HARNESSING PRE-COURSE APTITUDE TESTS TO PREDICT PERFORMANCE ON AN INTRODUCTORY PROGRAMMING ASSESSMENT IN HIGHER EDUCATION

Oliver Kerr

University of Lancashire, OKerr@lancashire.ac.uk

Linden J. Ball

University of Lancashire, LBall@lancashire.ac.uk

Nicky Danino

Leeds Trinity University, N.Danino@leedstrinity.ac.uk

When learning to program, students' efforts can be hampered by a variety of misconceptions pertaining to fundamental programming concepts, which can prevent them from developing appropriate mental models of these concepts. This can create a barrier to learning and subsequently impact students' confidence. As such, it is necessary to identify students who are likely to require support with learning to program at the earliest opportunity. This investigation utilises data collected from 285 first-year computer science undergraduate university students to examine the potential for using a pre-course aptitude test to predict the results of students' first introductory programming assessment, thereby providing an indication of which students would benefit most from additional support from the outset. The aptitude test, which was developed as part of this investigation, collates information on students' backgrounds and prior experiences, their perceived levels of confidence, and their likelihood of holding appropriate mental models for several core programming concepts. The data collected using the aptitude test were subsequently used to train a variety of regression and classification models to explore their potential for predicting students' assessment results. This culminated in the selection of a Random Forest Regressor and a Random Forest Classifier to be refined using Sequential Feature Selection and then finally validated against a holdout test-set to assess the generalisability of these models. The Random Forest Classifier achieved a good level of performance during training (AUC = 0.8688, F1 = 0.8353, accuracy = 0.7450). However, this was seen to reduce when evaluated on the hold-out test set (AUC = 0.7670, F1 = 0.7020, accuracy = 0.7020), demonstrating a moderate degree of overfitting, likely due to an imbalance in the classes being predicted and the limited amount of data available. In contrast, the Random Forest Regressor exhibited a generally consistent level of performance between training (RMSE = 0.1616, MAE = 0.1209) and testing (RMSE = 0.1713, MAE = 0.1396). Although there is still a sizeable margin of error, the results suggest that the Random Forest Regressor is not overfitting

the data and has the potential to be used as a guide for identifying students who would benefit from additional support. This work contributes a novel, pre-course, aptitude-testing approach that integrates students' mental models, background factors, and perceived levels of confidence to enable early identification of students who may require additional support through the prediction of introductory programming assessment results. As such, these findings provide a foundation for future work to develop targeted support interventions that can be integrated into introductory programming modules.

CCS CONCEPTS • Social and professional topics. Professional topics. Computing education

Additional Keywords and Phrases: Computer Science Education, Programming Misconceptions, Mental Models, Assessment Prediction

1 INTRODUCTION

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 [150], 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 [12] 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. Subsequently, prior research has highlighted the high failure rates in introductory programming courses, which were previously reported to be 33% on average [10, 159], although more recent studies suggest failure rates have begun to decline to between 25% and 28% [11, 144]. It is therefore essential to identify, as early as possible in the course, those students most in need of support, enabling timely interventions that directly address their misconceptions and may, in turn, contribute to further reductions in failure rates [127].

As such, numerous studies have attempted to develop methods capable of predicting students' programming abilities. This is highlighted by a search for the phrase "predicting student programming abilities" within the ACM Digital Library returning over 600,000 results dating back to the 1970s. For example, Simon et al. [143] 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. Similarly, Bergin and Reilly [12, 13] 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, a variety of data-mining approaches have been explored to predict students' programming abilities [18, 40, 67, 96, 156].

The overarching aim of the present research is to identify students who are likely to require support within their introductory programming module at the earliest possible opportunity, allowing for appropriate interventions to be put in place [102, 127]. Therefore, this investigation can be viewed as an initial step towards this goal by seeking to explore whether it is possible to make predictions on students' performance using an aptitude test (later termed the Programming

Checkup), which is issued to them before any teaching has taken place. As such, this work is guided by the following research question: *Can students' initial responses to the aptitude test be used to make predictions of their introductory programming assessment results?*

In the investigation reported below, data for the aptitude test were collected at the commencement of the students' course and therefore prior to any formal teaching. The results were utilised as input to a model that attempted to predict the result each student would achieve within their first introductory programming assessment, given that it assessed students' understandings of core programming concepts. This subsequently required an exploration of a variety of machine learning algorithms, with both regression and classification techniques being considered. These models can be built upon in future work that explores integrating the predictions into formal support mechanisms within the introductory programming module that students undertake.

The primary contribution of this work lies in demonstrating that pre-course aptitude tests could be utilised to predict students' performance in their introductory programming assessments prior to any teaching taking place. This study makes a novel contribution by combining measures of students' mental models of core programming concepts with data pertaining to students' backgrounds and perceived levels of confidence within a single pre-course instrument, enabling a more holistic examination of the factors influencing early programming performance. At this stage, the focus of the reported research is to examine whether it was at all possible to use the results from the aptitude test at the start of the course to make predictions on students' assessment results. This is seen to be a challenging task, given the inherent difficulties of making predictions at such an early stage due to the variety of factors that can influence students' performance [97]. Consequently, a degree of error can be tolerated when considering the performance of the models. As such, the contributions of this work serve as a foundation for future studies that build upon its findings to develop targeted support interventions that can be integrated into introductory programming modules.

2 LITERATURE REVIEW

2.1 The Difficulties of Learning to Program

In their initial analysis of literature relating to the problems faced by novice programmers, Konecki and Petrlic [85] indicate that the consensus amongst both students and teachers is that programming is a difficult topic to learn. Robins [123] 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 [66, 123, 124]. Rogalski and Samurçay [124] suggest 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 are abstract in nature, with no real-world counterparts [66, 83], which consequently makes understanding and applying them appropriately an area of difficulty for students [34, 66, 92, 98, 99, 123]. Indeed, students' knowledge is often limited to a surface level, "line by line" view of programs, resulting in them often struggling to identify where it is appropriate to use a particular concept, even if they have a general understanding of how it works [92, 113].

The broad nature of programming is reflected in the five key areas of difficulty that students face when learning to program, as described by Du Boulay [23]:

1. General orientation – Students must develop an understanding of what programming is, the types of problems it can solve, and the benefits of learning it.
2. Notional machine – Students can struggle to realise how a computer executes the instructions in a program because they lack an understanding of the notional machine. A notional machine represents an abstracted model of program execution that is influenced by the programming language being used rather than the specific computer hardware [8, 9, 56, 150]. Subsequently, a notional machine must be both simple enough for students to learn and comprehensive enough to support their understanding of programming concepts [56].
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 (e.g., 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 students are often overwhelmed during their first encounters with programming as they attempt to comprehend all the different issues at once [23]. As such, it is essential that students be adequately supported in the initial phases of learning to program. However, work by Berglund and Lister [15] has revealed a strong disconnect between teachers and students, as teachers tend to know little about their students' world and motivations, instead basing lessons on their own understandings of a particular concept rather than on how it should be taught effectively. Therefore, to make the teaching more accessible, educators must adapt their methods to meet students' viewpoints.

As has been mentioned previously, learning to program is a slow and complex task [66, 124], with Winslow [162] suggesting it takes approximately 10 years to turn a novice programmer into an expert. Consequently, a three-year undergraduate course can only provide a platform from which students can develop their programming abilities. Naturally, the conversion from novice to expert has several intermediate steps. Winslow [162] cites a commonly referenced scale from Dreyfus and Dreyfus [49], a republished version of which [50] has been cited over 1700 times, which breaks down the novice/expert continuum into five stages: Novice, Advanced Beginner, Competent, Proficient and Expert. It is hoped that by the end of an undergraduate degree students should be ranked between competent and proficient [162]. However, with many students being unable to produce working programs at the end of their introductory programming modules [85], the aim of having many graduates being classed as at least competent programmers seems optimistic at best. Bruce et al. [27] investigated in detail the processes that students go through while learning to program. Through interviews conducted with students, Bruce et al.'s [27] study revealed that students who have adapted a surface orientation are merely learning the answers to questions or strings of code needed to complete tasks. This consequently results in them struggling to apply the concepts they have supposedly learned in later programming tasks. In contrast, students who adopt a deeper orientation to the subject provide themselves with much firmer foundations for learning more advanced concepts in later programming classes, as they are beginning to see the meaning in what they are doing. However, Bruce et al. [27] go on to state that if it is the intention of an introductory programming module for students to understand and integrate concepts, which

consequently allows them to see the "bigger picture", then this must be explicitly incorporated into the teaching strategy. These remarks echo comments from Berglund and Lister [15], who note that teachers need to adapt their material to meet the views of the student to support them with learning to program.

2.2 Students' Interpretations of Programming Concepts

One factor that may create a barrier to students successfully learning to program was identified by Sorva [150] which is that in some disciplines, there may be concepts that are not fixed but which are instead open to interpretation. This is not the case when it comes to programming, as concepts are precisely defined and implemented within programming languages, which may be misinterpreted by novice programmers. Although these misinterpretations may appear trivial to experts, misunderstandings can be widespread amongst novice programmers and difficult to overcome [150].

Students' interpretations of fundamental programming concepts (correct or otherwise) can be described as their "mental models", which can be broadly defined as mental representations of the properties and behaviours of given concepts that are based upon an individual's prior knowledge and experience [105, 150]. Johnson-Laird, one of the pioneers of Mental Model Theory [130], suggests that humans view and interpret the world in accordance with their pre-existing mental models [71, 72]. Additionally, a set of general characteristics of mental models was developed by Norman [105], which highlights the fact that a person's mental model typically only represents a limited portion of a particular concept. Furthermore, Norman [105] states that people's ability to apply the model is often extremely limited, which in turn limits their ability to fully comprehend and utilise the concept to its fullest extent. 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 [9, 150].

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 [9, 59]. 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 [9, 150]. The process of learning to program aligns with a constructivist view of learning in which students actively construct and revise mental models, which represent the actions that fundamental programming concepts perform when processed by a computer [8, 9, 84, 154, 157]. However, misconceptions can arise when these models do not accurately reflect program behaviour. 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 [8]; that is, by presenting walkthroughs of how a particular concept operates and addressing any commonly held misconceptions directly [152]. If students' mental models are not considered 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 [162].

The literature discussed in this section has so far demonstrated the importance of appropriate mental model construction for students who are learning to program. However, during the development of these models, students can inadvertently develop misconceptions that 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 [150], who defines them as "understandings that are deficient or inadequate for many practical programming contexts" (p. 4), and by Qian and Lehman [117], who view programming misconceptions as conceptual misunderstandings. A more direct approach is taken by Chiodini et al. [35], who state that a "programming language misconception is a statement that can be disproved by reasoning entirely based on the syntax and/or semantics of a programming language" (p. 381).

Chiodini et al.'s. [35] 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. [55] take the view that misconceptions are the properties of the learner, not programs or languages, with Sorva [149] stating that generic misconceptions may lie behind more specific ones. Sorva [149] 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 the present investigation, the misconceptions being examined fall in line with the broader definitions of programming misconceptions [117, 149], 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 [20], hence the need to examine students' previous experiences. 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 learned in other subjects, as well as their interactions with the physical world [111, 115, 117, 122, 123, 147].

Both Pea [111] and Bonar and Soloway [20] 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. Similarly, Clancy [36] refers to the mismatch between some programming key words, such a "while", and natural English, noting that such linguistic transfer can be a source of confusion for students. These kinds of simple misunderstandings form a barrier to the development of accurate mental models and become compounded into stronger misconceptions that can then interfere with students' learning [34, 147]. These difficulties can be further exacerbated 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 [65, 115].

Pea [111] has proposed that students' difficulties in predicting program outputs (mentally tracing through the program and processing each instruction) may still be present after more than a year of instruction. Pea [111] further states that several misconceptions that 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 [111], the first of which is the *parallelism* bug. Pea [111] explains that whilst it can appear in a variety of contexts, fundamentally the parallelism bug relates 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 > 5)
{
    std::cout << "Hello World";
}
```

Since the variable **a** has the value 4, the condition in the if statement evaluates to false, and therefore the code inside the if block is not executed. 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 [111] 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 [111] also describes a third bug termed the *egocentrism* bug, which 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, students believe that a program can do more than what it has been explicitly told to do, which could prove particularly frustrating as they struggle to get to grips with the basic syntax and semantics of the language they are trying to learn.

Despite Pea's [111] work being carried out over thirty years ago, "little has changed" according to Simon [142] when highlighting that students still struggle with issues relating to Pea's [111] parallelism bug. Additionally, Kwon [91] provides a real-world example of the egocentrism bug where students 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 [91] 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 the form of "if m" or "if f". This demonstrates 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 [94] description of the superbug reflects Du Boulay et al.'s [24] view that teaching students how to control the machine they are using is one of the more difficult aspects of learning to program. A notional machine aids students' understanding of how programs are executed by providing an abstracted model of the computer, which is influenced by the programming language being used rather than the specific computer hardware [16, 17, 24, 150]. Sorva [149] highlights that many of the misconceptions that students exhibit is because of a lack of an appropriate mental model of the notional machine, as they do not have a clear model of how the program is executed. However, Sorva [149] goes on to state that in addition to understanding what happens when a program is run, students must also recognise that a notional machine (and therefore, a computer) only does what it is explicitly instructed to do by a programmer. Students who fail to recognise what needs to be defined within a program could be seen to be demonstrating elements of Pea's [111] superbug. Support for the use of notional machines within introductory programming modules is provided by Johnson et al. [70], who argue that teaching introductory Python without the use of a notional machine can result in students developing misconceptions and inadequate mental models [46, 70].

Although Pea's [111] work takes a more generalised view of students' mental models of program execution, an alternate approach would be to examine students' mental models of individual programming concepts. Such an approach was adopted by Dehnadi et al. [22, 42–44], who assessed students' mental models of variable assignment using multiple-choice questions. Dehnadi's [42] questions required students to predict the values of each variable after assignment operations had been carried out and ranged from simple one line assignment operations involving two variables, to more complex multi-line operations involving three variables.

By using multiple choice questions, Dehnadi [42] was able to map each potential answer to a specific mental model. Originally, Dehnadi had predicted eight different mental models. although 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 [42] 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 mental models to answer questions, with 39% of students being classed as inconsistent.
- Blank – Students refused to answer most of the questions, with 8% of students being classed as blank.

Dehnadi [42] noted that most students became consistent in their model usage after the second test. However, he did not provide any exact data and focused subsequent analyses on the results from the first test. As only 60 students were included in Dehnadi's [42] preliminary study, it is difficult to draw any statistically reliable conclusions. However, Dehnadi [42] indicated that a clear "separation of populations" (p. 29) could be observed when correlating the first test results against the official course results. Although a visual analysis of Dehnadi's [42] results does indeed reveal a separation between the consistent and inconsistent/blank groups, no statistical tests are provided to corroborate this.

Despite the lack of any in-depth statistical analysis, Dehnadi [42] documented his testing and marking process thoroughly, allowing his experiment to be replicated easily. However, Dehnadi's [42] claims to have developed a categorisation method that is "more likely to be used as a reasonable predictor of success in introductory programming" (p. 35), cannot be substantiated without statistical evidence.

It should be noted that additional claims made by Dehnadi and Bornat [43] regarding their aptitude test's ability to accurately predict students who are likely to fail their introductory programming module were later retracted by Bornat [21]. Subsequently, Dehnadi's [42] original study has been replicated several times by different researchers with mixed results. Bornat et al. [22] applied the test to 500 students across six institutions but they were unable to confirm the findings observed in their preliminary study. Consequently, Bornat et al. [22] reported that they were unable to separate the "programming goats from the non-programming sheep" (p. 8) within their expanded investigation, although they believed that their results indicate that further research into the consistency of mental models is warranted.

Additionally, a study by Caspersen et al. [32] applied Dehnadi's test to approximately 300 students and was unable to find a correlation like that originally presented by Dehnadi [42]. Caspersen et al. [32] question the viability of Dehnadi and Bornat's interpretation of their results, stating that their test instrument "does not measure what it is supposed to" (p. 210). However, a study by Strnad et al. [151] provides support for Dehnadi's aptitude test as a predictor of success with students who have had no prior programming experience.

In response to criticisms, Dehnadi [22] revised the design of his aptitude test by expanding the total number of models being examined from eight to eleven, as well as by making the judgment for consistency more explicit and repeatable. Additionally, Dehnadi et al. [44] conducted a meta-analysis using the refined test to confirm the initial findings, the results of which support claims of a relationship between consistent mental model usage and student performance, but do not suggest any explanation for it. Ultimately, Dehnadi's [42, 44] aptitude-test design presents an intriguing method for examining the mental models of students, which has had some mixed success in predicting the abilities of students with no prior programming experience and, as such, warrants further exploration.

In addition to Dehnadi's research analysing students' mental models of variable assignment, there have been several 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 [64, 75, 145]. Subsequently,

misconceptions relating to variable assignment (e.g., believing that 5 = A and A = 5) have also been prevalent in several studies [23, 100, 115, 116, 142, 145, 165, 166], thus supporting Dehnadi's [42] 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 [39, 142].

Pea and Kurland [112] state that handling conditional statements ("if" statements) is 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 without this understanding. To this end, several misconceptions have been identified that indicate students' difficulties with grasping conditional logic. These misconceptions include continuously monitoring the if statement throughout the program – identified by Pea [111] 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 [115, 146, 153].

In addition, work by Grover and Basu [64] has identified students' difficulties with grasping Boolean operators. Although most students answered questions involving the AND operator correctly, only half were able to answer questions about the OR operator correctly. A number of students exhibited a misconception whereby when both conditions are true the statement is evaluated to false, which Grover and Basu [64] explain is an embodiment of the natural-language use of "or", as students believe only one of the conditions can be true, which is equivalent to the XOR (exclusive or) condition. Consequently, Grover and Basu's [64] XOR misconception is an embodiment of Clancy's [36] linguistic transfer assumption.

Misconceptions that demonstrate potentially inadequate mental models have also been identified for two related programming concepts, namely, iteration and recursion [82]. 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 [31, 117, 146]. In some cases, students fail to recognise how the iterative loop affects the execution of the code [51, 64]. 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 = num + 1;
}
```

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 [55], or may simply not perform any iteration at all and produce an output of "0". Although 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. As a result, they may be unable to comprehend that the code inside the while loop is repeated and that the value of "num" increases by one during each iteration.

Recursion on the other hand, is a more complex concept than iteration, which Kahney [76] 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". 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 [62, 73, 90]. However, the complexities associated with recursion lend the concept to varying interpretations by students, although establishing an appropriate mental model for iteration prior to teaching recursion has been found to aid students' comprehension of the latter [82, 160].

In conducting a study into students' mental models of recursion, Kurland and Pea [90] identified several "general bugs" that were causing students difficulty. These bugs included:

- Decontextualized interpretation of commands – Students carried out "surface reading" of programs, whereby 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 [42], 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 [111] intentionality bug, whereby students 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 – Students 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 [90] described how students 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 students' predicted execution of the program and the actual result. Kurland and Pea's [90] 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 [12, 61], 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 [64].
- Mental model of embedded recursion as looping – The students in Kurland and Pea's [90] 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 [90] study was primarily focused on students' mental models of recursion, it does provide an insight into issues associated with several general mental models 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 diverse ways that mental models and misconceptions are identified and presented across many of the studies reviewed in this section, which in turn, makes comparisons between concepts more difficult.

Numerous studies have highlighted the prevalence and impact of misconceptions in learning programming, yet the need for early identification and targeted interventions remains a priority. Despite progress in understanding these challenges, much remains to be learned about the most effective strategies for addressing them. Subsequently, predicting students' performance in introductory programming modules has become a key area of research within the computer science education community [114], with a wide range of approaches being explored, as demonstrated by the examples presented

within Table 1. However, the primary focus of much of this work takes place once students are enrolled within their introductory programming courses. This investigation addresses a gap in the literature by examining factors that influence students' performance at the point of entry to their studies, before any teaching takes place, and exploring how these factors can predict outcomes in introductory programming assessments using a purpose-designed pre-course aptitude test. In doing so, the findings provide a foundation for the future development of targeted support interventions that could be implemented from the outset of a course.

Table 1: Examples of Prior Research on Early Prediction of Introductory Programming Outcomes

Study	Sample Size	Input Features	Approach Used	Reported Performance
Bergin & Reilly [14]	123 students across multiple post-high school institutions	Mathematics leaving certificate results, number of hours playing computer games and programming self-esteem	Logistic Regression	80% accuracy
Simon et al. [143]	177 students studying introductory programming at 11 post-secondary institutions	Learning approach and spatial visualisation abilities	Correlation	Marks positively correlate with deep learning approach and spatial visualisation skills (statistics not reported)
Blikstein et al. [19]	370 students from a single institution	Code snapshots from students' assignments	Hidden Markov Models	R^2 = .938 (after excluding a single outlier).
Hoq et al. [67]	772 students from a single institution	Students' programming submission metadata	Stacked ensemble regression method	RMSE = 0.18, R^2 = 0.41
Llanos et al. [96]	754 students from a single institution	Students' grades, delivery time and number of attempts	Variety of classification methods	Best performance was from Gradient Boost Classifier – F1 = 0.86
Ayalew et al. [1]	138 students from a single institution	Self-efficacy, test anxiety and goal orientation	Stepwise linear regression	R^2 = .246
Tomasevic et al. [156]	3166 students via the Open University Learning Analytics Dataset	Demographic data, engagement statistics and course performance data	Variety of classification and regression methods	Classification: F1 = 0.967 Regression: RMSE = 12.126
Costa et al. [40]	262 undergraduate studying via distance learning and 161 students studying on-campus	Demographic information and course performance	Variety of classification methods	Best performance was from Support Vector Machine – distance learning F1 = 0.92, on-campus = F1 0.83

3 INVESTIGATION METHODOLOGY

3.1 Investigation Context

This investigation was conducted with first-year computer science undergraduate students at the University of Lancashire, with data being collected between 2019 and 2021. Ethical approval for this research was obtained through the University of Lancashire Psychology and Social Sciences (PSYSOC) Ethics Committee (Reference number: PSYSOC 454).

This investigation is firmly within the realm of quantitative research, given the use of an aptitude test as a means of collecting data to answer the research questions at the heart of this work, as well as the ambition to support the development of a predictive model. As such, Bahari [2] states that Positivism and Objectivism are the appropriate epistemological and ontological orientations, respectively. The use of the aptitude test to examine factors that could influence a student's success within their introductory programming module reflects a Positivist research philosophy, which is centred around the empirical testing of hypotheses in an objective and unbiased manner [2, 57]. The Objectivist philosophy 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, investigations such as the present one allow for an increasingly accurate picture of reality to be gained [2]. This aligns with the overarching aims of this investigation, which centre around gaining an insight into the factors, particularly at the start of a course, that influence students' performance in their introductory programming module and exploring the potential for utilising these factors within a predictive model.

3.2 Aptitude Test Design

The notion of using a pre-course aptitude test as a means of identifying students who are likely to require additional support aligns with an equity-oriented approach to teaching and assessment, as it allows instructors to gain early awareness of their incoming cohort and to implement targeted interventions from the outset [54, 101, 140]. Whilst regular, low-stakes assessments administered early in a course have been shown to support learning [104], they still require students to have engaged with the course content, by which time misconceptions may already have begun to develop [9, 150]. Furthermore, intensive assessment schedules may negatively impact students' self-efficacy, particularly when the purpose of assessment is unclear or when sufficient support is not available [140]. It is also important to acknowledge that a pre-course aptitude test may similarly affect self-efficacy, particularly for students with little or no prior experience. For this reason, the aptitude test must be explicitly framed as a supportive diagnostic tool intended to understand students' current conceptions rather than a formal assessment of their learning.

Despite its criticisms, Dehnadi's [42] approach of using an aptitude test to directly examine students' mental models of variable assignment was seen as an appropriate starting point for the design of the aptitude test used in this investigation. However, to support the validity of the aptitude test, it was necessary to explicitly consider the context in which it was intended to be used when designing the aptitude test [103]. The selection of factors for inclusion in the aptitude test was guided by prior research which demonstrated their relevance to predicting students' performance in introductory programming modules, while also ensuring that they were suitable for administration prior to any teaching taking place [45, 103]. Furthermore, to support the content validity of the aptitude test, a series of pilot studies were conducted to refine its design by removing unnecessary sections that contributed to excessive completion times and by addressing issues related to question wording and formatting prior to formal data collection taking place [37, 45].

As such, the final version of the aptitude test comprised the sections described as follows.

3.2.1 Section 1: Background Factors, Motivations and Comfort Level

The mental models that students construct are influenced by what they believe to be relevant prior knowledge [9, 150]. As such, it is 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 is to determine whether they have had any prior programming experience, as 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, it is valuable to determine whether they have been teaching themselves to program, as this could be indicative of the motivation of the student. Previous work by Bergin and Reilly [13] has revealed that intrinsically motivated students, who are driven by the satisfaction of performing well in their course, show increased levels of performance as opposed to extrinsically motivated students who are primarily motivated by the rewards they can gain or to avoid punishment. Subsequently, students who have taught themselves to program are likely to be intrinsically motivated 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.

Curzon and Rix [41] 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. Students wishing to pursue a career involving programming are likely to be intrinsically motivated to perform well within their introductory programming course. However, Curzon and Rix [41] note that the proportion wanting to become a professional programmer dwindles as they progress through their course, which is likely due to the difficulties students encounter when learning to program.

There is also compelling evidence from the literature of a link between mathematics experience and programming abilities [12, 29, 61, 81, 161]. As such, it is important to identify students who have studied mathematics, or other math-based subjects such as physics or engineering (after completing secondary level education) to allow the impact of this additional experience on students' programming abilities to be examined.

Additionally, students' own estimations of their abilities, including their beliefs of how they are performing or will perform within their introductory programming module has been found to have had reasonable success in predicting students' programming abilities [12–14, 118, 161]. The term "comfort level" has been used to represent a series of variables that can be combined to produce a single continuous variable that is indicative of a student's level of anxiety surrounding a programming course [12, 161]

Studies by Bergin and Reilly [12, 13] as well as by Wilson and Shrock [161] have revealed comfort level to be a significant predictor of student performance, although comfort level questions were ultimately not included within the final predictive model developed by Bergin and Reilly [12, 13, 118]. However, several of the 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 the present aptitude test, which would be administered prior to any teaching. Nevertheless, students' initial beliefs about the difficulty of learning to program may be indicative of their subsequent programming performance, as fear of programming has been shown to form a "very real, almost physical barrier that causes intense emotions, a loss of confidence," ultimately resulting in a block in students' learning [125].

Additionally, it is important to consider factors such as gender, as prior studies have observed female students outperforming male students within an introductory programming module [95, 119]. Whether English is a student's first language should also be considered, due to the additional difficulties that non-native English-speaking students face when attempting to learn to program [65, 120].

3.2.2 Section 2: Modified Self-Efficacy Scale

Prior research has revealed significant relationships between students' levels of self-efficacy, which is a representation of their own judgments of their capabilities [6, 7], and factors including introductory programming course performance, levels of emotional engagement and occurrence of misconceptions [78, 80, 155].

Self-efficacy is not a personality trait that can measured by generic tests [6, 7, 121]. As such, it is essential that a self-efficacy scale that is specific to introductory programming be used when evaluating students' programming abilities. One such scale is Ramalingam and Wiedenbeck's [121] "Computer Programming Self-Efficacy Scale", which has been employed in numerous studies ranging from investigations of students' computer anxieties [48] to evaluating alternative pedagogic approaches and assessment styles for introductory programming courses [139, 158, 167]. Ramalingam and Wiedenbeck's 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" [121], with answers being recorded using a 7-point scale ranging from 1 (not at all confident) to 7 (absolutely confident).

Given its widespread use [167], Ramalingam and Wiedenbeck's [121] scale provides a firm foundation for assessing students' self-efficacy levels related to programming in general. However, Bandura [7] states, one measure does not fit all scenarios, meaning several small modifications are required to make Ramalingam and Wiedenbeck's [121] scale suitable for use within the aptitude test. For example, Ramalingam and Wiedenbeck [121] partitioned their original scale questions into four factors: Factor 1 – Independence and persistence; Factor 2 – Complex Programming Tasks; Factor 3 – Self-Regulation; and Factor 4 – Simple Programming Tasks. However, as the aptitude test is focused towards assessing students' fundamental programming skills, it was decided to omit the Complex Programming Task (Factor 2) questions, as many of the questions bared no relevance to the objectives of the aptitude test. Additionally, several of the Factor 2 questions pertain to the identification, creation, and use of classes, which falls outside the scope of the current investigation. The remaining questions were presented in the same order as Ramalingam and Wiedenbeck's [121] original scale, with one minor alteration being made through the removal of any specific references to C++ and replaced with "any programming language", to make the aptitude test language independent. Additionally, Ramalingam and Wiedenbeck [121] provided instructions to students when completing the scale, which have been replicated in this study, without referring to a specific programming language.

3.2.3 Section 3: Mental Model Diagnostic Questions

This section of the aptitude test posed a series of questions designed to identify any misconceptions held by students pertaining to a number of key programming concepts. This subsequently allows for an estimate of how likely a student is to be holding appropriate mental models for a given concept by evaluating their responses to relevant questions using Bayesian Knowledge Tracing (BKT). BKT originates from Corbett and Anderson's work with intelligent tutoring systems [38]. Within BKT, students' knowledge is a latent variable, and it is assumed that skills (knowledge components) can be represented in a binary fashion by being either mastered (learned) or not (not learned) [38, 164]. This approach to modelling students' knowledge can be represented as a Hidden Markov Model, as seen in Figure 1 [3].

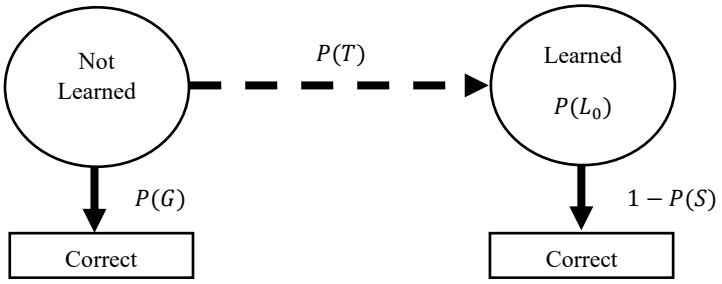

Figure 1: Bayesian Knowledge Tracing Hidden Markov Model from "Big Data and Education" by R. Baker, 2020, University of Pennsylvania

BKT utilises four parameters to model students' knowledge [3, 38]:

- $P(L0)$ – 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 1, where they answered correctly, or Equation 2, where they answered incorrectly, and then accounting for the possibility that the student has learned the skill while completing the task using Equation 3) [4].

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

$$P(L_{n-1}|Incorrect_n) = \frac{P(L_{n-1})*(P(S))}{P(L_{n-1}) * P(S) + (1-P(L_{n-1})) * (1-P(G))} \quad (2)$$

$$P(L_n|Action_n) = P(L_{n-1}|Action_n) + \big((1 - P(L_{n-1}|Action_n)\big) * P(T)) \quad (3)$$

Within the context of this investigation, $P(L)$ represents the likelihood of a student holding an appropriate mental model of a given concept, which is established by examining each relevant concept and evaluating whether the student has answered it correctly, or whether they have displayed a misconception associated with it.

Whilst BKT was not originally designed for use with aptitude tests, Pardos et al. [110] have demonstrated that it can be adapted for use in contexts other than Intelligent Tutoring Systems. Subsequently, the use of BKT is explored in this study as it focuses on individual knowledge components, which in this context represent students' mental models of specific programming concepts. This concept-level focus aligns with the overarching intention of the work, whereby the aptitude test provides an initial evaluation of students, which can inform future support interventions designed to offer targeted assistance as they progress through their introductory programming module.

The main concepts covered within the questions included: variable assignment; conditional statements; and iteration. However, additional areas that may cause misconceptions were also examined alongside the main topics, including program flow (Pea's [111] parallelism misconception), output statements, and whether the names of variables affect what they can hold. As such, the concepts examined within the questions broadly align with the notion of threshold concepts, as they are typically associated with key areas of difficulty and, once understood, can transform how students understand a subject and support further conceptual development [25, 77, 79, 129, 148].

To ensure that the aptitude test was language-independent, pseudocode based on the OCR GCSE Pseudocode Guidelines [106] was used for all programming-based questions, as the test places more emphasis on students' abilities to deduce answers logically rather than their understanding of the syntax of a particular language. For example, Figures 2 and 3 demonstrate variable assignment-based questions; including both single assignment operations (Figure 2) and multiple assignment operations (Figure 3), which are derived from Dehnadi et al.'s work [22, 42–44]. However, unlike their original aptitude test, students were not presented with a list of answers to choose from and were instead allowed to input their own values for each of the variables after the statements had been executed. Whilst open-ended questions may increase the risk of erroneous responses, they prevent students' answers from being influenced by the provided options. Furthermore, they also allow for the identification of additional misconceptions that may not have been previously considered, based on patterns in students' responses.

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 2: Example of a Single Assignment Operation 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
```

Figure 3: Example of a Multiple Assignment Operation Question

Misunderstandings surrounding conditional statements can cause students significant difficulties when writing programs of their own [112]. As such, questions which directly evaluated students’ comprehension of the AND, OR and NOT operators were adapted from Grover and Basu’s [64] assessment design, as shown in Figure 4.

Which of the following words starts with a ‘d’ OR ends with an ‘e’?
Select all words this applies to.

☐ dance

☐ delicious

☐ soccer

☐ share

Figure 4: Example of a Boolean Operator Question

Students were required to select all appropriate answers for the question to be considered correct. This approach was subsequently expanded to include assessment of students’ ability to correctly trace the execution of if statements, as shown in Figure 5. Questions relating to iteration followed a similar vein to the variable assignment-based questions, as students were required to examine the code and answer an open-ended question, as can be seen in Figure 6.

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

☐ goals

☐ guitars

Figure 5: Example of an If Statement Based Question

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
}
```

Figure 6: Example of an Iteration 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
```

Figure 7: Example of Variable Assignment with Inferred Meaning Variables Question

As was noted previously, the potential for students to be holding misconceptions that are not confined to the concepts of variable assignment, conditional statements or iteration were also examined through their responses to questions. For example, output statements, which in this scenario are indicated with the keyword “`print`”, occur in questions across all topics, consequently, students’ answers may contain misconceptions associated with output statements, such as outputting the name of a variable rather than its value.

Students’ understanding of the flow of control within a program was examined within both conditional statement and iteration-based questions to determine whether students correctly understood the order in which statements are executed. For example, a subsequent question to that shown in Figure 6 is the same but the statement “`i = i + 1`” is placed above “`print i`” instead of below it, as shown in Figure 6. 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.

Additionally, the potential exists for students to mistakenly believe, through a misconception which is similar in nature to Pea’s [111] intentionality bug, that a variable’s name affects the values it can hold. For example, a student could mistakenly believe that a variable called “largest” will always hold the largest value [75, 117], as shown in Figure 7. 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 to establish whether students hold this misconception.

Students’ perceived level of confidence has previously been seen to significantly influence their performance within introductory programming courses [125]. As such, students were asked to rate how confident they were that they had answered each question correctly using a 0 – 100 scale, subsequently allowing for a direct evaluation of their confidence in applying each concept, as well as across the aptitude test. Finally, students were asked to rate how much mental effort they felt was required to answer questions on each of the three main concepts being examined using a 9-point Likert scale ranging from very, very low mental effort (1) to very, very high mental effort (9). Previous research has shown this to be an exceptionally reliable measurement of mental effort and, in turn, cognitive load [109]. As such, it was hoped that the

mental effort ratings could potentially provide an indication of students who may be susceptible to experiencing cognitive overload [163].

3.3 Approach to Data Collection

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 the paper-based approach taken by Dehnadi [42]. It is important to stress that participation in this study was optional, and all responses were anonymised in accordance with the ethical approval obtained for this research. It should also be noted that students' university ID numbers were recorded to allow for their aptitude test results to be compared to their introductory programming module grades. Students were incentivised to take part by being entered into a prize draw for one of five £10 Amazon gift cards, as well as by being able to receive feedback on their answers. Additionally, the aptitude test was termed the "Programming Checkup" to avoid any negative connotations with the word 'test', although it was made clear to students that their participation would have no bearing on the evaluation of their performance in their introductory programming module.

Commencing in September 2019, and extending across a period of three years, the Programming Checkup was presented to all first-year computer science students at the University of Lancashire during the first week of their degree. It should be noted that 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, had 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 Academic Year 2019-2020 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. As a result, 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. A full account of the Programming Checkup can be found within the Open Science Framework (OFS) repository for this article:
https://osf.io/exyh9/overview?view_only=e6a36185548646ec8210d1a21e1caab3

4 PREDICTIVE MODEL DEVELOPMENT

4.1 Model Objectives

Previous research into developing predictive models of programming performance has 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 [13] were able to produce a regression model that 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% [33] and others at 40% [94, 156].

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, apart from functions, are examined within the Programming Checkup.

Having students complete the Programming Checkup at the earliest opportunity allows interventions to be implemented in the initial stages of a course, thereby directly supporting students' construction of appropriate mental models before any misconceptions become entrenched [107, 108, 162]. 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 together with students' Assessment 1 results. Of the available data, 70% was randomly selected to be used to train and develop the model and the remaining 30% was reserved as a testing holdout set [69]. 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 [128]. Additionally, using a random train/test split limits the impact of cohort-specific characteristics, particularly given that data collection took place during the Covid-19 pandemic.

The total dataset contained 285 responses after removal of students who did not complete Assessment 1 or who skipped 25% or more of the mental model diagnostic questions within the Programming Checkup, therefore resulting in training dataset of 200 responses and a testing dataset of 85 responses. Anonymised versions of these datasets can be found using the aforementioned OFS repository link.

Due to a change in assessment policy beyond the control of this investigation, Assessment 1 was altered after the first year of data collection from an in-person, proctored written exam to an unproctored practical programming assignment, which students completed independently in their own time. This practical assignment was designed to assess the same learning outcomes as the written exam, which tests students' abilities to construct a structured 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 pre-course responses to the Programming Checkup allows teaching staff to provide dedicated support to aid students in constructing appropriate mental models before moving on to more complex topics, which are typically covered during the second semester [1, 12, 14, 40, 67, 114, 126, 156]. The sections below detail the steps taken during the development and validation of the predictive model.

4.2 Data Pre-Processing

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 describes how the raw output from the Programming Checkup needed to be prepared to allow both classification and regression methods to make predictions.

To minimise the potential for human error, a Python script was used to extract data from the raw output generated by the online survey platform (Qualtrics). Much of the data could be extracted directly. However, some aspects of the dataset required additional processing before these data 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.

Additionally, students' answers to the Programming Diagnostic questions needed to be "coded" to indicate what misconceptions they were exhibiting based on their answers. This approach was based upon the work conducted by Dehnadi [42], in which a set of pre-defined answers for each question was developed using the literature discussed previously, to capture one or more misconceptions arising for each question (see Appendix A). Where a student's response matched one of the pre-defined answers, the response 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 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 to be coded, as both require comparisons to be made between two questions, as explained previously. 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 BKT to be used to evaluate whether students held appropriate mental models, by recording if a student had demonstrated a misconception associated with the mental model(s) being examined within each question, as detailed in Appendix A. 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 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 (SK) 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 for BKT to evaluate whether a student held an appropriate mental model or not, as well as for training the initial hyperparameters for each mental model.

Prior to the development of an automated marking script, the responses to the first data collection in September 2019 were initially manually coded. The coding was led by the lead author, who has a background in computer science education research and was involved in teaching the students on their introductory programming module. The coding scheme was based on the previously discussed literature and was then reviewed and agreed upon by the research team, whose expertise collectively spanned computer science education research and cognitive psychology. To verify the coding technique, an external marker from the computer science teaching faculty, who had regular contact with first-year students but no prior involvement in the development of the Programming Checkup, was asked to independently mark 10% of the responses. The external marker was provided with a list of examples of correct answers and a non-exhaustive list of incorrect answers, which indicated misconceptions for each question, complemented by a description of these misconceptions (see Appendix A). An interrater reliability analysis between the lead author (who had marked all responses) and the external marker was conducted on each question using the Kappa statistic and was found to be between 0.67 ($p < .001$) and 1.00 ($p < .001$). According to Landis and Koch [93], these Kappa statistics can be interpreted as ranging from substantial agreement to almost perfect agreement.

To ensure consistency, all responses were processed using the automated marking script. By continually checking the outputs, the correctness of the mappings could be verified, and any unexpected answers could be handled appropriately.

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.

As noted previously, 30% of the dataset was isolated to form a holdout-test set, with the remaining 70% being used to develop the predictive models. Following the splitting of the dataset, it was necessary to establish the initial values of the four BKT hyperparameters for each of the different programming concepts: *L0, G, S* and *T*. This was achieved by using a tool provided by Baker et al. [5], which takes a brute force approach to fitting the hyperparameters. However, parameter estimates for guess *G* and slip *S* were bounded to be below 0.3 and 0.1, respectively, to avoid model degeneracy, in which the model violates BKT's assumptions by implying that students are more likely to answer correctly without knowing a skill, or incorrectly despite knowing it [4, 5].

After the initial values for the hyperparameters had been found, the BKT equations shown previously were used to calculate the probability that a student had an appropriate mental model for each of the programming concepts. Although the ordering of questions may have appeared random to students, all participants completed the questions in the same sequence to support the BKT calculations, which must be performed in response order.

Equation 3 within the BKT calculations accounts for the possibility that a student has learned a task while 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 decided to remain consistent with the original BKT procedure and retain Equation 3. This decision was based on Corbett and Anderson's [38] assumption that a student can make the transition from an unlearned state to a learned state at any point where they can apply their knowledge, which in this case, relates to their mental models of the concepts being examined.

Following the mental model estimations being incorporated into the dataset, it was necessary to perform several pre-processing steps prior to starting the model training process. These steps included numerically encoding the features and normalising them using the MinMaxScaler [131] to accommodate models such as K-Nearest Neighbours and Support Vector Machines [68, 86], which are highly sensitive to the scale of the input features [52, 58]. To allow for the evaluation of classification models, a binarized variant of students' assessment results was created. A threshold of 50% was chosen, rather than the institutional undergraduate pass mark of 40% [33, 94, 156], in order to reduce the impact of class imbalance [88], as evidenced by the distribution of students' assessment results shown in Figure 8. This decision also reflects the intended purpose of the model, which was to identify students who are likely to require additional support rather than to predict assessment pass or fail outcomes.

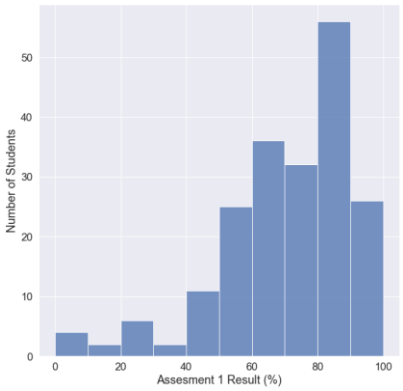

Figure 8: Assessment 1 Grade Distribution within the Training Dataset

4.3 Initial Feature Selection

The Programming Checkup collects data on a wide variety of factors. However, 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 [74, 87]. Numerous automated methods of feature selection exist [74, 87, 89], 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. [156], whereby features are placed into categories that 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 BKT.

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 due to the fact the Assessment 1 results were binarized for use within the classification model. It should be noted that all statistical analyses were conducted exclusively on the training data in order to avoid information leakage from the test set.

Tables 2 and 3 present an analysis of the relationships between the dichotomous background features and students' Assessment 1 results. To maintain consistency, both classification and regression models used the same set of features, therefore, the relationships between each of the features and both the binarized and continuous variants of the Assessment 1 results needed to be examined to identify appropriate features to utilise within the models. The distributions of the features were reviewed and confirmed to be generally non-normal, therefore necessitating the use of non-parametric statistical tests when analysing the data. The Bonferroni correction for multiple comparisons was considered when examining the

relationships between features and students' assessment results. However, given the number of features being examined and the overconservative nature of the Bonferroni correction [30], it was decided not to include the correction as to do so would likely exclude most features from the model. Furthermore, these analyses are intended as an exploratory feature-screening step rather than for formal hypothesis testing, meaning overly conservative corrections may remove variables that are weakly associated in isolation but informative within a predictive model [141].

Table 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. As "English is student's first language" violated the Chi-Squared expected count assumption [28], a Fisher's Exact Test was performed, yielding a result of p = .999.

Table 3: Chi-Squared Test Between Assessment 1 Results and Dichotomous Background Factors

Feature	U	z	p	r
Prior Programming Experience	3752.00	-1.82	.068	0.13
Previously studying 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

As students' levels of agreement regarding how strongly they considered themselves to be a "self-taught programmer" was measured using a Likert scale, a Jonckheere-Terpstra test was used to confirm a significant relationship between how strongly a student agreed/disagreed that they were a self-taught programmer and their binarized Assessment 1 result, T_{JT} = 2770.00, z = 2.19, p = .029, r = 0.15. This was then further confirmed through the significant correlation of r_s = .32, p < .001 being identified between students' level of agreement and the continuous Assessment 1 results.

The analysis of the Background Factors revealed that neither prior programming experience, nor previously studying computer science, significantly influenced students' Assessment 1 results. This may be due, in part, to the fact that a large majority of students indicated that they had prior programming experience and/or previously studied computer science (66.5% and 71%, respectively). Subsequently, these factors were not included in the predictive model due to their diminished predictive powers. The numbers of students who indicated they have previously studied computer science and potentially do not have prior programming experience suggests a need for more explicit recording of what students have previously studied, given that programming is a fundamental component of computer science courses. This is particularly prudent as many students who took part in the study were born after 2000, and as such, would have been in school when the new computer science curriculum was introduced [26].

Interestingly, how strongly students considered themselves to be self-taught did appear to be a useful predictor and 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 [12, 29, 61, 161] and was subsequently retained for both the classification and regression models, despite the non-significant chi-squared test on the binarized assessment results, in order 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 [28]. 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 was chosen to be included within the model. However, previous research has highlighted the challenges that non-native English-speaking students face when attempting to learn to program [65, 120], as well as the tendency for female students to outperform males, despite generally lower levels of self-efficacy [95, 119]. As such, the impact of these factors should be explored in future, larger, studies which ideally would involve institutions in different countries.

Students' responses for "Work in Software Engineering" were one-hot encoded 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 [132].

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.

Tables 4 and 5 examine the relationships between the continuous features within the Confidence category, and the binarized and continuous variants of students' Assessment 1 results. Upon reviewing the results, the vast majority of features appeared to be potentially useful predictors, although several features stood out as being prime candidates for being omitted from the model. For instance, students' estimation of how difficult their degree is going to be did not exhibit a significant relationship with either variant of the Assessment 1 results, subsequently leading to it not being included within the model.

Table 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

Table 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

Features which measure various aspects of students' confidence in programming were observed to have a strong relationship with performance within the Assessment 1 results, thus supporting previous claims that students' anxiety levels surrounding learning to program can have a significant impact on their performance [12, 161]. 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 [12, 29, 61, 161].

To help reduce the dimensionality of the model, it was decided to not include the category-specific confidence estimates in favour of an overall estimate, given the significant relationships observed with both the binarized and continuous variants of the Assessment 1 results. 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 strong predictor, particularly for the Conditional Statement and Iteration categories. Consequently, the mental effort estimations were not included within the model. However, it should be noted that the lower performance may be because

students were only asked to estimate their mental effort after completing all the questions rather than after each individual question, as was done with their confidence ratings. The features included in the Confidence category were as follows:

- 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).

Additionally, Tables 6 and 7 highlight the fact that most mental model estimations that were established using BKT appeared to have strong relationships with both variants of the Assessment 1 results. However, 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 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

Table 7: Spearman's Rank Tests Between Assessment 1 Results and 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

4.4 Model Evaluation and Testing

After completing the pre-processing stages described in the previous sections, 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. [156], wherein different combinations of feature categories are trialled to find an optimal model. This approach enabled a systematic comparison of a wide range of machine learning algorithms using the training data to identify the most suitable model type. GridSearchCV [133] was used to perform 10-fold cross-validation and to tune the hyperparameters for each model. For classification models, GridSearchCV employs stratified k-fold cross-validation, ensuring that class proportions are preserved across all folds [133]. The performance of the resulting models across different combinations of feature categories, evaluated solely on the training data, is reported in Tables 8 and 9, with Table 8 presenting the regression results in terms of RMSE and Table 9 summarising the classification performance using AUC.

Table 8: Ten-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. The best performing feature combination for each regression model is highlighted in bold. Lower RMSE values represent better performance.

Table 9: Ten-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	0.6667	0.6665	0.7113	0.6359	**0.7221**	0.7137	0.7194
Ridge Classifier	0.6549	0.6982	0.7191	0.6779	0.7154	**0.7232**	0.7052
SVC - Linear	0.6600	0.6609	0.7113	0.6407	**0.7137**	0.7034	0.7092
SVC - RBF	0.6471	0.6851	0.6386	0.6871	0.6020	**0.7337**	0.7054
Decision Tree	0.6562	**0.7612**	0.6644	0.7393	0.6644	0.6279	0.6292
Bagging Decision Tree	0.5582	0.5873	**0.7360**	0.6179	0.7222	0.6686	0.6791
Random Forest	0.7014	0.7578	0.7748	0.7670	0.7582	**0.7820**	0.7759
KNN - Uniform Weighting	0.5824	0.5914	**0.7484**	0.6348	0.6489	0.7351	0.6655
KNN - Distance Weighting	0.5795	0.5952	**0.7454**	0.6480	0.6462	0.7263	0.6618
Gradient Boost	0.6525	0.7268	0.7451	0.7179	**0.7799**	0.7301	0.7333
XGBoost Classifier	0.6869	0.7558	0.7395	0.7573	0.7413	0.7810	**0.7906**
MLPClassifier	0.6437	0.6771	0.7296	0.7007	0.7072	**0.7531**	0.7198

Note. The best performing feature combination for each regression model is highlighted in bold. Higher AUC values represent better performance.

The results presented in Table 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 9 range from an AUC of 0.5582 (Bagging Decision Tree, Background Factors) to 0.7906 (XGBoost, Background Factors, Confidence and Mental Models).

Based on the results presented in Tables 8 and 9, the Random Forest model was selected for further refinement for both the regression and classification tasks. For regression, the Random Forest Regressor consistently achieved among the lowest RMSE values across multiple feature combinations, particularly when mental model features were included. Two models were found to have achieved slightly better performance for specific feature combinations. Specifically, these were KNN Regressor with uniform weighting using Background Factors, Confidence, and Mental Model features (RMSE = 0.1770) and the Support Vector Regressor with an RBF kernel using Mental Model features (RMSE = 0.1772). However, the Random Forest Regressor performed consistently well across all input combinations, with its best performance observed using Confidence and Mental Model features (RMSE = 0.1773).

Similarly, for classification, the Random Forest Classifier achieved consistently high AUC scores across a range of feature combinations, with its highest AUC of 0.7820 recorded when utilising Confidence and Mental Model features. Although XGBoost achieved a slightly higher AUC of 0.7906 when using all available features, the greater stability demonstrated by Random Forest, along with its reduced susceptibility to overfitting [47, 69] as an ensemble method, made it the most appropriate model for subsequent optimisation and evaluation for both regression and classification tasks.

To refine the final models, Sequential Feature Selection [134] was applied to identify the most informative subset of features for both the regression and classification models. Hyperparameter tuning using GridSearchCV was performed within the same pipeline [135], ensuring that feature selection and parameter optimisation were conducted exclusively on the training data within each cross-validation fold and therefore preventing data leakage.

To provide an unbiased estimate of real-world performance, the final optimised models were evaluated using a holdout test set that was separated prior to model development and was not used during model selection or optimisation [128]. Given the random nature of Random Forests, this process was repeated three times to obtain an averaged measure of the performance of the models. The results were as follows:

Random Forest Regressor

Average Training RMSE:	0.1616	SD: 0.0007
Average Test RMSE:	0.1713	SD: 0.0007
Average Training MAE:	0.1209	SD: 0.007
Average Test MAE:	0.1396	SD: 0.008

Random Forest Classifier

Average Training AUC:	0.8688	SD: 0.0043
Average Test AUC:	0.7670	SD: 0.0321
Average Training F1:	0.8353	SD: 0.0057
Average Test F1:	0.8061	SD: 0.0043
Average Training Accuracy:	0.7450	SD: 0.0082
Average Test Accuracy:	0.7020	SD: 0.0055

Both the classification and regression models experienced a drop in performance when they were evaluated using the holdout test set when compared to the complete training dataset. Although a drop in performance is to be expected [60], there appears to 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 Random Forest Regressor and Classifier provide feature importance measures that indicate each feature's contribution to model performance [136, 137]. Figures 9 and 10 show the normalised feature importance scores for the classification and regression models respectively, where all feature importances sum to one and higher values indicate greater influence on the model. Whilst these plots offer insight into which features contribute most to model performance, they are model-specific and do not allow for any statistically reliable conclusions to be established about the underlying relationship between the feature and the predicted variable.

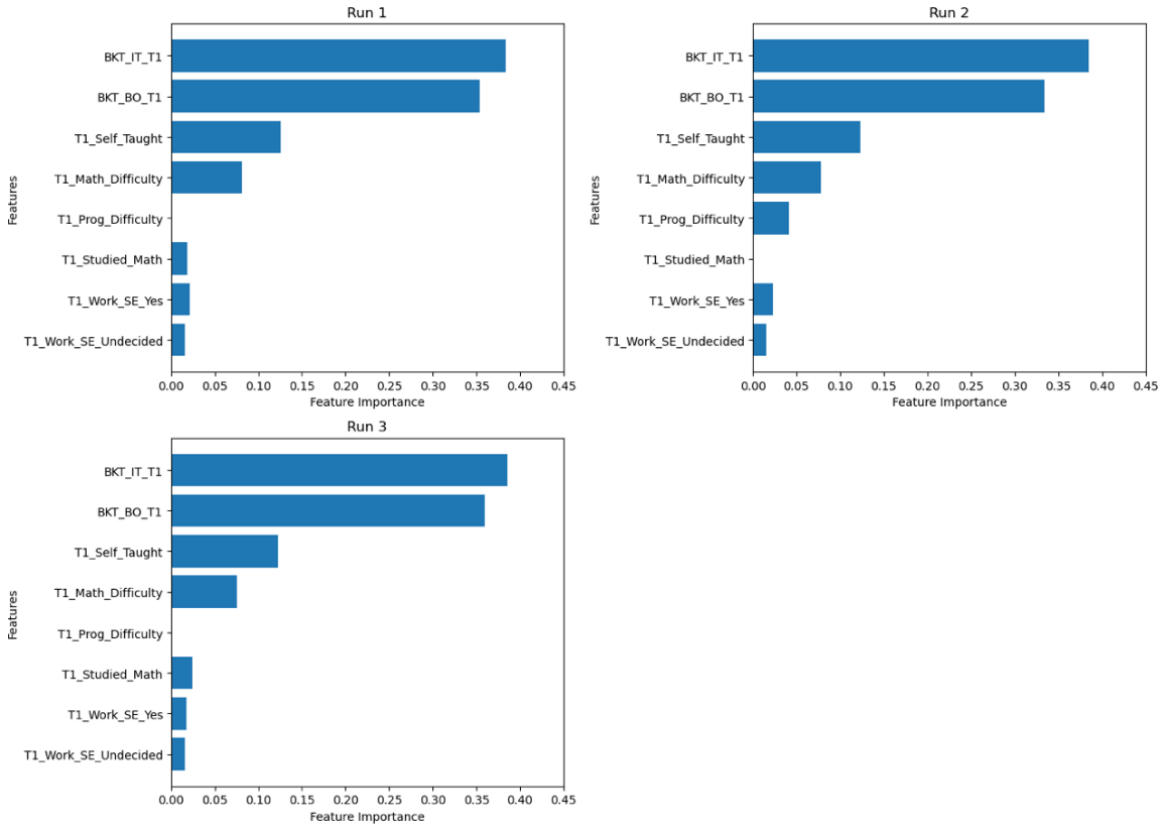

Figure 9: Random Forest Regressor Feature Importance Plots

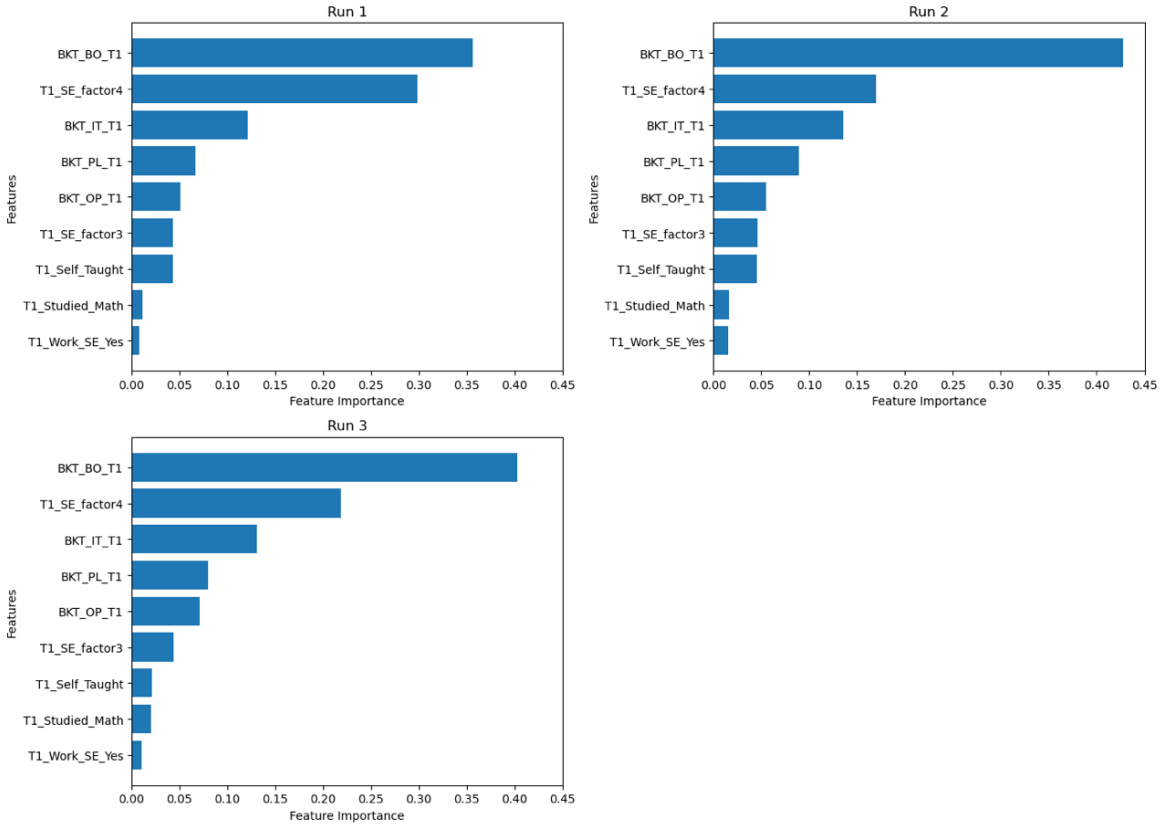

Figure 10: Random Forest Classifier Feature Importance Plots

As shown in Figures 9 and 10, feature importance scores were generally consistent across runs, with students' mental models of iteration and conditional statements being the strongest contributors to the regression model. Students' mental models of conditional statements were also identified as a highly important feature in the classification model. However, students' self-efficacy in completing simple programming tasks was also found to be an important contributor to the classification model. Although the importance of students' mental models of iteration was reduced relative to the regression model, it remained a significant feature.

4.5 Model Performance Discussion

The focus of this investigation was to explore the potential of using students' responses to the Programming Checkup to predict their assessment results. A range of machine learning algorithms and combinations of input variables derived from the Programming Checkup were evaluated, with Random Forest models ultimately selected for further refinement for both classification and regression tasks.

The performance of the regression model corresponds to an error margin of approximately 14–17 marks (14–17%), as reflected by its evaluation on the hold-out test set (RMSE = 0.1713, MAE = 0.1396). Although this represents 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, but rather to provide an indication of whether a student is likely to require additional support.

Predicting students' performance at such an early stage is inherently challenging due to the wide range of factors that may influence their outcomes [97]. As such, this margin of error is considered acceptable for use as a diagnostic guide to identify students who are likely to struggle and would benefit from additional support. Additionally, no strong evidence of overfitting was observed within the regression model, as performance remained broadly consistent between the training and test data, indicating a good level of generalisability.

A good level of performance was also observed for the classification model. However, the reduction in performance between training and evaluation on the hold-out test set, most notably in terms of AUC (training 0.8688, test 0.7670) suggests the presence of a moderate degree of overfitting. Given the consistency in performance across runs, as indicated by the relatively small standard deviations, it is likely that this overfitting is driven by class imbalance, as illustrated in Figure 8, and by the limited amount of available data. To mitigate this, future work could explore the use of a more appropriate classification threshold for identifying students likely to require support, as well as the use of a larger dataset, ideally drawn from multiple institutions, in order to improve generalisability.

5 CONCLUSIONS AND LIMITATIONS OF THE WORK

This investigation has demonstrated that by using an aptitude test, such as the Programming Checkup, it is possible to make predictions on students' assessment results prior to any teaching taking place. 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. There is, however, a need for further refinement of the both the Programming Checkup and the predictive model, which would be aided through a larger study conducted across multiple institutions.

When considering how the findings of this work could be implemented within a higher education setting, the Programming Checkup and the associated predictive model could be integrated into students' induction processes, thereby enabling targeted interventions to be put in place from the outset of a course. For instance, students identified as requiring additional support could be offered supplementary tuition and scaffolded exercises informed by the PRIMM methodology [138]. This approach is likely to be particularly beneficial in cohorts with a wide range of abilities, as targeted support could be provided to students who genuinely require it, whereas extension activities could be offered to more advanced learners who have demonstrated a strong understanding of core programming concepts. In addition to the output of the predictive model, tutors could be provided with an individual report for each student that leverages the wide range of variables examined within the Programming Checkup. The reports could include a summary of a student's prior experiences, their estimates of holding an appropriate mental model for each concept, perceived confidence levels, and predictions of their Assessment 1 performance. Reports of this type would encourage constructive dialogue between educators and students and subsequently inform the design of appropriate pedagogic interventions where necessary. This would be particularly beneficial for supporting students from non-traditional or widening participation backgrounds, who may require additional support and who may otherwise go unnoticed in the absence of an early diagnostic mechanism such as the Programming Checkup.

Whilst the findings address the original research question by demonstrating that it is possible to predict students' assessment results using a pre-course aptitude test, it is important to consider the context in which the research was conducted and to acknowledge the limitations of the work that may pose threats to internal validity.

The lead author of this study can be considered a relative insider [63], as they are involved in teaching the introductory programming module studied by the participants and therefore occupies a dual role as both educator and researcher [53]. Although this positionality provided valuable insight into the challenges faced by students, it also introduced a potential

threat to internal validity relating to power dynamics, namely the potential for students to feel compelled to participate or to be concerned that non-participation might negatively affect their grades. To mitigate this risk, students were repeatedly informed that participation was entirely voluntary and would have no impact on their assessment outcomes. They were also assured that their responses would remain confidential, and that feedback would only be provided if they explicitly opted to receive it. The fact that a sizeable proportion of students chose not to participate suggests that students did not feel unduly pressured to take part. It should also be noted that the other two co-authors had no direct interaction with the student participants during the study.

Perhaps the most significant limitations of this study are that the dataset is relatively small, with it also being constrained to students from only a single institution. Although data were collected over multiple academic years, there remains the potential for unobserved contextual factors to have influenced the results. To improve the generalisability of the findings, data should be collected from multiple institutions, thereby establishing a larger and more varied dataset. However, any future investigations involving multiple institutions must consider differences in the teaching and assessment of introductory programming modules, particularly when selecting an appropriate outcome variable to predict. Additionally, as this investigation adopted a purely quantitative approach, future work incorporating interviews and walkthroughs with students could help develop a fuller understanding of the difficulties students face when learning to program.

REFERENCES

[1] Yirsaw Ayalew, Ethel Tshukudu, Monkgogi Mudongo, Bontle Gopolang, Tsholofetso Taukobong, and Edward Zimudzi. 2025. Influence of Motivation and Learning Strategies on Performance in Introductory Programming. In *CompEd 2025 - Proceedings of the ACM Global Computing Education Conference 2025*, October 21, 2025. Association for Computing Machinery, Inc, 301–308. https://doi.org/10.1145/3736181.3747151

[2] Siti Fatimah Bahari. 2012. Qualitative versus quantitative research strategies: Contrasting epistemological and ontological assumptions. *J Teknol* (2012). https://doi.org/10.11113/jt.v52.134

[3] Ryan Baker. 2020. *Big data and education* (6th ed. ed.). University of Pennsylvania, Philadeplphia, PA.

[4] Ryan Baker, Albert Corbett, and Vincent Aleven. 2008. More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008. https://doi.org/10.1007/978-3-540-69132-7-44

[5] Ryan Baker, Albert Corbett, Sujith Gowda, Angela Wagner, Benjamin MacLaren, Linda Kauffman, Aaron Mitchell, and Stephen Giguere. 2010. Contextual slip and prediction of student performance after use of an intelligent tutor. In *International conference on user modeling, adaptation, and personalization*, 2010. Springer, 52–63. https://doi.org/10.1007/978-3-642-13470-8_7

[6] Albert Bandura. 1977. Self-efficacy: Toward a unifying theory of behavioral change. *Psychol Rev* (1977). https://doi.org/10.1037/0033-295X.84.2.191

[7] Albert Bandura. 2006. Guide for constructing self-efficacy scales. *Self-efficacy beliefs of adolescents* (2006). https://doi.org/10.1017/CBO9781107415324.004

[8] Mordechai Ben-Ari. 1998. Constructivism in computer science education. *ACM SIGCSE Bulletin* (1998). https://doi.org/10.1145/274790.274308

[9] Mordechai Ben-Ari. 2001. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* 20.1 (2001), 45–73. https://doi.org/10.1145/274790.274308

[10] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39(2) (2007), 32–36. https://doi.org/10.1145/3324888

[11] Jens Bennedsen and Michael E. Caspersen. 2019. Failure rates in introductory programming: 12 years later. *ACM Inroads* (2019). https://doi.org/10.1145/3324888

[12] Susan Bergin and Ronan Reilly. 2005. Programming: factors that influence success. In *36th SIGCSE technical symposium on Computer science education*, 2005. 411–415.

[13] Susan Bergin and Ronan Reilly. 2005. The influence of motivation and comfort-level on learning to program. *PPIG 17* (2005).

[14] Susan Bergin and Ronan Reilly. 2006. Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education* (2006). https://doi.org/10.1080/08993400600997096

[15] Anders Berglund and Raymond Lister. 2010. Introductory programming and the didactic triangle. In *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*, 2010. Australian Computer Society, Inc., 35–44.

[16] Michael Berry and Michael Kölling. 2013. The design and implementation of a notional machine for teaching introductory programming. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSE '13)*, 2013. https://doi.org/10.1145/2532748.2532765

[17] Michael Berry and Michael Kölling. 2016. Novis: A notional machine implementation for teaching introductory programming. *Proceedings - 2016 International Conference on Learning and Teaching in Computing and Engineering, LaTiCE 2016* (November 2016), 54–59. https://doi.org/10.1109/LATICE.2016.5

[18] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599.

[19] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. *Journal of the Learning Sciences* (2014). https://doi.org/10.1080/10508406.2014.954750

[20] Jeffrey Bonar and Elliot Soloway. 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human Computer Interaction* (1985). https://doi.org/10.1207/s15327051hci0102_3

[21] Richard Bornat. 2014. *Camels and humps: A retraction*. London, UK.

[22] Richard Bornat, Saeed Dehnadi, and Simon. 2008. Mental models, consistency and programming aptitude. *Conferences in Research and Practice in Information Technology Series* (2008).

[23] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[24] Benedict Du Boulay, Tim O'Shea, and John Monk. 1999. Black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human Computer Studies* (1999). https://doi.org/10.1006/ijhc.1981.0309

[25] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. 2007. Threshold concepts in computer science: Do they exist and are they useful? *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education* (2007), 504–508. https://doi.org/10.1145/1227310.1227482

[26] Neil Brown, Sue Sentance, Tom Crick, and Simon Humphreys. 2014. Restart: The resurgence of computer science in UK schools. *ACM Transactions on Computing Education* 14, 2 (2014). https://doi.org/10.1145/2602484

[27] Christine Bruce, Lawrence Buckingham, John Hynd, Camille McMahon, Mike Roggenkamp, and Ian Stoodley. 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 (January 2004), 145–160. Retrieved March 30, 2019 from https://www.learntechlib.org/p/111446/

[28] Peter Bruce, Andrew Bruce, and Peter Gedeck. 2020. *Practical statistics for data scientists: 50+ essential concepts using R and Python*. O'Reilly Media, Inc.

[29] Pat Byrne and Gerry Lyons. 2001. The effect of student attributes on success in programming. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE* (2001), 49–52. https://doi.org/10.1145/377435.377467

[30] Robert Cabin and Randall Mitchell. 2000. To Bonferroni or not to Bonferroni: when and how are the questions. *Bulletin of the Ecological Society of America* 81, 3 (2000), 246–248. Retrieved from http://www.jstor.org/stable/20168454

[31] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and validating Java misconceptions toward a CS1 concept inventory. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* 19, (July 2019), 23–29. https://doi.org/10.1145/3304221.3319771

[32] Michael E. Caspersen, Kasper Dalgaard Larsen, and Jens Bennedsen. 2007. Mental models and programming aptitude. In *ITiCSE 2007: 12th Annual Conference on Innovation and Technology in Computer Science Education - Inclusive Education in Computer Science*, 2007. https://doi.org/10.1145/1268784.1268845

[33] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 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* (March 2017), 111–116. https://doi.org/10.1145/3017680.3017792

[34] Chin Soon Cheah. 2020. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology* 12, 2 (2020).

[35] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André Santos, and Matthias Hauswirth. 2021. A curated inventory of programming language misconceptions. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* (June 2021), 380–386. https://doi.org/10.1145/3430665.3456343

[36] Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. *Computer Science Education Research* (2004).

[37] Louis Cohen, Lawrence Manion, and Keith Morrison. 2018. *Research Methods in Education* (8th ed.). Routledge, New York.

[38] Albert T. Corbett and John R. Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model User-adapt Interact* (1994). https://doi.org/10.1007/BF01099821

[39] Malcolm Corney, Raymond Lister, and Donna Teague. 2011. Early relational reasoning and the novice programmer : swapping as the "hello world" of relational reasoning. *Proceedings of the Thirteenth Australiasian Computing Education Conference* (2011). Retrieved November 18, 2023 from http://www.computing.edu.au/acsw2011/

[40] Evandro B. Costa, Baldoino Fonseca, Marcelo Almeida Santana, Fabrísia Ferreira de Araújo, and Joilson Rego. 2017. Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. *Computers in Human Behaviour* (2017). https://doi.org/10.1016/j.chb.2017.01.047

[41] Paul Curzon and Janet Rix. 1998. Why do students take programming modules? *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE* Part F1292, (1998), 59–63. https://doi.org/10.1145/282991.283022

[42] Saeed Dehnadi. 2006. Testing programming aptitude. In *PPIG 18*, 2006. 23–37.

[43] Saeed Dehnadi and Richard Bornat. 2006. The camel has two humps (working title). *Middlesex University, UK* (2006).

[44] Saeed Dehnadi, Richard Bornat, and Ray Adams. 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* (2009).

[45] Robert F DeVellis and Carolyn T Thorpe. 2021. *Scale Development Theory and Applications* (5th ed.). Sage publications.

[46] Paul Dickson, Neil Brown, and Brett Becker. 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 (June 2020), 159–165. https://doi.org/10.1145/3341525.3387404

[47] David Dietrich, Barry Heller, and Beibei Yang. 2015. *Data science and big data analytics: Discovering, analyzing, visualizing and presenting data*. Wiley.

[48] Eileen Doyle, Ioanna Stamouli, and Meriel Huggard. 2005. Computer anxiety, self-efficacy, computer experience: An investigation throughout a computer science degree. In *Proceedings - Frontiers in Education Conference, FIE*, 2005. https://doi.org/10.1109/fie.2005.1612246

[49] Hubert Dreyfus and Stuart Dreyfus. 1986. *Mind Over Machine*. Simon and Schuster.

[50] Stuart E. Dreyfus. 2004. The five-stage model of adult skill acquisition. *Bulletin Science Technology & Society* (2004). https://doi.org/10.1177/0270467604264992

[51] Dimitri Eckert, Dion Timmermann, and Christian Kautz. 2022. Student misconceptions about loops in introductory programming courses and the influence of representations. *Proceedings - Frontiers in Education Conference, FIE* 2022-October, (2022). https://doi.org/10.1109/FIE56618.2022.9962545

[52] Dino Esposito and Francesco Esposito. 2020. *Introducing machine learning*. Microsoft Press.

[53] Kim Etherington. 2007. Ethical research in reflexive relationships. *Qualitative Inquiry* (2007). https://doi.org/10.1177/1077800407301175

[54] Carol Evans. 2022. *The EAT Framework Enhancing assessment feedback practice in higher education ii*.

[55] Jonathan Evans. 2003. In two minds: Dual-process accounts of reasoning. *Trends in Cognitive Sciences* (2003). https://doi.org/10.1016/j.tics.2003.08.012

[56] Sally Fincher, Johan Jeuring, Craig Miller, Peter Donaldson, Benedict Du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice Pearce, and Andrew Petersen. 2020. Notional machines in computing education: The education of attention. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* 20, (June 2020), 21–50. https://doi.org/10.1145/3437800.3439202

[57] Paul Flowers. 2009. Research philosophies – importance and relevance. *European Journal of Information Systems* 3, 2 (2009), 112–126.

[58] Salvador García, Julián Luengo, and Francisco Herrera. 2015. *Data preprocessing in data mining*. Springer.

[59] Carlisle George. 2000. Experiences with novices: The importance of graphical representations in supporting mental models. *PPIG 12* (2000), 33–44.

[60] Aurélien Géron. 2022. *Hands-on machine learning with Scikit-learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc.

[61] Anabela Gomes, Lilian Carmo, Emilia Bigotte, and António Mendes. 2006. Mathematics and programming problem solving. In *3rd E-Learning Conference–Computer Science Education*, 2006. Citeseer, 1–5.

[62] Tina Götschi, Ian Sanders, and Vashti Galpin. 2003. Mental models of recursion. In *Proceedings of the 34th SIGCSE technical symposium on computer science education.*, 2003. https://doi.org/10.1145/792548.612004

[63] Morwenna Griffiths. 1998. *Educational research for social justice: Getting off the fence*. McGraw-Hill Education (UK).

[64] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean logic. In *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 2017. https://doi.org/10.1145/3017680.3017723

[65] Philip Guo. 2018. Non-native English speakers learning computer programming: Barriers, desires, and design opportunities. *Conference on Human Factors in Computing Systems - Proceedings* 2018-April, (April 2018). https://doi.org/10.1145/3173574.3173970

[66] Mark Guzdial. 2010. Why is it so hard to learn to program. In *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 111–124.

[67] Muntasir Hoq, Peter Brusilovsky, and Bita Akram. 2023. Analysis of an Explainable Student Performance Prediction Model in an Introductory Programming Course. In *Proceedings of the International Conference on Educational Data Mining*, 2023. International Educational Data Mining Society, 79–90. https://doi.org/10.5281/zenodo.8115693

[68] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. 2008. A practical guide to support vector classification. *BJU Int* 101, 1 (2008), 1396–1400. Retrieved from http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

[69] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Springer, New York.

[70] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of student misconceptions using python as an introductory programming language. *Proceedings of the 4th Conference on Computing Education Practice (CEP '20)* (January 2020). https://doi.org/10.1145/3372356.3372360

[71] Philip N. Johnson-Laird. 1983. *Mental models: Towards a cognitive science of language, inference, and consciousness* (6th ed.). Harvard University Press.

[72] Philip N. Johnson-Laird. 2010. Mental models and human reasoning. *Proceedings of the National Academy of Sciences* (2010). https://doi.org/10.1073/pnas.1012933107

[73] Philip N. Johnson-Laird, Monica Bucciarelli, Robert Mackiewicz, and Sangeet S Khemlani. 2022. Recursion in programs, thought, and language. *Psychonomic Bulletin & Review* 29, 2 (2022), 430–454. https://doi.org/10.3758/s13423-021-01977-y/Published

[74] Alan Jovic, Karla Brkic, and Nikola Bogunovic. 2015. A review of feature selection methods with applications. *2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO)* (2015), 1200–1205.

[75] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. In *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 2010. https://doi.org/10.1145/1734263.1734299

[76] Hank Kahney. 1983. What do novice programmers know about recursion. In *Conference on Human Factors in Computing Systems - Proceedings*, 1983. https://doi.org/10.1145/800045.801618

[77] Maria Kallia and Sue Sentance. 2017. Computing teachers⇔ perspectives on threshold concepts: Functions and procedural abstraction. In *ACM International Conference Proceeding Series*, November 08, 2017. Association for Computing Machinery, 15–24. https://doi.org/10.1145/3137065.3137085

[78] Maria Kallia and Sue Sentance. 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* (February 2019), 752–758. https://doi.org/10.1145/3287324.3287377

[79] Maria Kallia and Sue Sentance. 2021. Threshold concepts, conceptions and skills: Teachers' experiences with students' engagement in functions. *Journal of Computer Assisted Learning* 37, 2 (April 2021), 411–428. https://doi.org/10.1111/jcal.12498

[80] Geetha Kanaparan, Rowena Cullen, and David Mason. 2019. Effect of self-efficacy and emotional engagement on introductory programming students. *Australasian Journal of Information Systems* 23, (2019).

[81] Odd Kaufmann and Børre Stenseth. 2021. Programming in mathematics education. *International Journal of Mathematical in Education in Science and Technology* 52, 7 (2021), 1029–1048. https://doi.org/10.1080/0020739X.2020.1736349

[82] Claudius. Kessler and John. Anderson. 1986. Learning flow of control: Recursive and iterative procedures. *Human Computer Interaction* 2, 2 (1986), 135–166.

[83] Joseph. Khalife. 2006. Threshold for the introduction of programming: Providing learners with a simple computer model. In *28th International Conference on Information Technology Interfaces, 2006.*, 2006. IEEE, 71–76.

[84] Natalie Kiesler. 2022. Reviewing Constructivist Theories to Help Foster Creativity in Programming Education. In *Proceedings - Frontiers in Education Conference, FIE*, 2022. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/FIE56618.2022.9962699

[85] Mario Konecki and Marko Petrlic. 2014. Main problems of programming novices and the right course of action. In *Central European Conference on Information and Intelligent Systems*, 2014. Faculty of Organization and Informatics Varazdin, 116.

[86] Sotiris Kotsiantis. 2007. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering* 160, 1 (2007), 3–24.

[87] Sotiris Kotsiantis, Dimitris Kanellopoulos, and Panagiotis Pintelas. 2006. Data preprocessing for supervised leaning. *Int J Comp Sci* 1, 2 (2006), 1–7. https://doi.org/10.1080/02331931003692557

[88] Max Kuhn and Kjell Johnson. 2013. *Applied predictive modeling*. Spinger. https://doi.org/10.1007/978-1-4614-6849-3

[89] Max Kuhn and Kjell Johnson. 2019. *Feature engineering and selection: A practical approach for predictive models*. Chapman and Hall/CRC. https://doi.org/10.1201/9781315108230

[90] D. Kurland and Roy. Pea. 1985. Children's mental models of recursive Logo programs. *Journal of Educational Computing Research* (1985). https://doi.org/10.2190/jv9y-5pd0-mx22-9j4y

[91] Kyungbin Kwon. 2017. Novice programmer's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools* (2017). https://doi.org/10.21585/ijcses.v1i4.19

[92] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin* 37, 3 (June 2005), 14–18. https://doi.org/10.1145/1151954.1067453

[93] Richard Landis and Gary Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977). https://doi.org/10.2307/2529310

[94] Soohyun Nam Liao, Daniel Zingaro, Kevin Thai, Christine Alvarado, William G. Griswold, and Leo Porter. 2019. A robust machine learning technique to predict low-performing students. *ACM Transactions on Computing Education* 19, 3 (January 2019), 18. https://doi.org/10.1145/3277569

[95] Alex Lishinski, Aman Yadav, Jon Good, and Richard Enbody. 2016. Learning to program: Gender differences and interactive effects of students' motivation, goals, and self-efficacy on performance. In *2016 ACM conference on international computing education research*, 2016. 211–220.

[96] Jose Llanos, Víctor A. Bucheli, and Felipe Restrepo-Calle. 2023. Early prediction of student performance in CS1 programming courses. *PeerJ Computer Science* 9, (2023). https://doi.org/10.7717/PEERJ-CS.1655

[97] Javier López-Zambrano, Juan Torralbo, and Cristóbal Romero. 2021. Early prediction of student learning performance through data mining: A systematic review. *Psicothema* 33, 3 (2021), 456–465. https://doi.org/10.7334/psicothema2021.62

[98] Andrew Luxton-Reilly. 2016. Learning to program is easy. In *2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016. 284–289.

[99] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett Becker, Michail Giannakos, Amruth Kumar, Linda Ott, James Paterson, Michael Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: A systematic literature review. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* (July 2018), 55–106. https://doi.org/10.1145/3293881.3295779

[100] Linxiao Ma, John. Ferguson, Marc Roper, Isla Ross, and Murray Wood. 2008. Using cognitive conflict and visualisation to improve mental models held by novice programmers. *ACM SIGCSE Bulletin* (2008). https://doi.org/10.1145/1352322.1352253

[101] Tia C. Madkins, Nicol R. Howard, and Natalie Freed. 2020. Engaging Equity Pedagogies in Computer Science Learning Environments. *Journal of Computer Science Integration* (November 2020), 1–27. https://doi.org/10.26716/jcsi.2020.03.2.1

[102] Carlos Márquez-Vera, Alberto Cano, Cristobal Romero, Amin Yousef, Mohammad Noaman, Habib Mousa, and Sebastian Ventura. 2015. Early dropout prediction using data mining: a case study with high school students. *Expert Systems* 33, 1 (February 2015), 107–124. https://doi.org/10.1111/exsy.12135

[103] Samuel Messick. 1995. Validity of psychological assessment: Validation of inferences from persons' responses and performances as scientific inquiry into score meaning. *American psychologist* 50, 9 (December 1995). https://doi.org/10.1002/j.2333-8504.1994.tb01618.x

[104] Jason W. Morphew, Mariana Silva, Geoffrey Herman, and Matthew West. 2020. Frequent mastery testing with second-chance exams leads to enhanced student learning in undergraduate engineering. *Applied Cognitive Psychology* 34, 1 (January 2020), 168–181. https://doi.org/10.1002/acp.3605

[105] Donald Norman. 1983. Some observations on mental models. In *Mental Models*.

[106] OCR. 2015. OCR GCSE (9-1) Computer Science Pseudocode Guide. Retrieved January 3, 2026 from https://www.ocr.org.uk/Images/202654-pseudocode-guide.pdf

[107] Uzma Omer and Muhammad Shoaib Farooq. 2020. Cognitive learning analytics using assessment data and concept map: A framework-based approach for sustainability of programming courses. *Sustainability* 12, 17 (2020).

[108] Uzma Omer, Muhammad Shoaib Farooq, and Adnan Abid. 2021. Introductory programming course: Review and future implications. *PeerJ Computer Science* 7, (2021). https://doi.org/10.7717/peerj-cs.647

[109] Fred Paas and Jeroen Van Merriënboer. 1994. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational Psychological Rev*iew (1994). https://doi.org/10.1007/BF02213420

[110] Zachary Pardos and Neil Heffernan. 2010. Navigating the parameter space of Bayesian knowledge tracing models: Visualizations of the convergence of the expectation maximization algorithm. In *Educational Data Mining 2010 - 3rd International Conference on Educational Data Mining*, 2010.

[111] Roy Pea. 1986. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research* (1986). https://doi.org/10.2190/689t-1r2a-x4w4-29j2

[112] Roy Pea and D. Midian Kurland. 1984. On the cognitive effects of learning computer programming. *New Ideas in Psychology* (1984). https://doi.org/10.1016/0732-118X(84)90018-7

[113] David Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of learning in novice programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55.

[114] João P.J. Pires, Fernanda Brito Correia, Anabela Gomes, Ana Rosa Borges, and Jorge Bernardino. 2024. Predicting Student Performance in Introductory Programming Courses. *Computers 13*. https://doi.org/10.3390/computers13090219

[115] Y Qian and J Lehman. 2016. Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning* 5, 2 (2016), 73–83. https://doi.org/10.5539/jel.v5n2p73

[116] Yizhou Qian, Susanne Hambrusch, Aman Yadav, Sarah Gretter, and Yue Li. 2020. Teachers' perceptions of student misconceptions in introductory programming. *Journal of Educational Computing Research* 58, 2 (April 2020), 364–397. https://doi.org/10.1177/0735633119845413/ASSET/IMAGES/LARGE/10.1177_0735633119845413-FIG6.JPEG

[117] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*. https://doi.org/10.1145/3077618

[118] Keith Quille and Susan Bergin. 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* (July 2018), 15–20. https://doi.org/10.1145/3197091.3197101

[119] Keith Quille, Natalie Culligan, and Susan Bergin. 2017. Insights on gender differences in CS1: A multi-institutional, multi-variate study. In *2017 ACM conference on innovation and technology in computer science education*, 2017. 263–268.

[120] Adalbert Raj, Kasama Ketsuriyonk, Jignesh Patel, and Richard Halverson. 2017. What do students feel about learning programming using both English and their native language? In *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, 2017. IEEE, 1–8.

[121] Vennila Ramalingam and Susan Wiedenbeck. 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* (1998). https://doi.org/10.2190/C670-Y3C8-LTJ1-CT3P

[122] Anthony Robins. 2010. Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education* 20, 1 (March 2010), 37–71. https://doi.org/10.1080/08993401003612167

[123] Anthony Robins. 2019. Novice programmers and introductory programming. In *The Cambridge handbook of computing education research*. 327–376.

[124] Janine Rogalski and Renan Samurçay. 1990. Acquisition of programming knowledge and skills. In *Psychology of programming*. Elsevier, 157–174.

[125] Christine Rogerson and Elsje Scott. 2010. The fear factor: How it affects students learning to program in a tertiary environment. *Journal of Information Technology Education: Research* (2010). https://doi.org/10.28945/1183

[126] Cristóbal Romero and Sebastian Ventura. 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 (2010), 601–618. https://doi.org/10.1109/TSMCC.2010.2053532

[127] Cristóbal Romero and Sebastian Ventura. 2019. Guest editorial: Special issue on early prediction and supporting of learning Performance. *IEEE Transactions on Learning Technologies* 12, 2 (2019), 145–147. https://doi.org/10.1109/TLT.2019.2908106

[128] Stuart Russell and Peter Norvig. 2020. *Artificial intelligence: a modern approach* (4th ed.).

[129] Kate Sanders and Robert McCartney. 2016. Threshold concepts in computing: Past, present, and future. In *ACM International Conference Proceeding Series*, November 24, 2016. Association for Computing Machinery, 91–100. https://doi.org/10.1145/2999541.2999546

[130] Angela Sasse. 1997. Eliciting and describing users' models of computer systems. (Doctoral dissertation, University of Birmingham).

[131] Scikit-Learn. MinMaxScaler— scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[132] Scikit-Learn. OneHotEncoder – scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

[133] Scikit-Learn. GridSeachCV — scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[134] Scikit-Learn. SequentialFeatureSelector – scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html

[135] Scikit-Learn. Pipeline — scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

[136] Scikit-Learn. RandomForestRegressor — scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[137] Scikit-Learn. RandomForestClassifier — scikit-learn 1.8.0 documentation. Retrieved January 3, 2026 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[138] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education* 29, 2–3 (2019), 136–176. https://doi.org/10.1080/08993408.2019.1608781

[139] Sadia Sharmin, Daniel Zingaro, Llisa Zhang, and Clare Brett. 2019. Impact of open-ended assignments on student self-efficacy in CS1. *CompEd 2019 - Proceedings of the ACM Conference on Global Computing Education* 19, (May 2019), 215–221. https://doi.org/10.1145/3300115.3309532

[140] Eman Sherif, Jayne Everson, F. Megumi Kivuva, Mara Kirdani-Ryan, and Amy J. Ko. 2024. Exploring the Impact of Assessment Policies on Marginalized Students' Experiences in Post-Secondary Programming Courses. In *ICER 2024 - ACM Conference on International Computing Education Research*, August 13, 2024. Association for Computing Machinery, Inc, 243–245. https://doi.org/10.1145/3632620.3671100

[141] Galit Shmueli. 2010. To explain or to predict? *Statistical Science* 25, 3 (August 2010), 289–310. https://doi.org/10.1214/10-STS330

[142] 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* (2011), 10–15. https://doi.org/10.1145/2094131.2094134

[143] Simon, Sally Fincher, Anthony Robins, Bob Baker, Quintin Cutts, Patricia Haden, Margaret Hamilton, Marian Petre, Denise Tolhurst, Ilona Box, Michael de Raadt, John Hamer, Raymond Lister, Ken Sutton, and Jodi Tutty. 2006. Predictors of success in a first programming course. *Conferences in Research and Practice in Information Technology Series* (2006).

[144] Simon, Andrew Luxton-Reilly, Vangel V. Ajanovski, Eric Fouh, Christabel Gonsalvez, Juho Leinonen, Jack Parkinson, Matthew Poole, and Neena Thota. 2019. Pass rates in introductory programming and in other STEM disciplines. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, December 18, 2019. Association for Computing Machinery, 53–71. https://doi.org/10.1145/3344429.3372502

[145] Teemu Sirkiä and Juha Sorva. 2012. Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings - 12th Koli Calling International Conference on Computing Education Research, Koli Calling 2012*, 2012. https://doi.org/10.1145/2401796.2401799

[146] Derek Sleeman, Ralph T. Putnam, Juliet Baxter, and Laiani Kuspa. 1986. Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research* (1986). https://doi.org/10.2190/2xpp-ltyh-98nq-bu77

[147] John Smith, Andrea DiSessa, and Jeremy Roschelle. 1994. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences* (1994). https://doi.org/10.1207/s15327809jls0302_1

[148] Juha Sorva. 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* (2010), 21–30. https://doi.org/10.1145/1930464.1930467

[149] Juha Sorva. 2012. *Visual program simulation in introductory programming education.* Aalto University.

[150] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (July 2013). https://doi.org/10.1145/2483710.2483713

[151] Mateja Strnad, Irena N. Šerbec, and Jože Rugelj. 2009. Programming aptitude and learning success in the introductory course on programming. In *12th International Conference on Interactive Computer aided Learning*, 2009. Villach, Austria.

[152] Leigh Sudol and Ciera Jaspan. 2010. Analyzing the strength of undergraduate misconceptions about software engineering. (2010), 31. https://doi.org/10.1145/1839594.1839601

[153] A Swidan, F Hermans, and M Smit. 2018. Programming misconceptions for school students. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research* (August 2018), 151–159. https://doi.org/10.1145/3230977.3230995

[154] Claudia Szabo, Judy Sheard, Lauri Malmi, and Paivi Kinnunen. 2025. Parsons problems and computing education learning theories. In *Proceedings of 25th International Conference on Computing Education Research, Koli Calling 2025*, November 10, 2025. Association for Computing Machinery, Inc. https://doi.org/10.1145/3769994.3770032

[155] F. Boray Tek, Kristin S. Benli, and Ezgi Deveci. 2018. Implicit theories and self-efficacy in an introductory programming course. *IEEE Transactions on Education* 61, 3 (August 2018), 218–225. https://doi.org/10.1109/TE.2017.2789183

[156] Nikola Tomasevic, Nikola Gvozdenovic, and Sanja Vranes. 2020. An overview and comparison of supervised data mining techniques for student exam performance prediction. *Computers & Education* 143, (January 2020), 103676. https://doi.org/10.1016/J.COMPEDU.2019.103676

[157] Tammy VanDeGrift, Dennis J. Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Beth Simon. 2010. Commonsense computing (episode 6) logic is harder than pie. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 2010. 76–85.

[158] Phill Ventura and Bina Ramamurthy. 2004. Wanted: CS1 students. no experience required. *ACM SIGCSE Bulletin* 36, 1 (2004), 240–244.

[159] Christopher Watson and Frederick Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, 2014. https://doi.org/10.1145/2591708.2591749

[160] Susan Wiedenbeck. 1989. Learning iteration and recursion from examples. *International Journal of Man-Machachine Studies* 30, 1 (January 1989), 1–22. https://doi.org/10.1016/S0020-7373(89)80018-5

[161] Brenda Wilson and Sharon Shrock. 2001. Contributing to success in an introductory computer science course: A study of twelve factors. *ACM SIGCSE Bulletin* 33, 1 (2001). https://doi.org/10.1145/366413.364581

[162] Leon E. Winslow. 1996. Programming pedagogy---a psychological overview. *ACM SIGCSE Bulletin* (1996). https://doi.org/10.1145/234867.234872

[163] Muhammed Yousoof, Mohd Sapiyan, and Khaja Kamaluddin. 2007. Measuring cognitive load – A solution to ease learning of programming. *Proceedings of World Academy of Science, Engineering and Technology* 1, 2 (2007), 216–217.

[164] Michael Yudelson, Kenneth Koedinger, and Geoffrey Gordon. 2013. Individualized Bayesian knowledge tracing models. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013. https://doi.org/10.1007/978-3-642-39112-5-18

[165] Žana Žanko, Monika Mladenović, and Ivica Boljat. 2019. Misconceptions about variables at the K-12 level. . *Education Information Technologies* 24, 2 (March 2019), 1251–1268. https://doi.org/10.1007/S10639-018-9824-1/TABLES/18

[166] Žana Žanko, Monika Mladenović, and Divna Krpan. 2022. Analysis of school students' misconceptions about basic programming concepts. *J Comput Assist Learn* 38, 3 (June 2022), 719–730. https://doi.org/10.1111/JCAL.12643

[167] Daniel Zingaro. 2014. Peer Instruction contributes to self-efficacy in CS1. *SIGCSE 2014 - Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (2014), 373–378. https://doi.org/10.1145/2538862.2538878

APPENDIX A

Misconception Definitions and Associated Mental Models

Code	Misconception	Description	Associated Mental Model
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 / AND
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 / IF
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 / NOT
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 / OR
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

Code	Misconception	Description	Associated Mental Model
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