT
cold, exceed	NOT

20.

The variables 'smallest', 'middle' and 'largest' are initialised in the lines of code below.

```
smallest = 5
middle = 15
largest = 10
```

What are the values of 'smallest', 'middle' and 'largest' after carrying out the following operation?

```
largest = middle
smallest = largest
middle = smallest
```

Example Answers	Code
smallest = 15 middle = 15 largest = 15	Correct
if answer is incorrect and different to Q18	Include VN alongside any other misconceptions
smallest = 1 middle = 10 largest = 15	VN
smallest = 10 middle = 5 largest = 15	MA
smallest = 15 middle = 10 largest = 15	MA
smallest = 10 middle = 10 largest = 5	REV
smallest = 0 middle = 15 largest = 0	EX

smallest = 30 middle = 45 largest = 25	AD
smallest = 10 middle = 15 largest = 5	SW
smallest = 5 middle = 15 largest = 10	NC
smallest=smallest middle=middle largest=largest	OP

21.

Examine the following code.

What would be outputted on the screen when it is run?

```
i = 0

while i <= 5 {
      i = i + 1
      print i
}
```

Example Answers	**Code**
123456	Correct
12345	ET
1234567	LT
012345 (If Q17 has been answered correctly)	PL
0	NI
1	NI
23456	SP
0123456	SP
6	SM
i	OP

Note. If Q17 has been answered incorrectly, and the same answer is provided for this question, then the response should be coded as PL.

22.

Which of these words would result in the following program outputting **False**? Select all words this applies to.

```
if word.firstLetter == t AND word.lastLetter == s then {
      print "True"
}
else {
      print "False"
}
```

- tongue
- trains
- goal
- guitars

Example Answers	**Code**
tongue, goal, guitars	Correct
goal	AND
guitars	AND
tongue	AND
trains	IF

23.

Which of these words would result in the following program outputting **False**?

Select all words this applies to.

```
if word.firstLetter == t OR word.lastLetter == s then {
      print "True"
}
else {
      print "False"
}
```

- tongue
- trains
- goal
- guitars

Example Answers	Code
goal	Correct
tongue	OR
guitars	OR
trains	OR
tongue, trains, guitars	IF

24.

Which of these words would result in the following program outputting **False**?

Select all words this applies to.

```
if word.firstLetter == t AND NOT word.lastLetter == s then {
      print "True"
}
else {
      print "False"
}
```

- tongue
- trains
- goal
- guitars

Example Answers	Code
trains, goal, guitars	Correct
guitars	AND
goal	AND
trains	AND + NOT
tongue	IF

25.

Which of these words would result in the following program outputting **False**? Select all words this applies to.

```
if word.firstLetter == t OR NOT word.lastLetter == s then {
      print "True"
}
else {
      print "False"
}
```

- tongue
- trains
- goal
- guitars

Example Answers	Code
guitars	Correct
trains, goals, guitars	OR
goal, tongue	NOT
goal, trains, tongue	IF

26.

Examine the following code.

What would be outputted on the screen when it is run?

```
A = 5
B = 10

while B >= A {
      B = B - 1
      print B
 }

if A <= B {
      A = 10
      B = 10
}
```

Example Answers	Code
987654	Correct
98765	ET
9876543	LT
10	PL + NI
10976510	SP + PL
9	NI
5	SM
A	OP
B	OP

27.

The variables 'smallest', 'middle' and 'largest' are initialised in the lines of code below.

```
smallest = 8
middle = 1
largest = 11
```

What are the values of 'smallest', 'middle' and 'largest' after carrying out the following operation?

```
largest = middle
smallest = largest
middle = smallest
```

Example Answers	**Code**
smallest = 1 middle = 1 largest = 1	Correct
smallest = 11 middle = 8 largest = 1	MA
If answer is incorrect and different to Q18	Include VN alongside any other misconceptions
smallest = 1 middle = 8 largest = 11	VN
smallest = 1 middle = 11 largest = 8	REV
smallest = 0 middle = 1 largest = 0	EX

smallest = 20 middle = 21 largest = 12	AD
smallest = 11 middle = 1 largest = 8	SW
smallest = 8 middle = 1 largest = 11	NC
smallest=smallest middle=middle largest=largest	OP

28.

Examine the following code.

What would be outputted on the screen when it is run?

```
A = 55
B = 65
Val = 100
Total = A + B

if Total >= Val then {
      print "True: "
      print Total
}
else {
      print "False: "
      print Total
}
C = 40
Total = A + B + C
```

- True: 120
- True: 160
- True: Total
- False: 120
- False: 160
- False: Total

Example Answers	Code
True: 120	Correct
True: 160	PL
True: Total	OP
False: 120	IF
False: 160	IF + PL
False: Total	IF + OP

29.

Examine the following code.

What would be outputted on the screen when it is run?

```
A = 5
B = 0
for i = 0; i <= 2 {
	B = B + i
	i = i + 1
	print B
}
if B > A then {
	print "Success"
}
else {
	print "Failed"
}
```

Example Answers	**Code**
013Failed	Correct
01Failed	ET
0136Success	LT
A	OP
B	OP
Failed	OP
Success	OP
3Failed	SM
13Failed	SP
0Failed	NI
013	SE
Answer contains incorrect evaluation of If statement i.e., B = 6 Failed, B = 1 Success	IF

30.

The variables 'A', 'B' and 'C' are initialised in the lines of code below.

```
A = 3
B = 2
C = 9
```

What are the values of 'A', 'B' and 'C' after carrying out the following operation?

```
B = A
C = B
A = C
```

Example Answers	Code
A = 3 B = 3 C = 3	Correct
A = 9 B = 3 C = 2	MA
A = 2 B = 9 C = 2	REV
A = 3 B = 0 C = 0	EX
A = 17 B = 5 C = 14	AD
A = 3 B = 9 C = 2	SW

A = 3	
B = 2	NC
C = 9	
A=A	
B=B	OP
C=C	

31.

Examine the following code.

What would be outputted on the screen when it is run?

```
MAX = 7
MIN = 0
for i = 2; i <= 4 {
      MIN = MIN + i
      i = i + 1
      print MIN
}
if MIN > MAX then {
      print "Success"
}
else {
      print "Failed"
}
```

Example Answers	Code
259Success	Correct
25Failed	ET
25914Success	LT
MIN	OP
MAX	OP
Failed	OP
Success	OP
9Success	SM
259	SE
013610	SP
Answer contains incorrect evaluation of If statement i.e., MIN = 9 Failed, MIN = 1 Success	IF
If answer is incorrect and different to Q29	Include VN alongside any other misconceptions

32.

Examine the following code.

What would be outputted on the screen when it is run?

```
A = 30
B = 20
Val = 100

Total = A + B
if Total >= Val then {
      print "True "
      print Total
}
else {
      print "False: "
      print Total
}
C = 60
Total = A + B + C
```

- True: 50
- True: 110
- True: Total
- False: 50
- False: 110
- False: Total

Example Answers	Code
False: 50	Correct
True: 50	IF
True: 110	IF + PL
True: Total	IF + OP
False: 110	PL
False: Total	OP

Section 4: Mental Effort Ratings

Students were instructed to rate how much mental effort they felt was required on each of the concepts they had encountered during the Programming Checkup using a scale of 1 (very very low mental effort) to 9 (very very high mental effort). Examples of questions associated with each concept were provided.

1. Variable Assignment

   e.g.,

   ```
   B = A
   C = B
   A = B
   ```

2. Conditional Statements

   e.g.,

   ```
   if word.firstLetter == t AND word.lastLetter == s then {
         print "True"
   }
   else {
         print "False"
   }
   ```

3. Iteration

   e.g.,

   ```
   for i = 1; i <= 6 {
         num = num + i
         i = i + 1
   }
   ```

Appendix B

Misconceptions Examined by the Programming Diagnostic Questions Within the Programming Checkup

Code	Misconception	Description	Concept
AD	Addition	Right-hand value added to left (a<-a+b; b unchanged) Can be combined with EX if b becomes 0.	Variable Assignment
AND	AND	Statement including AND operator is not evaluated correctly, i.e., a statement is incorrectly evaluated to be true when only one operand is true, instead of both operands.	Conditional Statements
ET	Early Termination	Loop does not iterate enough times.	Iteration
EX	Extraction	Values extracted from right to left, right value becomes 0 (a<-b; b<-0).	Variable Assignment
IF	If Statement Evaluation	If/ if else statements evaluated incorrectly. I.e., statement is believed to be false when it should be true.	Conditional Statements
LT	Late Termination	Loop iterates too many times.	Iteration
MA	Multiple Assignment	Refers to original variable values instead of carrying changes across to subsequent lines. Applies to answers where this has occurred on at least 1 line.	Variable Assignment
NC	No Change	No change to original variable values.	Variable Assignment
NI	No Iteration	Original values returned /statement passed through once.	Iteration
NOT	NOT	Statement including NOT operator incorrectly evaluated, i.e., failure to recognise that the NOT operator inverts the expression being evaluated.	Conditional Statements
OP	Output	Misconception of program outputs - i.e., outputting a variable name instead of the value or outputting an incrementor value.	Output

OR	OR	Statement including OR operator is not evaluated correctly, i.e., a statement is incorrectly evaluated to be false despite one of the operands being true.	Conditional Statements
PL	Parallelism	Misconception related to the understanding of the flow of the control within a program. There may be the assumption that all lines of the program are continuously being monitored. i.e., not recognising the difference in output when a variable is incremented either before or after an output statement.	Parallelism
REV	Reverse	Assignment operator applied from left to right.	Variable Assignment
SE	Scope Error	Misunderstanding of how the execution of the program continues after a loop has been completed.	Iteration
SM	Summation	Views procedure as single element - i.e., does not display all iterations - just outputs final result. Must be > 1 iteration	Iteration
SP	Start Point Error	Iterative loop starts at the wrong index.	Iteration
SW	Swapping	Variables swap values.	Variable Assignment
VN	Variable Naming	Answers are affected by the name of the variable - i.e., MAX will always hold the largest value.	Variable Naming