Master's thesis
Master's Programme in Data Science

Predicting programming assignment difficulty

Esa Harju

May 16, 2019

Supervisor(s): Dr Arto Hellas and MSc Juho Leinonen

Examiner(s): Associate professor Petri J. Ihantola
Dr Arto Hellas

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Esa Harju

Työn nimi — Arbetets titel — Title

Predicting programming assignment difficulty

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	May 16, 2019	70

Tiivistelmä — Referat — Abstract

Teaching programming is increasingly more widespread and starts at primary school level on some countries. Part of that teaching consist of students writing small programs that will demonstrate learned theory and how various things fit together to form a functional program. Multiple studies indicate that programming is difficult skill to learn and master. Some part of difficulty comes from plethora of concepts that students are expected to learn in relatively short time.

Part of practicing to write programs involves feedback, which aids students' learning of assignment's topic, and motivation, which encourages students to continue the course and their studies. For feedback it would be helpful to know students' opinion of a programming assignment difficulty. There are few studies that attempt to find out if there is correlation between metrics that are obtained from students' writing a program and their reported difficulty of it. These studies use statistical models on data after the course is over.

This leads to an idea if such a thing could be done while students are working on programming assignments. To do this some sort of machine learning model would be possible solution but as of now no such models exist. Due to this we will utilize idea from one of these studies to create a model, which could do such prediction. We then improve that model, which is coarse, with two additional models that are more tailored for the job.

Our main results indicate that this kind of models show promise in their prediction of a programming assignment difficulty based on collected metrics. With further work these models could provide indication of a student struggling on some assignment. Using this kind of model as part of existing tools we could provide a student subtle help before his frustration grows too much. Further down the road such a model could be used to provide further exercises, if need by a student, or progress forward once he masters certain topic.

ACM Computing Classification System (CCS):
Social and professional topics → *Computing education*
General and reference → *Verification*
General and reference → *Metrics*

Avainsanat — Nyckelord — Keywords

replication, re-analysis, automated assessment, assignment difficulty, programming assignments

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1. Introduction

While programming is being taught increasingly already in primary school level around the world ([40], [111] and [53]), it is tertiary level education that teaches it for those who are planning to become professional programmers. For the youth goal in programming is to learn logical thinking, problem solving, creativity, expressing oneself or combination of these and other such skills. At tertiary level teaching focuses more on skills, like algorithms or object-oriented paradigm, that require e.g. higher level of abstract thinking.

According to various studies, e.g. [73], [49], [85] and [94], learning to program is difficult and on average approximately 67% of introductory course students pass it, worldwide, as noted by Bennedsen and Caspersen [8] and Watson and Li [112]. Interestingly pass rates varies a lot and is anything from 5% upwards according to Bennedsen and Caspersen, which means that on some cases 95% of students either fail, quit or otherwise skip the course. Watson and Li note that there has not been improvement on these numbers over time.

As what makes programming difficult have been studied in numerous papers and from multiple viewpoints. Winslow [114] looked at used pedagogy from psychological side. He noted that it is odd to teach more advanced material to students when they still have issues with basic concepts like loop constructs, inter alia. Jenkins [66] looked at it from teaching perspective that consist of cognitive side, which includes one's learning style and motivation, and what makes up programming in the first place, such as requirement for multiple skills or learning language syntax. He summarizes multiple items to improve teaching programming at universities including that it should be left to those who can teach instead of those who do. Lister [75] looked at student's ability to abstract things at various levels. For example, a student may provide correct description of a program line-by-line or summarize overall what given code does. He notes that students' abilities vary a lot in this respect and that effects on their ability to learn to program.

MD Derus and Mohamad Ali [85] looked at difficulty from students' point of view. They covered parts related to programming, understanding of a topic, practicalities that would be helpful and factors affecting performance. Among other things they noted that students' have difficulties comprehending basic topics and creating a solution for given assignment. At more hands-on level McCracken et al. [84] asked students to write short programs, evaluated them and concluded that students are unable to translate assignment description to a working program. Lister et al. [76] asked students to read, evaluate and select correct solution for various program codes. They indicate that students do not have understanding and expertise that are forerunners of solving problems.

Common theme to these and several other studies is that they produce information and results after things are done. When students are learning to program their main activity is writing small programs that show them how theory works in practice. The programming assignments difficulty level range from easy to difficult. From educational point of view having knowledge of an assignment difficulty is interesting, both afterwards and when it happens. In the former case a teacher obtains information that would allow him to improve teaching, assignments or both in those areas that students' find more challenging. In the latter case students could be provided subtle help before they will give up on some assignment or the course altogether. A key in the latter is timing, as rushing in with help too early or waiting for too long can do more harm than good.

In traditional teaching only way to judge a programming assignment difficulty is up to a teacher observing students and his interactions with them when they work on those assignments. This tends to work on a small group but when there are several tens or hundreds of students it fails. Similarly, if it is a massive open online course (MOOC) then traditional way is no longer applicable and we need something else. There are various ways available ranging from simply asking a student to using psychophysiological sensors, which could measure e.g. size of pupil in the eye. However, they are not equally good or usable on all occasions. One simpler option is to utilize data based on what students' output, code, and what they do. From code we can obtain various software metrics like number of variables in it, length of the code in lines, number of function calls made and whether the program passes all the tests that staff has created for this assignment. Likewise, using suitable tools we can collect data on what they do like typing speed, time spent on an assignment or utilization of copy-pasting.

Using these kinds of metrics combined with knowledge on a programming assignment difficulty is a quite new field. In one study, by Alvarez and Scott [3], researchers collected some metrics from students who completed programming assignments and for each assignment students rated its difficulty both before and after it was done. In another study by Ihantola, Sorva and Vihavainen [62] a set of metrics were collected along with a difficulty rating provided by a student. Both studies used statistical tools to find out if there are any correlations between collected metrics and difficulty rating provided by students.

An interesting step further on this path would be having ability to do what they did but in real time. To do so we would need a machine learning model that would attempt to predict an assignment difficulty based on collected data while such assignment is being worked on. If we had such a model available, we could utilize it to predict what is a student's opinion about given assignment difficulty before he has finished that assignment. However, as of now no such model exists. In this thesis we will build couple of such envisioned machine learning models using data, which is collected from a single programming course. We will also build a model based on idea stemming from Ihantola, Sorva and Vihavainen paper to be used as a baseline.

This thesis is organized as follows. In chapter 2 we will cover background information on what difficulty is, how to measure it, what requirements there exists for a programmer and how difficulty can be measured in programming. Finally, at the end of that chapter we will dive into related work on a topic. Next chapter 3 will cover

basics of machine learning followed by closer look at one algorithm category called decision tree and discusses about metrics to evaluate a machine learning model. In chapter 4 we present our research questions and context along with description of used data. In chapter 5 we present results of the models along with observations from them. In chapter 6 we discuss the models that we created and talk about their limitations. Finally, chapter 7 wraps things up along with possible ideas for future work.

2. Measuring difficulty in programming

In this chapter we will cover what difficulty is and how it could be measured. Next, we will discuss what kind of requirements there are for a person who wants to be a programmer to get an idea why programming can be difficult and how difficulty in programming could be measured. Last we go over other work on the topic of this thesis.

2.1 Difficulty and cognitive load

In this section we will talk about difficulty, which can be defined as "the quality, fact, or condition of a thing being hard to deal with or of presenting obstacles to progress or accomplishment" (Oxford English Dictionary [90]), via field of cognitive psychology. We cover long-term and working memories, and the latter closer due to relating more to the subject of difficulty and its measurement. We end with discussion of various ways to measure working memory use.

Borg, Bratfisch and Dornič [11] noted three main factors affecting how one has trouble related to a task: identification, background factors and momentary conditions. In the first one searches his general experience and possible memories of a same kind of task than what is currently faced. The second one consists of one's past that form his backgrounds like habits, aspiration, personality traits and general attitudes. The third one includes current conditions such as fatigue, motivation, tasks importance to one and anticipated outcome of the task. Utilizing all these factors together effects on person's experience of difficulty on a task. They indicate that there might be secondary factors, which they do not explained further, that might affect to a degree of difficulty that we experience. It appears that difficulty and its measurement is a non-trivial thing to do, so according to Borg, Bratfisch and Dornič [11] Borg [10] suggested that we would measure perceived difficulty instead.

We will cover perceived difficulty from cognitive load theory point of view. Cognitive load attempts to describe the effort of the working memory usage and belongs to a field of cognitive psychology. Cognitive load theory is based on Sweller's [103] study of problem solving. To understand how this relates to difficulty we need to dive into a bit of human cognitive architecture part that is related to memory.

2.1.1 Memory and cognitive load

Miller [86] described that the span of immediate memory, i.e. its capacity, is seven items plus or minus two. This means that on average human can hold seven items on his mind, give or take two. The span of immediate memory was late termed as working memory by Miller, Galanter and Pribram [87]. Sweller, Merriënboer and Paas [106] considered working memory to be equivalent to consciousness and the rest of our cognitive functions being unavailable to us up to the point that they are brought to our working memory. It is a place for holding information temporarily while we are processing it.

Working memory capacity is quite small and it appears to get overwhelmed quickly if there are interactions between the items that are held in it. As an example, a set of items like {9, 9, 2, 7, 4, 3, 3} can be held in working memory relatively easy since interactivity between items is low. It means that these items can be separate from one another save for possible ordering, if that is relevant to the case. On the other hand, a weekday problem, like one in study by Schmeck et al. [97], is much harder and is an example where items have high interactivity. Problem states: "Suppose last Tuesday was the 2^{nd} day. What day of the week it is in 17 days, if the 8^{th} day is in two days?" (Schmeck et al. [97].) The high interactivity items cannot be learned in isolation since it does not make any sense. This means that we need to do some processing while solving the problem and while processing we add more items to working memory, which lowers its capacity. This can be easily seen by considering mentioned weekday problem where one might calculate that today is two days before 8^{th} day, so today is 6^{th} day and that item is added to working memory along with other items held in it in order to continue solving the problem.

Based on the limited capacity of working memory and fact that anything more than uncomplicated cognitive activities appears to swamp working memory, it is amazing that we can accomplish a lot like master game of Go or solve complex mathematical issues such as asymptotic Dirichlet problem [57]. A study by de Groot [34] on various level of chess players from grandmasters on the World Chess Federation (Fédération Internationale des Échecs (FIDE)) to class C equivalent players on the United States Chess Federation (USCF) attempted to find out what differentiated them from one another. From various statistics related to searching, moves considered, etc. there were no noticeable differences other than masters and higher appeared to come up with better moves than less skilled players. On that same study de Groot performed an experiment related to memory use in which players were attempting to reconstruct a board configuration of scene taken from a real game after seeing it only for five seconds. Figure 2.1 shows an example case of this study. On this experiment master level players were clearly better than players below that level. When De Groot and later by Chase and Simon [25] utilized random board configurations for that reconstruction experiment master level players did not outperform less skilled players. It appeared to confirm that master level players were restricted by same working memory constrain than less skilled players, so something else was behind their better performance.

Based on those observations something from already played games appears to hold a key to master's performance. De Groot [34] noted that grandmasters could identify several thousands of real game board configurations. Simon and Gilmartin

Figure 2.1: Dutch national championship preliminaries on April 10, 1936 as a position A in de Groot [34] study with picture reproduced by Bilalić, McLeod and Gobet [9]. White to move.

[102] concluded after their simulation study that master level players have capacity of less than 100 000 configurations, but they could achieve their performance with as little as 10 000 configurations, which is still a lot compared to less skilled players. Master level players once seeing a certain board configuration can recall it from their memory and utilize suitable moves related to that configuration instead of using their working memory to search for good moves. Consequently, less skilled players who have less knowledge available to them must resort to complex reasoning, which is likely to overload their working memory. Similar findings between novices and experts have been acquired in various other domains like software development by Barfield [6], electronics by Egan and Schwartz [38] and mathematics by Sweller and Cooper [105].

Chase and Simon [25] noted that if master level players can hold location of twenty or more pieces in their memory but are constrained by working memory limits then they must somehow group these pieces into some meaningful structure. They called this group a chunk, which is one of many terms assigned to it, but more commonly used term is a schema in cognitive related literature. It allows us to organize categories of information and how they relate to one another as noted by e.g. Chi, Glaser and Rees [27]. As stated in schema theory skilled performance develops when we combine simple low-level schemata into more complex high-level schemata, see e.g. Sweller, Ayres and Kalyuga [104] pp. 22-23. To get an idea how this presents itself in our mind we can think of a small task on programming and robotics. You would like to program a robot to go to movies. Quite likely you will start that process by breaking it down to several suitable smaller steps that you will continue to split even smaller steps and refine them until you have achieved your goal. For you going to see a movie is easy since you know what it entails like how to find what movies are shown, how to obtain a ticket, how to get to a theater, what to do at the theater, and so on. All those steps

can be thought as low-level schemata and when put together they form a higher-level schema about going to movies. This schema is stored in long-term memory that acts as a single element in working memory, thus utilizing only space of single item in it.

When we attempt to learn something, we utilize our working memory to store and manipulate information before it will end up in long-term memory. As noted, working memory has very limited capacity and anything put into it creates a load, which in this case is called cognitive load. Cognitive load can be divided into two categories; intrinsic or extraneous cognitive load, see Sweller et al. [104] chapter 5. Former refers to intrinsic nature of the information that one must obtain in order to achieve learning goals. An example of this can be observed in basic arithmetic. It is easier to add two numbers together, say 4 + 7, than to make division between two numbers, say 13 / 7. Division creates more intrinsic cognitive load than addition. Latter type of load, extraneous, comes from possible activities involved in learning or how material that should be learned is presented*. For example, we can define a square, as in Usiskin et al. [107], to be a parallelogram, which is a quadrilateral with two pairs of parallel sides and a quadrilateral being a four-sided polygon, that is both a rectangle, which is an equiangular quadrilateral, and a rhombus, which is a parallelogram with four equal sides, or show it visually as seen in Figure 2.2. As observed verbal description creates much higher extraneous cognitive load than visual presentation of a square although result should be same; a person has clear understanding of what a square is. These two types of cognitive loads are additive, thus total cognitive load is sum of them and resources that we use to deal with them come from same working memory pool.

When talking about task difficulty we are dealing with intrinsic cognitive load and interactivity of items. Extraneous cognitive load affects to difficulty also but in a different way than intrinsic cognitive load and it can be modified to an extent, cf. description of a square, by instructional design. Based on quick thought it might be tempting to say that task with low intrinsic cognitive load is easy but that would be incorrect. Sweller et al. [104] p. 61 take an example of learning a new language vocabulary. When there is a low interactivity between items, words, it might require only a small working memory load, thus intrinsic cognitive load is low. Difficulty here comes from fact that there is a huge number of items to be learned, thus difficulty does not come from item complexity. Similarly, a small number of items but with high interactivity between them, cf. weekday problem, can be difficult but in this case difficulty comes from high interactivity between items, not from amount of them. Straightforwardly we can conclude that if there is large number of items and interactivity between them is high then learning this kind of material is, for most of us, unusually hard.

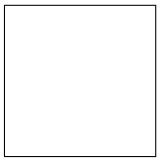

Figure 2.2: A square or a parallelogram that is both a rectangle and a rhombus, which is parallelogram with four equal sides.

*This is a part of instructional design that is also covered by e.g Sweller et al. [104] but is beyond scope of this thesis.

2.1.2 Measuring cognitive load

There are five main categories to do measurements of cognitive load as noted by Sweller et al. [104] ch 6: indirect, subjective, efficiency, dual-task methodology and physiological.

Indirect measurements were used early in the field as proxies. Its methods included computational models, performance indicators and error profiles. In computational models a language modeling cognitive processes, such as PRISM, see Klar, Langley and Neches [71], was used to create a suitable model, e.g. after novice and expert behavior as mentioned in Sweller [103] pp. 263-265, to obtain theoretical support for a study but models created were considered to have a limited use in estimating cognitive load. Performance indicators consisted of items like instructional time (Chandler and Sweller [23] & [24]) or error rates (Owens and Sweller [89]) that were hypothesized to increase with increase in cognitive load. This increase in load would have a marked effect on performance when learning a topic as with increased cognitive load both learning time and task acquisition accuracy were thought to have negative impact. Some early studies as noted by Sweller et al. [104] p. 72 supported this. Error profiles utilized error rates also and tried to find out places where most errors occurred in participant problem solving to indicate a location where cognitive load was highest. There were limited studies utilizing this method, but they appeared to provide evidence towards working memory demands.

Subjective measure asks participants provide response to suitable question in a rating scale that indicates cognitive load. Paas [91] performed a study in which he asked participants to report mental effort on a given task and Ayres [5] requested participants to rate task difficulty. Both studies found a match between test performance and rating provided by participant. While the two, mental effort and task difficulty, appear to be similar measurements and are typically used for same purpose, they are two different questions and might lead to dissimilar interpretations, thus obtained results might not be comparable as noted by Gog and Paas [48]. They indicate that mental effort relates more to a process, which includes given task, whereas difficulty relates to the task itself. Despite they do appear to be usable for purposes of measuring cognitive load. Sweller et al. [104] pp. 74-75 indicate that subjective measures have been very good and consistent in predicting performance data.

Efficiency measurement results are obtained via cognitive cost of learning. Its idea is that if two or more instructional methods cause same outcome in learning then one that requires least amount of cognitive resources is the most efficient. Paas and van Merriënboer [92] developed a relative condition efficiency measurement, which compared a z-score* of a test to z-score of mental effort. Results could be plotted on the two-dimensional Cartesian plane, which has an imaginary line with a slope of one corresponding to an efficiency score of zero, Figure 2.3. Positive values indicate method to be more efficient than one with negative values. While this measurement has been used in a lot of studies there are some concerns on it. Hoffman and Schraw [58] note that two different scores, which have different scales, need to be converted to some common scale and that might be problem since what is the meaning of two subtracted

*A term in statistics that is also known as standard score, z-value, normal score or standardized variable

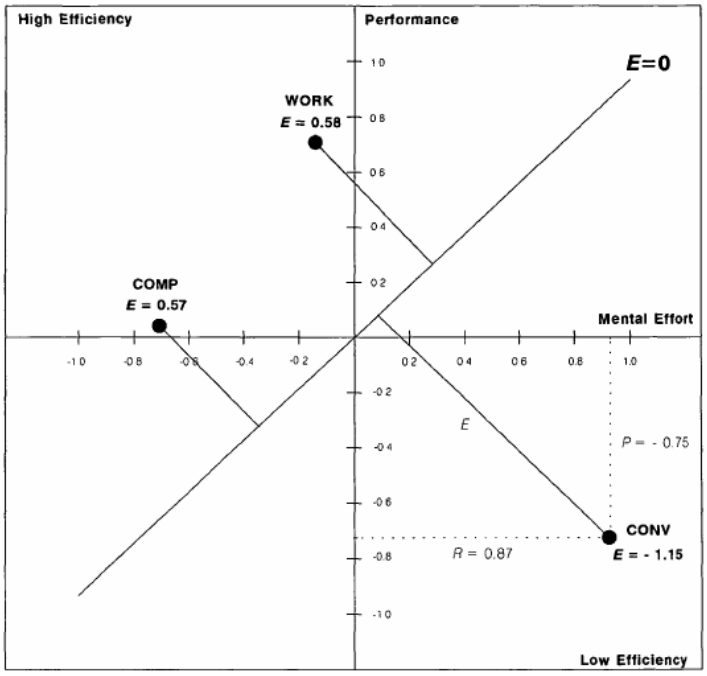

Figure 2.3: Condition efficiency measurement depicted as in Paas and Merriënboer [92] figure 2

variables that are dissimilar to one another conceptually. According to them we could compare person's height and an IQ but what would that tell us. Regardless of this Sweller et al. [104] p. 77 noted that it might be a good indicator.

Dual-task, or secondary task, measurements ask participants to perform a secondary task alongside to a primary task. Depending on load imposed by the primary task results on the secondary task might go down, which would indicate heavier load on the primary task, or up. Typically, the secondary task would differ from the primary task and requires very little of working memory in order to avoid it from distracting the primary task too much. Brünken et al. [18] utilized a single letter that changed its color on random intervals from black to red as a secondary task while participant primary task was to study the human cardiovascular system from multimedia presentation. Participants reaction time to the secondary task, i.e. noting when a letter color has changed to red, was an indication of cognitive load and the lowest load was on learning approach that produced the best performing results. Dual-task measurements require careful planning regarding to secondary task and it might also require additional equipment like when secondary task is auditory signal. Due to these extra requirements they are not as widely used as subjective measures. Its main advantage over subjective measurements is nearly continuous measurement of cognitive load while a task is being performed.

Physiological measurements include a wide variety of sources such as heart rate, brain waves, eye-related like movement or dilation and skin-related like temperature and emotions. Afergan et al. [1] utilized functional near-infrared spectroscopy (fNIRS), which detects hemoglobin oxygen levels, brain sensing to create a model that changed test scenario in real-time based on perceived difficulty to keep participants workload in comfortable level. They indicate that such real-time model was successful, thus concluding that they are certain of user state regarding to cognitive load. Kosti et

al. [72] used electroencephalography (EEG) recordings from brain to create a model predicting workload and concluded that it can show the true difficulty that subjects encounter. While results appear to indicate that it is possible to detect cognitive load levels using physiological indicators and those indicators could be good choice to subjective measures, it is still too early to conclude that according to Sweller et al. [104] p. 81. In addition, extra devices might not be usable in all circumstances as some, like functional magnetic resonance imaging, requires special equipment and subject to avoid any movement during measurements. Still modern devices like a multi-sensory wrist band* do provide interesting opportunities.

2.2 Difficulty in programming

In this section we will discuss programming related matters. We will peek at knowledge areas and concepts that beginning programmer will typically encounter as well as which of them students think are difficult. We point out why reading of a code is considered an important part of programming as is normally thought about writing of a code. We will cover briefly ways to estimate difficulty and note differences between complexity and difficulty. At the end we look one interesting new theoretical framework that might help in estimating difficulty once it is completed.

Programming like many other abilities belong to biologically secondary knowledge category of skills. Division between biologically primary and secondary knowledge was introduced by Geary [46] & [47] and difference between them is that the primary knowledge can be learned but not taught but the secondary knowledge can be learned and taught. An example of biologically primary knowledge is our ability to learn our first language by listening and speaking. These two knowledge are part of cognitive load theory, but they are more related to instructional design side of it. However knowing that biologically primary knowledge is something we acquire based on what we are after long evolution compared to biologically secondary knowledge, which we can absorb given that it is specifically taught, we put our effort and conscious to it, might help us to understand some part of differences between perceptions of difficulty. As secondary knowledge is explicitly taught and typically learned in deliberate and intentional way, it is futile to expect person to acquire such knowledge via immersion as noted by Sweller [104] ch 1.

2.2.1 Requirements for a programmer

Beginning programmer may have some basic knowledge of computers and how to use them to carry out basic tasks like writing text, sending an email, browsing web or playing games, or they might be self-taught programmers who can create software of their own. Regardless of their backgrounds there are various knowledge areas that they need to learn from computer science to move from novice towards becoming expert. These

*An example could be Empatica E3 that includes electrodermal activity sensor measuring the electrical conductance of the skin, photoplethysmography sensor illuminates and measures light from skin to observe e.g. oxyhemoglobin in blood, temperature sensor and a three-axis acceleration sensor, see Grabarino et al. [45].

knowledge areas include things like programming languages, operating systems, algorithms and complexity, discrete structures, information management, networking and communications, platform-based development and software development fundamentals as noted in Computer Science Curricula 2013 [67]. While some of these knowledge areas are directly programming related like algorithms and complexity, others are closely linked to it like networking and communications. These higher-level knowledge areas can be further elaborated to get an idea of how comprehensive they are. As the main interest here is related to programming we will concentrate more into that part.

From Luxton-Reilly et al. [79] we can obtain a list of common introductory programming course, which is typically called CS1 course in the USA and Canada but also in many papers, concepts: variables and assignment, data types, data and control structures, operations and functions, recursion, pointers and memory management, input/output operations, object-oriented concepts, libraries, programming processes and abstract thinking. These can be further divided into more fine-grained concepts like data structures consisting of arrays, matrices, lists, sets and relations, structures and records, linked list and collections other than arrays. Overall Luxton-Reilly et al. list consist of over seventy items. That is a long list of concepts to digest for beginning programmer in an introductory programming course that is anything from half of semester upwards.

When looking at the concepts it can be observed that some of them have very low level of abstraction like variables or constants whereas abstract classes or polymorphism are examples of high-level abstraction. In survey done by Lahtinen, Ala-Mutka and Järvinen [73] they asked students and teachers to rate twelve programming concepts to find out what is difficult and whether there were differences between students' and teachers' thoughts on these concepts. Results indicate that the most difficult concepts from students' point of view were pointers and references, error handling, recursion, using language libraries and abstract data types. Teachers appear to agree with this list save for order of concepts, but they think that all concepts in the study are more difficult to learn than what the students think. Piteira and Costa [94] conducted a replication study where results on the student's side were similar but due to omission of recursion, they observed parameters to take its place on the top list. Teachers thoughts were somewhat different as they rated pointers and references, parameters, structured data types, abstract data types and error handling to lead the list. In same way than in Lahtinen, Ala-Mutka and Järvinen survey teachers rated all concepts to be more difficult than what the students indicated them to be. Out of these concepts error handling requires student to have good and comprehensive understanding of large entity, which is challenging compared to grasping some details of whole. Similarly using language libraries means that the novice must do searching on his own from large amount of information. For example, in Java Platform Standard Edition 8 [65] there are 217 packages with over 4000 classes. Abstract data types, recursion and pointers and references are high level abstractions, thus being cognitively complex.

Some of these concepts have been studied on their own like recursion. Götschi, Sanders and Galpin [52] observed three different groups of students to form various mental models from recursion. In model termed copies, which is hypothesized to be a model that experts have, recursion creates new instance of itself, gives control to this instance and receives it back after instance terminates. Depending on group results

indicate that about a quarter, 26%, to a half, 50.6%, of students formed this model. In looping model recursion is deemed to be some sort of iteration that utilizes same instance of an object repeatedly, and result is obtained when base case is reached. This model does result correct outcome but is associated to be used by novice programmers. In step model recursion is seen sort of if-then-else construct and execution appears to do recursion either once or twice before obtaining results. In syntactic, or magic, model students do not appear to have clear idea of how program achieves its results, so it happens like magic, but they appear to recognize that there is something syntactical elements in algorithm that makes it recursive. In algebraic model student interpret recursion to be some sort of algebraic expression but without ability, in some cases, to include recursion or its result into it at all. None of these other models, meaning models other than copies and looping, and few others mentioned in the paper are deemed to be a viable model to build on knowledge and understanding of recursion as noted by Götschi et al.

To enable novices to learn these concepts they obviously need to be taught to them. In addition to learning theories novices should practice programming by reading and writing programs. While writing a program is easier to see as necessary part of becoming programmer, reading is also very important. Winslow [114] noted that there appears to be very little connection between one's ability to read a program and writing one. In a study done by Innovation and Technology in Computer Science Education (ITiCSE) working group by Lister et al. [76], they looked at students' abilities to read code in two type of cases. In the first type students were given short code fragment and asked its outcome when executed. In the second type they were shown a function that was missing parts of it and students were asked what pieces of code would complete it based on description about what function should do. An example of the first type of question is shown in Listing 2.1 and, as can be noted, they used multiple choice questions, so students had also possibility to make a guess. In this particular question idea is to do filtering by copying only those values from the first array that are greater than or equal to two to the second array, but it is noteworthy that the first element of the first array is ignored due to way starting index is initialized. This one was considered on easier side of the questions and out of those students who answered it 68% got it correct. Out of total of twelve questions they had, even the easiest question was answered correctly by 74% of students while on the hardest one that percentage was 38%. They concluded that abstracting the problem that needs to be solved from given description is the most difficult part for students.

```
int[] array1 = {2, 4, 1, 3};   int[] array2 = {0, 0, 0, 0};   int a2 = 0;

for (int a1=1; a1<array1.length; ++a1)
{
    if ( array1[a1] >= 2 )
    {
        array2[a2] = array1[a1];
        ++a2;
    }
}

After this code is executed, the array "array2" contains what values?
a) {4,3,0,0}    b) {4,1,3,0}    c) {2,4,3,0}    d) {2,4,1,3}
```

Listing 2.1: Multiple choice question #10 from Lister et al. [76].

Single loop	Clean first	Clean multiple

repeat until find sentinel {
  if next input is non-negative
  increment count
  add it to the running sum
}
if count is at least 1
  compute the average as sum/count
else report “no data”
	repeat until find sentinel {
	  if next input is non-negative
	  add to set of clean data
	}
	if at least one clean datum
	  count the clean data
	  sum the clean data
	  compute average as sum/count
	else report “no data”
		repeat until find sentinel {
		  if next input is non-negative
		  increment count
		}
		repeat until find sentinel {
		  if next input is non-negative
		  add it to the running sum
		}
		if count is at least 1
		  compute average as sum/count
		else report “no data”

Table 2.1: High-level rainfall problem structures as in Fisler [41]

In similar fashion to reading, writing a program might be challenging. Sometimes challenge may come from student's ability to combine simpler tasks into a bigger whole. An example of such a thing is a rainfall problem*, which requires one to write a program that reads non-negative integers until some stopping value is read and then output average of inputs. The problem goal can be divided into sub goals that are combined to form a final solution. Fisler's [41] division; *sentinel* that indicates end of input, *negative* that are values to be ignored, *count* that is number of inputs, *sum* that is total of inputs, *divzero* that handles special case where there are no inputs and *average* that is final output. Fisler presented a few pseudocode variants of possible correct solution as seen on Table 2.1. Seppälä et al. [99], which used above-mentioned sub goal division, reported results from three different universities. Percentage of students who provided completely correct solution varied from 45% to 72.1%. In discussion they note that students should understand the problem by reading carefully program specifications, attempting to find out possible special or corner cases and assessing their resulting program in addition to writing a program. Apparently not an easy task for novices.

2.2.2 Attempts in measuring difficulty

When reading various studies on programming difficulty it is not always clear whether programming is difficult or whether there are other reasons explaining obtained results. Luxton-Reilly [78] points out that children have been programming for several years quite successfully and that different countries are now including programming into curriculum for primary school children, so if children are able to learn to program why it would appear to become difficult at later age†. He questions whether it is expectations that instructors have for students that appear to make programming difficult instead of it being difficult. In multiple papers, he mentions, there are notes along the line “Do students in introductory computing courses know how to program

*There are numerous variations to this problem and good overview can be obtained from a study by Seppälä et al. [99]

†It needs to be noted that when evaluating children and adults they are evaluated quite differently. Typically young children studies are more experience type of studies rather than evaluating what they accomplished in a given time period. In those studies engagement into what is being done can be seen as learning in some contexts.

at the *expected* skill level?" (McCracken et al. [84]. Emphasis added.) Whalley et al. [113], who according to Luxton-Reilly [78] are expert teachers and active computer science education researches, admit that they had underestimated level of cognition in questions required in their study. As noted above there are a lot of knowledge areas and concepts to learn in introductory programming course. Teaching varies a lot even within a single country as can be noted from paper by Seppälä et al. [99]. In one university introductory course is seven weeks long, focuses on imperative programming paradigm in the first half but adds object-oriented paradigm later and has lectures with plenty of small programming exercises. In another university this course is nine weeks long, appears to be imperative programming paradigm only and has some programming exercises every week of the course. In the last university this course is fourteen weeks long, focuses only in imperative programming paradigm, utilizes a flipped classroom pedagogy and requires multiple small programming assignments to be solved weekly. The paper notes that on the first university there are a two-hour lecture weekly, in the second university there are four hours' worth of lectures per week and the third one does not specify any lectures.

Based on above mentioned and other studies attempting to find out answers related to difficulty, various methods are utilized. Lister et al. used results from multiple choice questions as noted above. They also utilized interviews and doodles, which are scratch papers for students to scribble on, but due to lack of resources and abundance of students they were able to include only a fraction of participants in them. Seppälä et al., as noted above, and McCracken et al. [84] tasked students to create a program. Kasto and Whalley [69] asked students to trace, cf. Lister et al., or explain in plain English given code to obtain difficulty ranking for that code. They calculated various software metrics from each code sample used in these exercises in attempt to find out if such metric would correlate difficulty of a given task. Fritz et al. [44] used psycho-physiological sensors including eye-tracking (pupil size correlate with cognitive load), electrodermal activity (EDA, indication of cognitive load or task difficulty level) and electroencephalography (EEG, predict e.g. working memory load). Many of these do not have direct correspondence to earlier mentioned cognitive load measurements save for psycho-physiological ones.

It is good to keep in mind differences between complexity vs. difficulty via software metrics. Software metric attempts to provide a standard measurement to what degree some software possess a given property. These metrics could be simple like amount of lines in code or more complicated like count of paths in a given program such as cyclomatic complexity, McCabe [83]. Studies using these metrics tend to measure and focus on software complexity, which could be defined as how difficulty code is to comprehend and work with as by Magel et al. [80]. But being complex does not necessarily imply being difficulty although it might appear so based on given definition of complexity and there is some intertwine between the two. McCabe keeps value of ten in complexity as upper bound on cyclomatic complexity and code that is more complex should be reorganized. To get an idea whether such a complex code is necessarily difficult we can look at Listing 2.2. Its complexity value is 14, which basically comes from number of control statements in it. Based on this metric given code is rather complex but when looking at function it does not appear to be that difficulty to understand. It receives an integer, which represents a month, and returns a name of

that month as a string. In a similar fashion number of lines might not tell much about difficulty as can be observed from Listing 2.3. Quite often a short code is thought to be easy but when it utilizes e.g. regular expression* it might become rather difficulty to understand what it does unless some time is spent to decipher it. However, software metrics can be utilized for studying difficulty as seen in e.g. Kasto and Whalley paper.

```
String getMonthName (int month) {
        switch (month) {
                case 1: return "January";
                case 2: return "February";
                case 3: return "March";
                case 4: return "April";
                case 5: return "May";
                case 6: return "June";
                case 7: return "July";
                case 8: return "August";
                case 9: return "September";
                case 10: return "October";
                case 11: return "November";
                case 12: return "December";
                default:
                    throw new IllegalArgumentException();
        }
}
```

Listing 2.2: Java method with cyclomatic complexity of 14

```
def validate_phone_number(nr):
  return re.search(r'
␣␣␣␣(\d{3}|\(\d{3}\))(:?\s+|-|\.)?(\d{3})(:?\s+|-|\.)?(\d{4})
␣␣␣␣', nr) is not None
```

Listing 2.3: Python method validating phone number in the US format using regular expression. Note that the code beginning with 'return' and ending with 'None' should be on single line but is split for formatting purposes.

Duran, Sorva and Leite [37] propose an interesting theoretical framework that provides presently two metrics from a program. The first one is a plan depth† that describes overall complexity of a program and it allows comparison between different programs utilizing it. They include a sample program‡, Figure 2.4, and its plan tree with plan depth that can be seen in Figure 2.5. The second metric is a plan interactivity that describes number

```
int i, input, sum;
sum = 0;
for (i = 1; i <= 10; i++) {
    read(input);
    sum = sum + input;
}
```

Figure 2.4: Summing program from Duran et al. [37]

*The regular expression is specifically crafted sequence of metacharacters, which have a special meaning, and literal characters that form a search pattern and that some program interprets. There are some variations on it as no standard exists.

†This term is their equivalent to action's "order" or "level" in the model of hierarchical complexity's (MHC), which is a "theoretical model for analyzing the complexity of actions within a domain" (Duran, Sorva and Leite [37]). More about MHC can be read from article by Commons et al. [33].

‡For full analysis see Duran et al. [37] 3.1.1 Case Study 1: Summing Program.

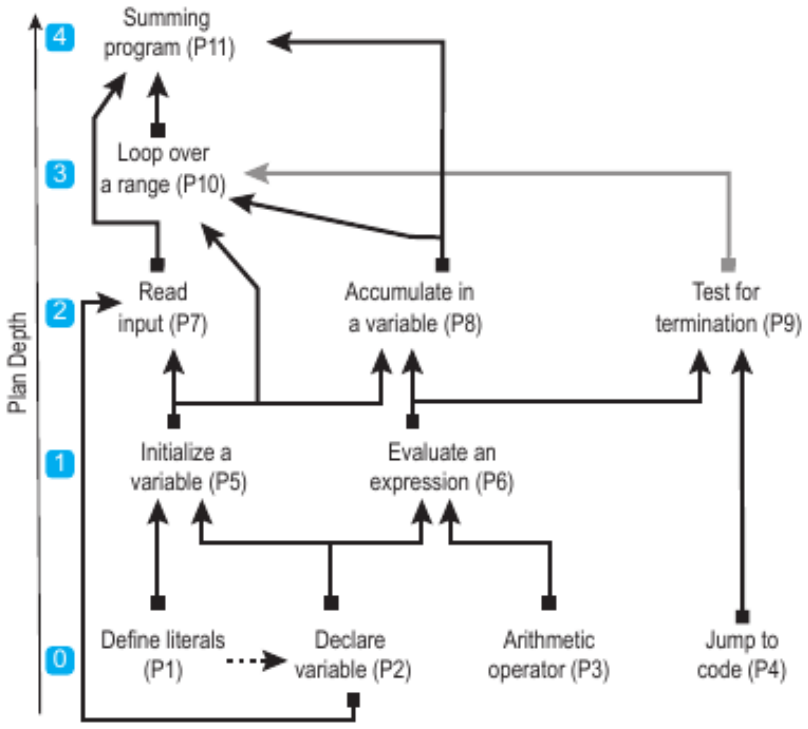

Figure 2.5: Plan tree with plan depth for summing program from Duran et al. [37] figure 2.

of schemata a programmer must hold in his memory at once. They provide comparison between two different ways that the rainfall problem can be solved and analyze both plans interactivity measurement*. The first one is called a merged-plans solution, Figure 2.6, and the second one is called a sequenced-plans solution, Figure 2.7. Each color in code represents a single schema. Based on their analysis the first solution maximal plan interactivity (MPI) is eight. This MPI means that programmers must hold in his working memory at most that many schemata at same time. So, this first solution MPI, as noted earlier, is pushing ones working memory capacity limits. The second solution MPI is only two, which leaves much more room in ones working memory. While this appears to be good solution for estimating cognitive load authors acknowledge that the framework is still work in progress and is missing, among other things, a well-defined analysis process.

*For full analysis see Duran et al. [37] 3.2.3 Case Study 3: Averaging Rainfall.

```
var count = 0
var sum = 0
var average = 0
while (true) {
    var input = readInt()
    if (input >= 0) {
        if (input >= 999999) {
            if (count == 0) { println(``No data!''); break }
            println(average);
            break
        }
        count += 1
        sum += input
        average = sum / count
    }
}
```

Figure 2.6: A merged-plans solution as in Duran et al. [37] figure 4.

```
def isNotSentinel (input : Int) = input < 999999
def isValid(input : Int) = input >= 0
def average(numbers : List [Int]) =
    if (numbers.nonEmpty)
        numbers.sum / numbers.size
    else 0
val validInputs = inputs.takeWhile(isNotSentinel).filter(isValid)
val averageRainfall = average(validInputs)
```

Figure 2.7: A sequenced-plans solution as in Duran et al. [37] figure 5.

2.3 Estimating difficulty

The field attempting to evaluate programming task difficulty using automated tools that use software metrics is somewhat newish. The study done by Alvarez and Scott [3] consisted of a small number of students from introductory programming course who completed six programming assignments and provided rating for each assignment difficulty in five-point rating scale before and after each assignment. They also calculated four metrics from source code: number of lines of code (LOC) excluding comments and empty lines, number of flow-of-control (FOC) constructs, number of function definitions and number of variables. Using mentioned metrics, the results from surveys and grades that students received they tried to find out factors that make an assignment difficult. Out of used software metrics LOC and FOC appeared to be moderately predictive when looking at either rating given before or after assignment. They conclude that as program gets longer, i.e. LOC increases, or more complex, i.e. FOC count increases, it becomes more difficult. They do point out that only averages over all students in results are predictable, not individual results. In addition, they note that students can make good judgment of given assignment difficulty after only reading it with variation being only 0.3 point in their estimate on difficulty in one way or another when compared to rating given after that assignment was done.

Ihantola, Sorva and Vihavainen [62] study included a few hundred students from introductory course as well. These students completed over one hundred programming assignments and provided rating for each assignment difficulty after completing it using five-point rating scale. They utilized LOC and FOC software metrics as well but in addition they used amount of time spent on given assignment, number of depresses of a key and percentage of those two in a non-compiling state. In their findings time correlated the most as a predictor of given task difficulty while LOC and FOC had notably lower correlation as far as individual factors go. When they combined factors used time was still dominating but other factors like program complexity or size was noticeable and confirmed positive correlation to student perceived difficulty.

A study by Francisco and Ambrosio [43] relates to developing an online judge system that would support teaching in introductory course. Their primary goal regarding to a difficulty rating is to get students point of view of it compared to the one provided by the teacher when he enters exercise into the system. This way the teacher would be able to better understand how difficult given list of exercises is and help students to select first easier ones before progressing to more difficult ones, thus reducing their frustration when doing exercises. In their system when student submits solution to given programming assignment, they also provide estimate on its difficulty in three-point rating scale from easy to difficult. Afterwards the system calculates various software metrics from submission including LOC, repetition structures, selection structures, few graphs, which represents the algorithm, related values like height of a tree and number of topics involved with the problem. Topics in their case are related to topics covered in the course such as input/output, selection, repetition and functions. They utilize these metrics to find correlations between them and perceived difficulty. The highest correlation they find is topics, which means that more complex topics that build on earlier ones tend to make given task more difficult for the students.

3. Making and evaluating machine learning model

In this chapter we will cover briefly machine learning, supervised machine learning workflow, how to select used method or algorithm, how to asses obtained model and finally introduce one class of supervised methods called decision trees. The section is based on synthesis from various books including An Introduction to Statistical Learning with Applications in R [64], The Elements of Statistical Learning: Data Mining, Inference, and Prediction [56], Machine Learning: A Probabilistic Perspective [88], Data Mining: Practical Machine Learning Tools and Techniques [115] and Machine Learning: An Algorithmic Perspective [81] and other sources as indicated.

Machine learning in computer science could be described as a field of study that utilizes a set of methods trying to detect patterns in data and utilizing these found patterns to predict future data. This is attempted to achieve with computer software that can learn these patterns autonomously. Two of the most common applications for utilizing machine learning are expert systems, which "uses artificial-intelligence methods to solve problems within a specialized domain that ordinarily requires human expertise" (Encyclopædia Britannica [116]), and data-mining, which is "the process of discovering interesting and useful patterns in large volumes of data" (Encyclopædia Britannica [29]). Machine learning tasks can be divided into two broad categories; unsupervised and supervised learning.

3.1 Building a machine learning model

In unsupervised learning we have collection of observations. Based on these observations we can try to find something interesting from the data such as are there any clusters, i.e. similar things occurring together, or anomalies in it. For example, in astronomy observations could consist of object's astrophysical measurements and based on them we could classify whether given object belongs to some existing type of star or have we found a new type of star like described by Cheeseman and Stutz [26]. Or in finance, credit card transactions would form observations that can be used to build a model that attempts to detect fraudulent transactions in order to prevent credit card fraud like noted by Brause, Langsdorf and Hepp [14].

In supervised learning we have similarly collection of observations but in addition to them we have, at least for some observations, outputs as well. Utilizing observations with outputs we can build a model that can be used with observations, whether existing or new, for which we do not have outputs to do a prediction task. If prediction is

categorical, nominal, variable, such as male or female, then it is termed classification or pattern recognition problem. If prediction is a real-valued scalar then it is termed to be regression problem. An example of classification problem is one where observations are a collection of handwritten digits that have label indicating what digit given observation represents. After training we obtain a model that would allow us to classify new handwritten digits like LeCun et al. [74] has done. Weather forecasting has a lot of possible regression problems like daily temperatures.

To begin building a machine learning model we need to have data. If data is unprocessed, raw, then we typically must do some processing for it for data to be usable, Figure 3.1. In this processing we need to find out what format data is presently and decide what format it needs to be for an algorithm, how to handle missing or incorrect values, what type of given variable is, what range it should be, remove outliers and so on. Once data is processed it will have inputs, which might be called with other names like features, independent variables or predictors, and possibly outputs, which can be called response or dependent variables.

After this processing we can do feature selection, which basically means that we select a subset of all possible features that are considered the most relevant for a given task. The main reasons for this selection are shortening models training time, models simplification for interpretation, enhancing generalization to reduce variance and avoiding the curse of dimensionality. The first two are quite straightforward, the third will be touched during discussion of variance later and the last one has to do with data sparsity. If we have data in a line, 1D space, it might be dense but when adding a new dimension, or feature, with same amount of data it becomes sparser and by adding another dimension the data becomes even more sparse, Figure 3.2. This issue with dimensionality is rarer in lower dimensions but is more typical when there are hundreds or thousands of dimensions, which occurs rather easily if not taken care of. In order to obtain statistically solid results, we need increasing amounts of data to support our results and this amount increases often exponentially when dimensionality increases.

One possibility to counter data sparsity is to use imputation. In imputation we use existing data but for cases where we are missing values we will replace them with some substituted values. This requires careful thought on questions like what to replace, how to do it, why data is missing and how that imputed data will be used, as without this process obtained results might be invalid or unusable. There exists various methods to do imputation depending on data. One method is mean imputation in which missing values are replaced with mean of that variable from other cases. Its benefit is leaving the sample mean as is but drawback is that it does not factor correlations between features. Another one is using the most frequent or some constant values. Its benefits include easy to do and works with categorical data but drawbacks include introducing bias. Imputation can be beneficial in other cases also. For example if we need to work with a dataset, which we have no control over, to perform a study utilizing it and obtaining our own dataset is not possible to do, we could impute selected parts to allow us to proceed with our study.

At latest in this point we tend to decide which one of above mentioned two categories of machine learning our task belongs to and start thinking about what kind of method we could utilize to investigate it. We would like to use the best possible

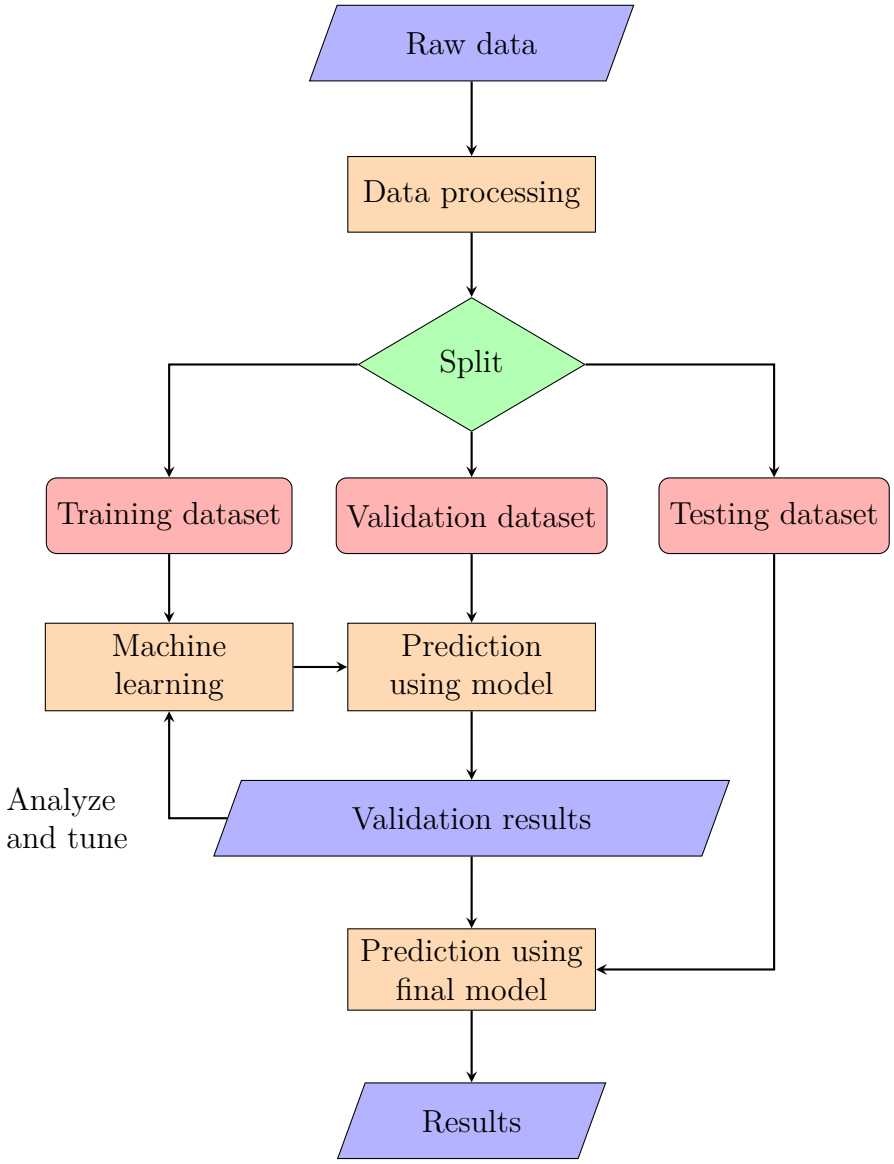

Figure 3.1: Generalized supervised machine learning workflow

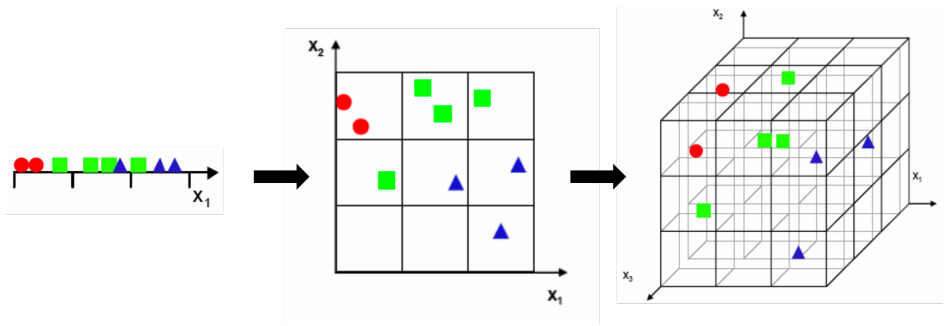

Figure 3.2: Same amount of data becomes sparser as number of dimensions increase. Picture by Nikhil Buduma [19].

method but there is no single one that would be the best solution to all possible problems. Basic reason for this is that behind every model there are a set of assumptions that work well in some domain but may work poorly or not at all in another domain. In a supervised learning, which is our interest in this thesis, we need to asses somehow how accurate a model is given our data in order to choose a good method. When we are creating a model based on data that we have we tend to split it into three parts as illustrated in Figure 3.1, which do not have to be equal in size. The first part is called a training dataset, which is used to create a model, so that methods we utilize will have some inputs with corresponding outputs to learn from. The second part is called a validation dataset, which is used to validate whether the model created during training phase works on unseen data and allows possibility to tune hyperparameters of the model created. Some algorithms require that certain parameters are set before that algorithm is executed and these parameters are called hyperparameters, which is to separate them from those parameters that we might derive during the training. The third dataset is called a test dataset and its purpose is to provide unbiased evaluation of the final model*.

Described validation set approach is quite straightforward to do but it has two issues that we need to be aware of. The first one is the test error rate, which can vary a lot depending on what observations are selected in training and validation datasets. It is possible that selected training data may predict validation data nearly perfectly, but opposite end is also possible outcome. The second one comes from fact that statistical models' performance is better if more data is used for training it. Since available data is used for three different datasets this means that if there is small amount of data this can reduce training dataset size considerably. There are methods to cope with these issues and one such method is cross-validation. In cross-validation we do not separate training and validation datasets but give them to the method as a single dataset. The method divides this dataset at random for suitable groups called folds and number of these folds, typically denoted with k, depends on option given to method, thus it is called k-fold cross-validation. The method picks one of these folds as a validation set, trains the model using remaining folds and finally calculates some metric like the mean square error for held-out fold observations. It then repeats this process k times while using always different fold as a validation set. In the end there are k such estimates and final estimate is computed by averaging these results. There is a special case where k is equal to number of items in dataset and that is called leave-one-out cross-validation (LOOCV). One of its benefit is less bias due to increased training data amount but drawback is potential consuming a lot of time on large amounts of data. Due to this k-fold cross-validation is often preferred in practice over LOOCV.

To asses our model accuracy we can use various measures. One of the commonly-used measure in the regression setting is the mean square error (MSE) and in the classification setting the error rate. In the former we calculate averaged squared prediction error, which is predicted output subtracted from known output and that is squared, and in the latter, we calculate proportion of errors made, which is sum of misclassified labels divided by the total amount of data to be classified. In either case

*In some literature terms validation and test datasets are swapped in their meanings or they are used as a synonyms to one another. In latter case they often refer to a test dataset as used in this thesis context and in that case a validation dataset is called a holdout dataset.

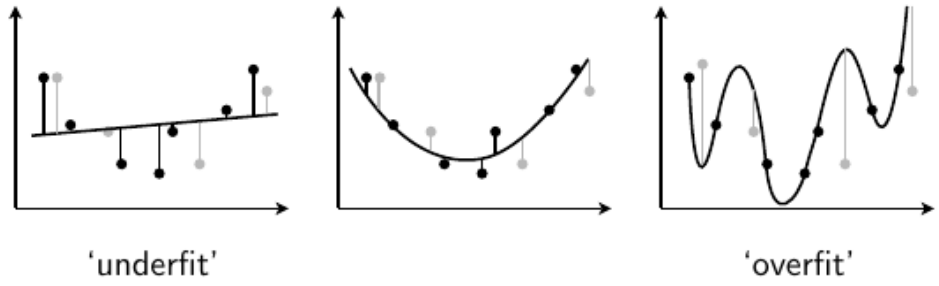

Figure 3.3: Various models fitting in regression setting. Picture from course material by Teemu Roos et al. [96]

the smaller the value we obtain by using validation dataset, the better we rate our model accuracy to be. Often this accuracy value is calculated also during the training phase but attempting to minimize it during that phase tends to result that created model will overfit the data. In other words, the model might be very good for purposes of training dataset, but we are really interested in results when the model is used for unseen data.

If we have observations like depicted using black circles in Figure 3.3 we can calculate the MSE for various models that we attempt to fit to it. In the left hand case we attempt to fit a linear model to observations and note that the MSE between observations and the model, depicted using gray color, is fairly good for some observations but rather poor for others and conclude that the model does not fit very well to observations, it underfit. In the right hand case, we are attempting to fit a very high degree polynomial model to observations and note that it will cover them all but the MSE is higher side like with a linear model, so we conclude that this model overfit. In the middle case we are attempting to fit parabola type of curve to observations, notice that it is quite close fit and the MSE is lowest of the three, so we have a robust fit.

In a supervised learning we need to be aware of few major issues when selecting an algorithm. The first one relates to variance and bias. If we have a model that is given an input of X and it predicts output of Y_1 when the model has been trained with one dataset but predicts Y_2 when the model has been trained with another dataset, we say that the model has variance for given input. If variance is high the model is typically complex. Even though it might represent training dataset quite well, there is a risk that it will overfit the model to some random or unrepresentative parts of data on the training dataset, Figure 3.4 on the right. If we use this model on validation or test dataset, we commonly get very poor results. On the other hand, if variance is low the model tends to be simpler, but it might miss some important factors on the data, which means it underfit, Figure 3.4 on the left. While it might produce some results with validation or test datasets, they tend to be meager at the best. If the model constantly predicts incorrect output for given input, then it is said to have bias. If the model is complex, representing training dataset more accurately but typically failing on other datasets, it is said to have low bias as visualized on Figure 3.4 on the right. In contrast relatively simpler models tend to have higher bias, as seen on Figure 3.4 on the left, but their predictions are of lower variance outside of the training data. For the best model we would like to have low variance and low bias at the same time but achieving that is not possible since when the variance increases the bias will decrease and vice verse. At some point in our model accuracy assessment we encounter a point

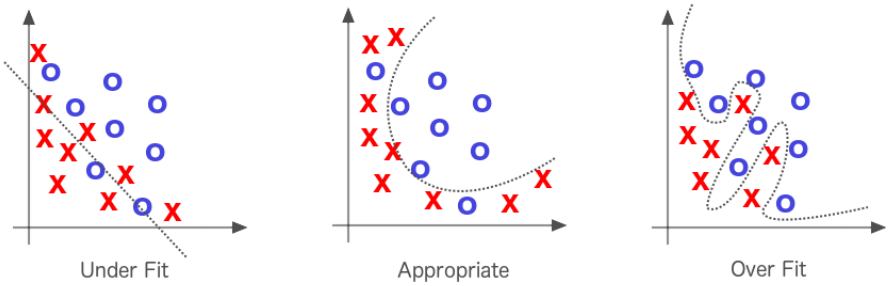

Figure 3.4: Various model fitting in classification setting. Picture from blog by Raheel Shaikh [100].

that is considered as a good tradeoff between the two, Figure 3.4 in the middle, and while not perfect it will provide good results on the unseen data.

The second issue is how complex the model is if we were to know the true model and how much data we have for training. If the model is complex, we would need large amount of training data to learn it, which mean that we would end up having low bias and high variance situation. In case the model is simple the results would be reversed. The third issue has to do with the dimensionality of input. If there are a lot of input features learning the model from them might cause the model to be overly complex, thus having high variance and low bias. To help with this issue we can use a feature selection to reduce number of input features. The fourth issue is related to issues with outputs, which might have noise in them due to errors in methods that are used to obtain data or due to ordinary human errors. In case there is a lot of noise in outputs and the model attempts to learn from it exactly the resulting model tends to be overly complex with high variance and low bias. To counter this we can try to e.g. remove noise from a training dataset.

There are other potential issues that need to be considered when algorithm is being selected including the data heterogeneity, which means features are diverse in e.g. their type, data redundancy, which means there are a lot of highly correlated features, or presence of interactions, which means that features have complex interactions with one another. Due to various needs and issues as noted, in machine learning there are quite a lot of different algorithms and myriad number of variations to them to improve one thing or another, to counter certain issues, to tune algorithm performance in selected cases and so on. To begin selection process of algorithms we can utilize information of them we can obtain e.g. from books, combine that knowledge with other information that we have like requirements what we need to accomplish and knowledge of data we have. For example, if we have a dataset that contain both inputs and outputs we can turn into supervised algorithms and if we have further knowledge that we would like to make predictions as to which of various categories output belongs to, we can check out multiclass classification algorithms. While this narrows number of algorithms there are still many remaining, so we might need further information, expert advice or utilize comparisons like one done by Caruana and Niculescu-Mizil [21] and based on this we might have a suggestion to investigate some algorithms like decision trees.

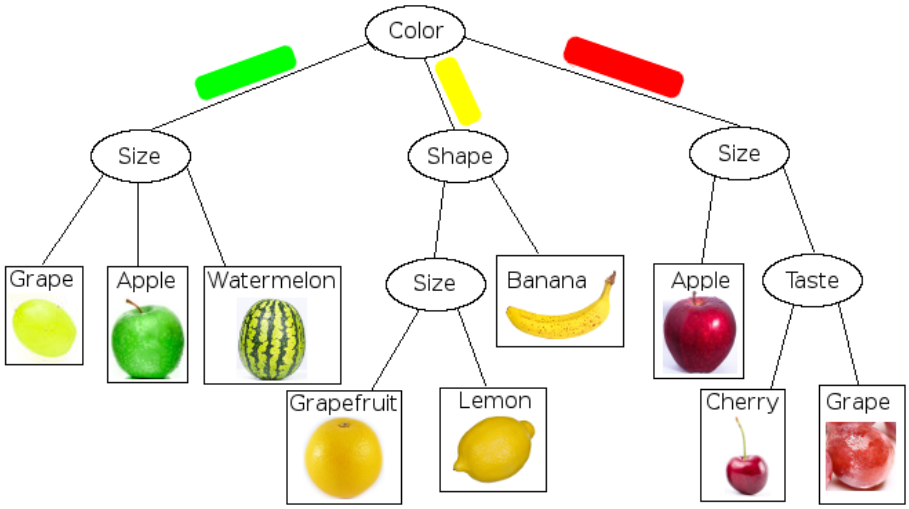

Figure 3.5: Very simple decision tree to recognize a fruit.

3.2 Decision tree

A simple decision tree is rather easy to understand, and many have encountered them in one form or another. They could be used for various things like to create steps to illustrate how some decision-making process works such as deciding what a given fruit is, Figure 3.5. This decision tree can be thought of as a visualization of how a person recognizes certain fruit or as how computer algorithm using machine vision performs same operation. By detecting one feature such as color from the inputs we have narrowed our selection of what type of fruit we are dealing with. Utilizing this same process repeatedly we pick up next feature, compare it to available options and proceed towards final decision, which is at the end of the chain. This type of decision tree can be turned into a set of if-then rules for a computer but obviously this becomes cumbersome when the tree of decisions grows bigger.

To construct a decision tree, we need to decide what is the first feature that we use. For that there are various metrics to choose from that depend on used algorithm and include things like Gini impurity, variance reduction and information gain. For illustration purposes we utilize information gain that uses information entropy, which is a measurement proposed by Shannon [101]. To get an idea what this measurement is we can think about having three containers with each of them having four colored balls as follows: in the first container all the balls are red, in the second container three balls are red and one is blue, and in the third container two balls are red and the other two are blue, Figure 3.6. If we are asked for what color of a ball we get when we pick it up randomly from the first container, we would know for certain that it is red as there are no other colored balls in it. On the other hand, on the second container case we know that with 75%, $\frac{3}{4}$, probability it is red and on the third container case it is red with 50%, $\frac{2}{4}$, probability. Now entropy is in a way opposite of this knowledge. It might be helpful to think this through entropy's definition, which is "the degree of disorder or uncertainty in a system" (Merriam-Webster [39]). In the first container case there is no uncertainty as to what color of ball we get if we pick it up randomly but as we

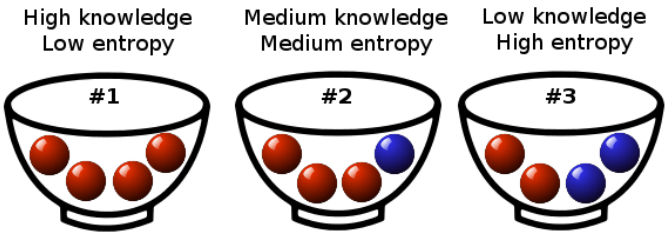

Figure 3.6: Entropy vs information via the balls in the bowls.

move to the second container case our uncertainty increases and in the third container case there is an even more uncertainty as which of two colors the ball is, Figure 3.6.

Constructing actual decision tree is a recursive process in which we select a feature, split data into number of subsets equal to number of values given feature has and then repeating the process. If at any given time all data belongs to a single subset then the process is done on that branch of the tree and continues on remaining ones. To decide what feature to use at what point we choose one, calculate the entropy of whole dataset and then subtract the entropy of subset based on selected feature. This difference is called information gain and needs to be calculated for all other features as well. For example, in pictured fruit detection we could choose color, shape, size or taste as a feature. If we picked size as the first feature, we need to perform this calculation for color, shape and taste as well. Whatever feature provides largest information gain is then selected as the feature of choice in a greedy manner, which means that we select feature that will provide the most gain always.

Other metrics beside just illustrated information gain work in similar fashion that some sort of estimate is obtained from various splits that features produce and then the most favorable one is picked as a selected feature and the tree construction proceeds. It would be ideal to choose the most optimal way to make partitioning but due to fact that such calculation is NP-complete* problem, Hyafil and Rivest [60], we must utilize some other way to produce good solution instead of the most optimal. One such practical method is to use heuristics like described. They do not necessarily produce globally optimal result but may produce locally optimal one and that is one limitation of decision trees.

Issues that need to be addressed here are bias and overfitting. They are typical issues with decision trees although not applicable to all algorithms. The bias comes when heuristic algorithm attempts to select a feature to use for splitting and it tends to favor those features that provide more split points as noted by Breiman et al. [17] p. 42 among others. In overfitting case decision tree becomes overly complex as it attempts to create a good match for the training dataset as noted earlier. To counter these we can attempt to tackle them individually such as using pruning for overfitting and adaptive leave-one-out feature (ALOOF) selection for bias or try to solve them with suitable selection of an algorithm. Pruning attempts to reduce created tree complexity by replacing nodes with the most common class if prediction accuracy is retained. Painsky and Rosset [93] ALOOF utilizes LOO score instead of training

*This is a term in computational complexity theory. NP stands for nondeterministic polynomial time and describes a problem that can be verified in relatively speaking quickly if we are given a solution but to find that solution speedily is unknown, i.e. with current knowledge it takes a lot of time to find optimal solution.

sample performance when selecting feature for splitting. Utilizing suitable algorithm depends on various needs but could include options like ctree by Hothorn, Hornik and Zeiles [59] or random forests by Breiman [16].

Ctree is easier of these two to understand as it merely has its own way to handle feature selection and splitting but is otherwise more like any other decision tree algorithm. In ctree a feature selection and splitting of dataset are two separate procedures. This will allow it to construct of interpretable tree without bias towards many splits or missing values. It also allows use of a statistically solid and intuitive stopping criterion. It uses a null hypothesis of independence between output and any of the inputs and in case that cannot be rejected at some given level α it will stop. So, it will handle noted issues and produce comparable results to other more elaborate decision trees as shown in the results of Hothorn, Hornik and Zeiles [59].

To begin working with random forests we need multiple datasets of same size than our training dataset. Since it is unlikely that we would have such datasets available to us we will sample original training dataset with replacement, so we might end up having some data multiple times in a new dataset and others not at all. Resulting sample is known as a bootstrap sample. Once we have multiple of these, at least tens or even thousands, datasets we fit equal number of models to them and get results from unseen data by combining them either by voting, if we are doing classification task, or by averaging output in case of regression task. That is, we are aggregating results. This whole process is known as bootstrap aggregating, or bagging, as introduced by Breiman [15]. Idea behind this is to reduce variance without affecting bias. As noted, a single decision tree might be overly complex, thus it has high variance and low bias but when we average over multiple trees, a forest, this issue is reduced if trees are not correlated, which is what bootstrap sampling attempts to do. However, this alone might not produce wanted results as if there is small amount of very strong predictors for outputs then it is very likely that these predictors will be selected by many of the trees that we have created and that makes them correlated, which is problematic. To counter this we select randomly a subset of features that will be used any time there is a split in a tree, so that we can avoid issues with strong predictors and correlation.

These processes introduce two new parameters that we must deal with - how many features per tree and how many trees in the forest. It appears that random forest is not very sensitive to number of features, so there are some rule of thumbs indicating that a square root of features is suitable for classification and one third of features is good for regression, as in Hastie, Tibshirani and Friedman [56] p. 592. The other parameter, number of trees, depends on our needs but typically we keep adding more trees until error stop decreasing. Typical choice for this error is to utilize out-of-bag (OOB) error estimate. When we use a bootstrap sample, we end up using roughly two-thirds of the original data, Aslam, Popa and Rivest [4], and rest are referred as the OOB observations. To get a single prediction for given observation we will average predicted response, in regression, or take a majority vote, in classification, from those predictions that were produced by trees that given observations was OOB. After we have obtained predictions for all possible observations, we calculate MSE or classification error and obtain out-of-bag error, which performance is nearly identical to cross-validation error, Bylander and Hanzlik [20].

	Reference				
Prediction	*1*	*2*	*3*	*4*	*5*
1	1	0	0	1	0
2	2	2	2	0	0
3	1	3	4	1	1
4	0	0	1	1	0
5	0	0	0	0	1

Table 3.1: An example of a multiclass confusion matrix for five labels.

3.3 Metrics for evaluating a model

Like noted earlier once we have our model, we want to measure how well it performs but used measurements could vary with the case. As an example, we take a case where we are predicting whether some instance belongs to one of multiple classes. For illustration purposes we will be using confusion matrix shown in Table 3.1. Confusion matrix is a table layout that enables us to show visually how selected algorithm performs. Each row shows instances of predicted class and each column represents instances in an actual class. This order of what row and column represents can be swapped but it needs to be known as it affects how to read it. In our example we can observe that out of four instances of 1's our system predicted one correctly, two were predicted to belong to class 2, one belonging to class 3 and none were from other remaining two classes by reading values from the first column.

One commonly used metric is accuracy, which is a number of correct results among all possible results, and we can obtain it by summing up diagonal numbers and dividing that number by sum of all values, i.e. $\frac{1+2+4+1+1}{21} \approx 0.429$. Accuracy gives overall effectiveness of a classifier, but it might be misleading, which we will discuss later. Despite of this it is still quite widely used and easily understood by many. Due to its scale being from 0 to 1 many tend to think it as a percentage value between 0% and 100%. In case instances are unevenly distributed it might be more useful to shift attention to average accuracy instead. In this case we change problem view to multiple binary classification tasks, so that each one looks at how classification is done for one class and remaining classes are grouped as another class. Table 3.2 show how these matrices would look like in our example and Table 3.3 shows them summed up. From the summed up table we can compute average accuracy similarly than how normal accuracy is computed and the result would be $\frac{9+72}{105} \approx 0.771$.

Another commonly used metric is F1 score, which is harmonic mean or weighted average of precision and recall. Precision is the fraction of correct predictions for a certain class, e.g. in our example for label 3 it is $\frac{4}{10} = 0.4$. Recall is the fraction of instances of a class that were correctly predicted, e.g. for label 3 it is $\frac{4}{7} \approx 0.571$. Ideally, we would like both precision and recall being high but sometimes one of them is considered more critical than the other, thus keeping them separate might be correct choice. In case neither one of them is more important than the other we can settle for a single number, which is what F1 score does. F1 score is calculated using formula $2 * \frac{precision * recall}{precision + recall}$,

	Reference	
Prediction	*I*	*II*
I	9	12
II	12	72

Table 3.3: An example of a multiclass summed one-vs-all confusion matrix.

	Reference	
Prediction	*1*	*2 + 3 + 4 + 5*
1	1	3
2 + 3 + 4 + 5	1	16

	Reference	
Prediction	*2*	*1 + 3 + 4 + 5*
2	2	3
1 + 3 + 4 + 5	4	12

	Reference	
Prediction	*3*	*1 + 2 + 4 + 5*
3	4	3
1 + 2 + 4 + 5	6	8

	Reference	
Prediction	*4*	*1 + 2 + 3 + 5*
4	1	2
1 + 2 + 3 + 5	1	17

	Reference	
Prediction	*5*	*1 + 2 + 3 + 4*
5	1	1
1 + 2 + 3 + 4	0	19

Table 3.2: An example of a multiclass one-vs-all confusion matrices.

	precision	recall	F1
1	0.500	0.250	0.333
2	0.333	0.400	0.364
3	0.400	0.571	0.471
4	0.500	0.333	0.400
5	1.000	0.500	0.667

Table 3.4: An example of a multiclass precision, recall and F1 score for five labels.

see Table 3.4 for values in our example. Similarly, to average accuracy we can also calculate micro-averaged F1 score utilizing combined confusion matrix shown in Table 3.3 to obtain value of 0.429. This micro-averaged score favors classes that have higher number of instances. F1 score has its own pitfalls like not utilizing information of correctly classified negative instances or giving equal importance to precision and recall, as typically different type of incorrect classifications have different costs as noted by Hand and Christen [54].

If used data is imbalanced one might have a thought whether obtained models are any better than random guess or majority class classifier results. Random guess classifier can be thought of having a dice with sides equal to number of classes and then throwing that dice to obtain results. More formally it is a probability of chosen label to be correct or a proportion of instances belonging to a class. Since our example has five classes random guess classifier accuracy is $\frac{1}{5}$. Majority class classifier looks at what class is the most dominant in given datasets and tries to predict that one class for all instances. In our example that would be class 3 and its accuracy would be $\frac{1}{3}$. Neither one of these two metrics are very useful as is, but they serve as a reference to cases where earlier obtained or other metrics would indicate that the model performance is poor. In our example we can observe that both the model accuracy and average accuracy are above either one of these two accuracies.

Some measures try to solve above mentioned as well as other issues such as correlation coefficient introduced by Matthews [82], which is termed as Matthews correlation coefficient, MCC. It uses all values from confusion matrix, and it is deemed as the most suitable metric for imbalanced data in comparison to accuracy, F1 and

	Reference	
Prediction	*Healthy*	*Disease*
Healthy	980	8
Disease	10	2

Table 3.5: Confusion matrix from a model attempting to predict some disease.

AUC* by Boughorbel, Jarray and El-Anbari [13]. MCC is a value from -1 to +1, which in binary classification case has interpretation as -1 means negative correlation, 0 means no correlation and +1 means complete correlation. In multiclass case as described by Gorodkin [51] the minimum can take values arbitrarily close to -1, negative values obtained do not carry exactly same meaning than in binary case and +1 remains as maximum value. Since MCC utilizes everything from confusion matrix it is slightly more complex to calculate than accuracy or F1 score especially in multiclass case where formula is $\frac{\sum_k \sum_l \sum_m C_{kk} C_{lm} - C_{kl} C_{mk}}{\sqrt{\sum_k (\sum_l C_{kl})(\sum_{k'|k' \neq k} \sum_{l'} C_{k'l'})} \sqrt{\sum_k (\sum_l C_{lk})(\sum_{k'|k' \neq k} \sum_{l'} C_{l'k'})}}$. In previous formula C is confusion matrix of size K * K and in our example, it produces of value 0.231 as MCC.

To get an idea why accuracy or F1 score might be misleading consider a case where something like a disease is quite rare, say only 3% of patients have it. Table 3.5 shows confusion matrix of how some model might make its predictions on this disease. We will notice that out of 1000 patients it classified 982 of them correctly, i.e. 980 as healthy and 2 having disease. From the remaining patients the model indicates 10 healthy as having disease and 8 being healthy even though they have disease. Accuracy for this model is 0.982 and F1 score is 0.991 but MCC is only 0.174. If looking at only accuracy or F1 score one could think that used model is performing excellent job but MCC reveals its performance being close to a random guessing.

*AUC stands for area under receiver operating characteristic (ROC) curve. ROC is a graphical presentation of two type of errors a binary classifier does for all possible thresholds and AUC summarizes overall performance of a classifier.

4. Research questions and methods

Past research on this field has suggested a model that uses all course assignments features to predict given assignment perceived difficulty. In this thesis we will try to improve that model idea by creating two models. One of them attempts to predict assignment difficulty by utilizing only given assignment features and leaving all other features out. The other one utilizes all those assignment's features that lead to current assignment including its features in its attempt to predict that assignment difficulty.

This chapter describes context where used data comes from and looks into that data. Then we we present our research questions and methods.

4.1 Context

The dataset used here is the original one from study by Ihantola, Sorva and Vihavainen and can be found at http://bit.ly/1oZnEKG. The dataset is from an introductory course to Java programming, which covers both procedural and object-oriented programming paradigms. The course was held in Spring of 2014 at the University of Helsinki. The course details including course organization, exercise material and how tests are crafted can be found from paper by Vihavainen, Luukkainen and Kurhila [108]. The course utilizes an extreme apprenticeship (XA) method as its pedagogical approach as described in paper by Vihavainen, Paksula and Luukkainen [109] and it is based on cognitive apprenticeship method, see Collins, Brown and Newman [32] & Collins, Brown and Holum [31]. The XA method two core values are practicing the craft in question to become a master of it and receiving continuously feedback that helps a student to know that he is progressing and that progress is towards desired direction. Due to this there are a lot of programming assignments done during the course. Programming work is done using industry-strength programming tools, which provides students experience what they would obtain when working in the field using same tools than professionals. Scaffolding used to obtain data is described in paper by Vihavainen et al. [110]. Exact processing done to originally collected data to produce dataset used here is undocumented by authors.

4.2 Data

There were 417 students attending to the course who provided a total of 31255 submissions and out of those submissions 11161 contained difficulty rating. There are total of 108 assignments. One assignment, #72, consist of three sub-assignments, which are arranged so that there is no assignment #72 at all but assignments #72.1, #72.2

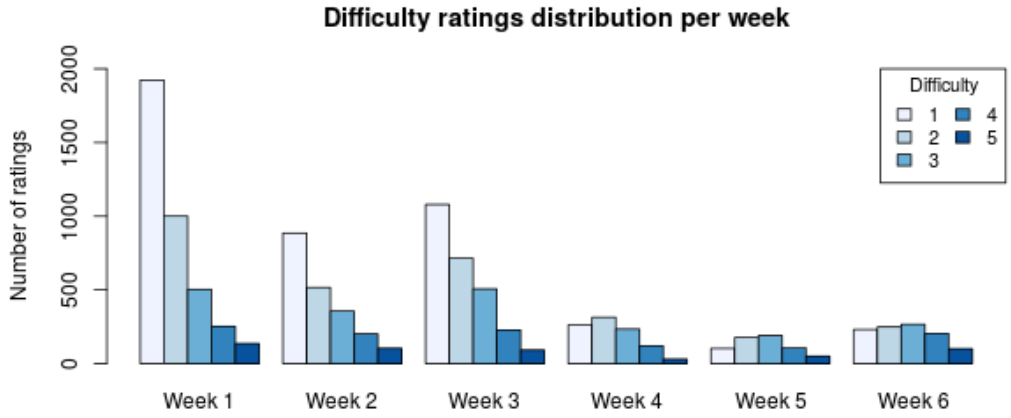

Figure 4.1: Distribution of difficulty ratings over course weeks.

and #72.3 form it. The second last assignment, #107, is placeholder for course feedback, and does not contain actual work at all, thus there is no interesting information associated with it and it is omitted from the models.

The data contains some generic information including user identification in form of universally unique identifier (UUID*), gender, year of birth and whether user had previous programming experience or not. This generic information is used in the original paper for looking at how learner's programming background affect on perceived difficulty and analyzing individual factors affecting on perceived difficulty. Rest of the fields are summarized in Table 4.1 and they exist for each assignment. Given exercise difficulty is a numeric value on scale from one, easy, to five, hard, as reported by students but providing it was voluntary for them.

Table 4.2 shows breakdown of different difficulty values over all assignments in given week and Figure 4.1 gives an overview of them. These values are totals that the dataset holds but as noted later not everything is used. Based on this information the first four weeks exercises appear to be easier for majority of students whereas the last two weeks distribution of ratings given seem to even out albeit the highest difficulty rating is still less prominent than others.

4.3 Research questions

The first part of this research is re-analysis of results regarding to combined factors from the paper by Ihantola, Sorva and Vihavainen [62]. The term re-analysis, which comes from the R.A.P. taxonomy suggested by Ihantola et al. [63], means that new researches duplicate the analysis done earlier utilizing the original dataset and methods. The second part is to develop models in order to study whether such models would be better in attempt to answer part of the original research question; "How do ... automatically analyzable programming behavior relate to the perceived difficulty of different programming assignments?" (Ihantola, Sorva and Vihavainen [62]). Research

*UUID consist of a 128-bit number. It is also known as GUID, which stands for globally unique identifier. See e.g. Wikipedia universally unique identifier.

Variable	Description
Compiles2	Proportion of time during which the program compiles. *Computed*
Flow control element count	Amount of flow control elements (if, while, for, ...) in the solution
Lines of code	How many lines of code the students last submitted solution have
Number of states	Number of different source code states that the student visited *Number of keystrokes*
Percentage compiles	Percentage of states that the student was in that compiled *Compiles1 variable in ctree*
Seconds in compiling state	How many seconds did the student spend in a state that did compile
Seconds in non-compiling state	How many seconds did the student spend in a state that did not compile
Time	Total time spent on the exercise. *Computed*
Difficulty	Difficulty of the exercise as indicated by the student *Attempted to predict*
Seconds spent on exercise	Number of seconds student spent on the exercise *Unused in models*
Educational value	Educational value of the exercise as indicated by the student *Unused in models*
Submitted	Whether student submitted the exercise *Unused in models*
Worked on exercise	Whether student worked on the exercise *Unused in models*

Table 4.1: Variables in dataset and those that are computed during the model creation.

	Difficulty value					
Week	*1*	*2*	*3*	*4*	*5*	Totals
#1	1924 (50,38%)	1001 (26,21%)	504 (13,2%)	253 (6,62%)	137 (3,59%)	3819
#2	886 (42,87%)	517 (25,01%)	357 (17,27%)	202 (9,77%)	105 (5,08%)	2067
#3	1080 (41,15%)	716 (27,28%)	507 (19,31%)	229 (8,72%)	93 (3,54%)	2625
#4	263 (27,25%)	315 (32,65%)	235 (24,35%)	121 (12,54%)	31 (3,21%)	965
#5	103 (16,32%)	178 (28,21%)	191 (30,27%)	108 (17,12%)	51 (8,08%)	631
#6	232 (22,01%)	249 (23,62%)	267 (25,34%)	205 (19,45%)	101 (9,58%)	1054
Totals	4488	2976	2061	1118	518	

Table 4.2: Distribution of difficulty ratings given over the course.

questions that are explored here are

RQ 1. How do re-analysis results compare to the original results published in paper by Ihantola, Sorva and Vihavainen and what problems were noticed in this re-analysis?

RQ 2. How will model that is assignment specific compare to original model?

RQ 3. How will model that is assignment specific with history information compare to original model?

The research question 1 (RQ1) will be answered through following the methodology described in an article 'Automatically detectable indicators of programming assignment difficulty' by Ihantola, Sorva and Vihavainen, writing up the issues raising up in the re-analysis, studying the results from re-analysis and contrasting them with the original results. To answer to the RQ2 we will be constructing an assignment specific model using random forests machine learning method that utilizes original dataset, looking at the obtained results and comparing them to the course model results. In assignment specific model we will be utilizing data that consist of all features related to single assignment. The course model utilizes all assignment features expect for their difficulty values but it does include currently handled assignment difficulty value. The course model idea stems from the study by Ihantola, Sorva and Vihavainen. The RQ3 will be handled similarly to the RQ2 but the model constructed will contain also history information, i.e. features, of those assignments that lead to and include assignment in question, so it is an assignment specific model with history.

4.4 Research methods

For the RQ1 we use same variables that were used in the original paper when researches were looking into combined factors. These variables, which are described in Table 4.1, are percentage compiles, number of states, lines of code, flow control element count, compiles2 and time. As in the original paper compiles2 is computed by dividing seconds in compiling state with total time. While collected data contained seconds spent on exercise, i.e. total time, information as a separate variable, it is calculate by combining seconds in compiling and non-compiling states. Even though we verified that adding up these two numbers result seconds spent on exercise value, we decided to keep the original way in order to use exact same process than in the original research.

For the RQ2 and RQ3 models we omit generic information from our datasets since it consist of user specific information and we are interested in finding out whether assignment specific features can be used to predict perceived difficulty of some assignment. In addition, all the models use these same six features as described above. The fields submitted and worked on exercise were omitted. Finally, educational value was also omitted since it is something that student provides and is something that could be attempted to predict like perceived difficulty.

Selected features are based on following reasoning. Out of used variables LOC and FOC are noted to be reasonable indicators of perceived difficulty by e.g. Alvarez and Scott [3] and Scott [98]. Ihantola, Sorva and Vihavainen note that keystrokes

and information being in non-compiling states can be possible signs of students having issues with a given assignment. By spending longer time than others on average in a state where the program does not compile, might indicate that assignment is more difficult for that student than for others. In similar manner if student uses more steps compared to others on average without getting the program to compile it could indicate that given assignment is more difficult for that student. Total time spent on exercise needs to be also compared to others on average as exercise complexity increases towards later exercises and so does total time spend for them.

We set a criteria that an assignment to be included in our dataset must have each difficulty value given by at least three students. This allowed distribution of data between three datasets as described in section 3.1. There are 15 assignments that are missing one or two of possible difficulty values. There are total of 60 assignments where at least one difficulty value has been given at most two times. This leaves total of 49 assignments where each difficulty value has been given three or more times. Number of submissions that contain all used variables including difficulty rating varies from 47 to 190 as show in Figure 4.2. If only those assignments are considered where each difficulty value has been given at least three times this count varies from 57 to 171 submissions. No single student has done all assignments in a manner that there would be information of all used variables including difficulty rating.

Due to low number of assignments we decided to do data imputation, which is briefly covered in section 3.1. Since our main interest here is estimated difficulty value that is also one being imputed. In order to do that a method producing a class value is needed. There are variety of suitable algorithms to do that but due to small amount of data none of tried algorithms appeared to produce strong results. In the end four different methods were chosen for the job: a decision tree classifier, a complement naive Bayes classifier, a gradient boosted regression tree classifier and a deep learning. A grid search was performed for the classifier methods to find out most promising hyperparameters to be used for the job. Similar type of operation was done for the deep learning model although the model architecture was based on experiment of few variants of simple network structure. Each assignment was handled separately due to variations between them. The dataset for given assignment was split between those that contained all features including predicted one and those that contained all features except for one being predicted. After this data in the first set was split into two sets, training and testing. Training dataset contained 70% of all samples and testing dataset remaining 30% of samples. Each model was trained over multiple times utilizing same dataset for all models but changing it between training iterations. The deep learning model was run for five hundred epochs for each training iteration. After the models were trained, they predicted testing set difficulty values and a model that had highest accuracy score was selected as winner for given iteration. Once all iterations were done there were multiple models that were rated based on their accuracy score and the top five models were selected. Then final prediction for difficulty value was obtained using majority voting over these selected models. After imputation there were 54 assignments that had at least each difficulty rating value three or more times and number of submissions varied between 167 and 396, which is shown in Figure 4.2. All imputed results were stored in separate file and those values were utilized only for training of various models, not for validation or testing purposes.

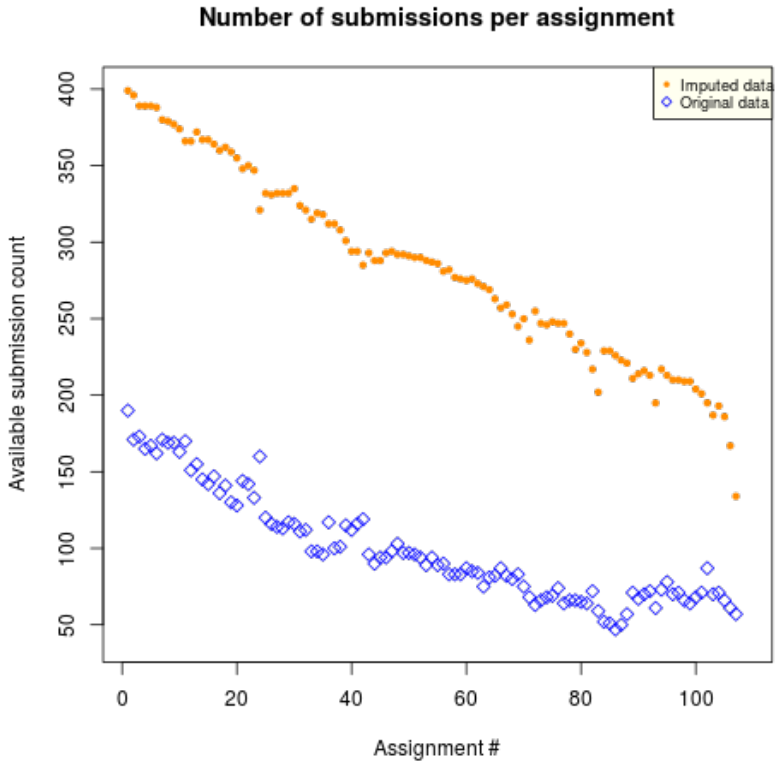

Figure 4.2: Number of samples that contain information about exercise difficulty and where such information has been imputed.

5. Results

In this chapter we will discuss the results that we obtained. We will cover our re-analysis first along with some notes on it. Next, we will discuss each of the models that we created along with what various metrics of those models tell us. Finally, we will combine model metrics together, so that we can make easier comparison between them.

5.1 Combined factors re-analysis

To answer RQ1 first part as how re-analysis results compare to the original results we will compare decision tree produced by re-analysis as seen in Figure 5.1 and original decision tree, which is included here for easier comparison taken from Ihantola, Sorva and Vihavainen paper, as shown in Figure 5.2. We can observe that the new decision tree is nearly identical to the original one with slight differences in time values on the root and on the right branch of the tree. On the left branch we can observe single LOC appearing where in the original tree no such split is noted. On the right branch FOC split is absent in the new decision tree and compiles1 has changed its place. Both decision trees have a range of difficulty values depicted at the leaf nodes. Their ranges as well as sizes are close to one another but due to some variations in split points there are small differences between them. Based on this we can note that using original data and methods we can duplicate results to some extent from the original paper regarding to combined factors.

To answer to the second part of RQ1 we discuss observed issues encountered in our re-analysis. During this re-analysis it was found out that there are some things done in production of the original decision tree that are undocumented in the paper or in provided code. One of the main issues is that neither sources reveal exact information about how to create ctree presented in the original paper. When looking at the scripts found from the data package there is no visible call to ctree, which means that exact formula, dataset given to it, possible subsets or weights and other options are unknown. Two of the original authors indicate that it is unlikely that any subsets or weights were used and most likely same goes with other options, thus this still leaves exact formula and used dataset as unknowns. Based on assumption that formula is basically utilizing difficulty vs. all other variables is quite logical and one that is used in this re-analysis. Another one of the main issues relates to used dataset. There are seven scripts provided that appear to be creating this dataset but there are subtle differences in them such as what fields are computed. None of these scripts are documented at all. Based on some investigation the script #7 appears to be the most likely candidate for producing

used dataset except that there are some additional fields that do not appear in the results, thus it is assumed that these are dropped at some point from the dataset that is used to produce original ctree. Yet another one of the main issues is omission from anywhere a fact that the first ten assignments data do not appear to be used at all in presented ctree. It means that over 40% of first week assignments are omitted from the model. While none of the scripts creating dataset used those first ten assignments presumption is that all assignments are used in the final dataset when creating the ctree shown in the paper. However, when using full dataset the results, are different, Figure 5.3, compared to case when those first ten assignments are omitted and is, as noted, quite similar to the original one. With all the data used time is still the single most dominant factor but there is increased effect from other factors such as number of keystrokes and FOC.

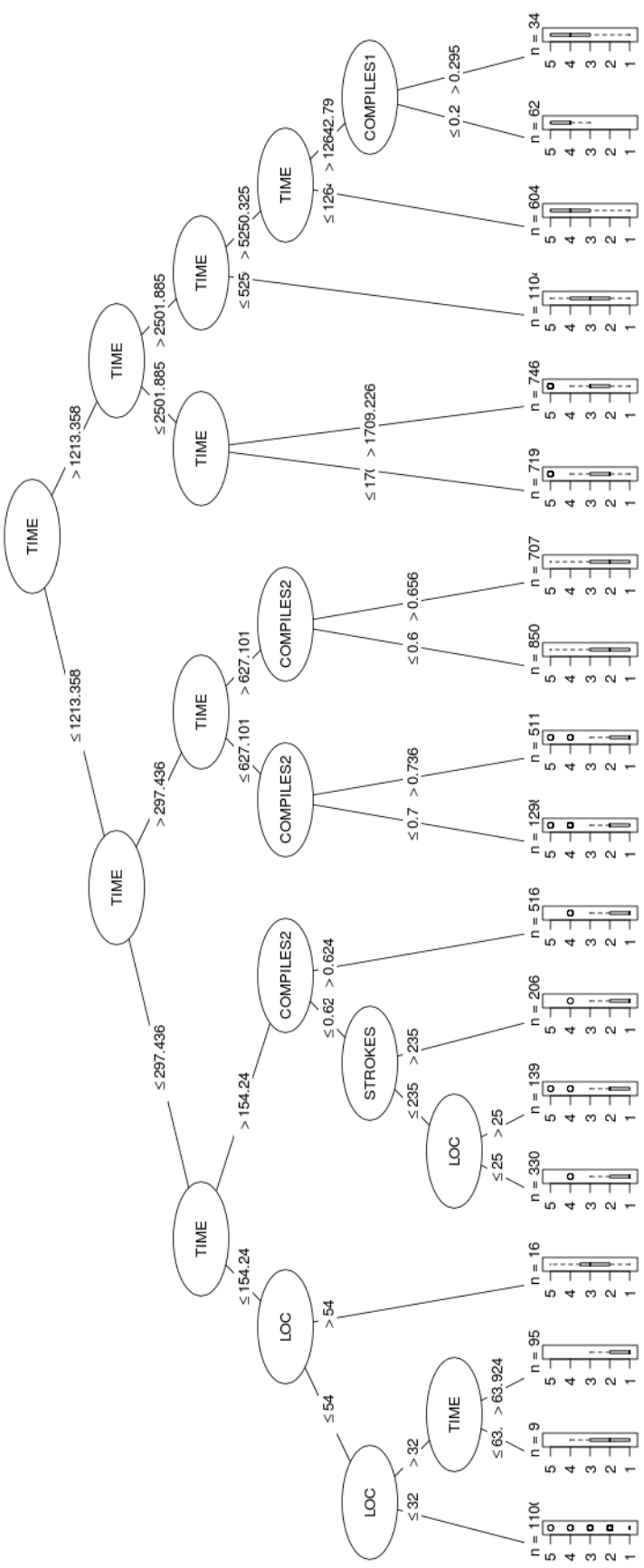

Figure 5.1: Decision tree obtained in re-analysis after noticing unmentioned twist in original decision tree and omitting first ten assignments from dataset.

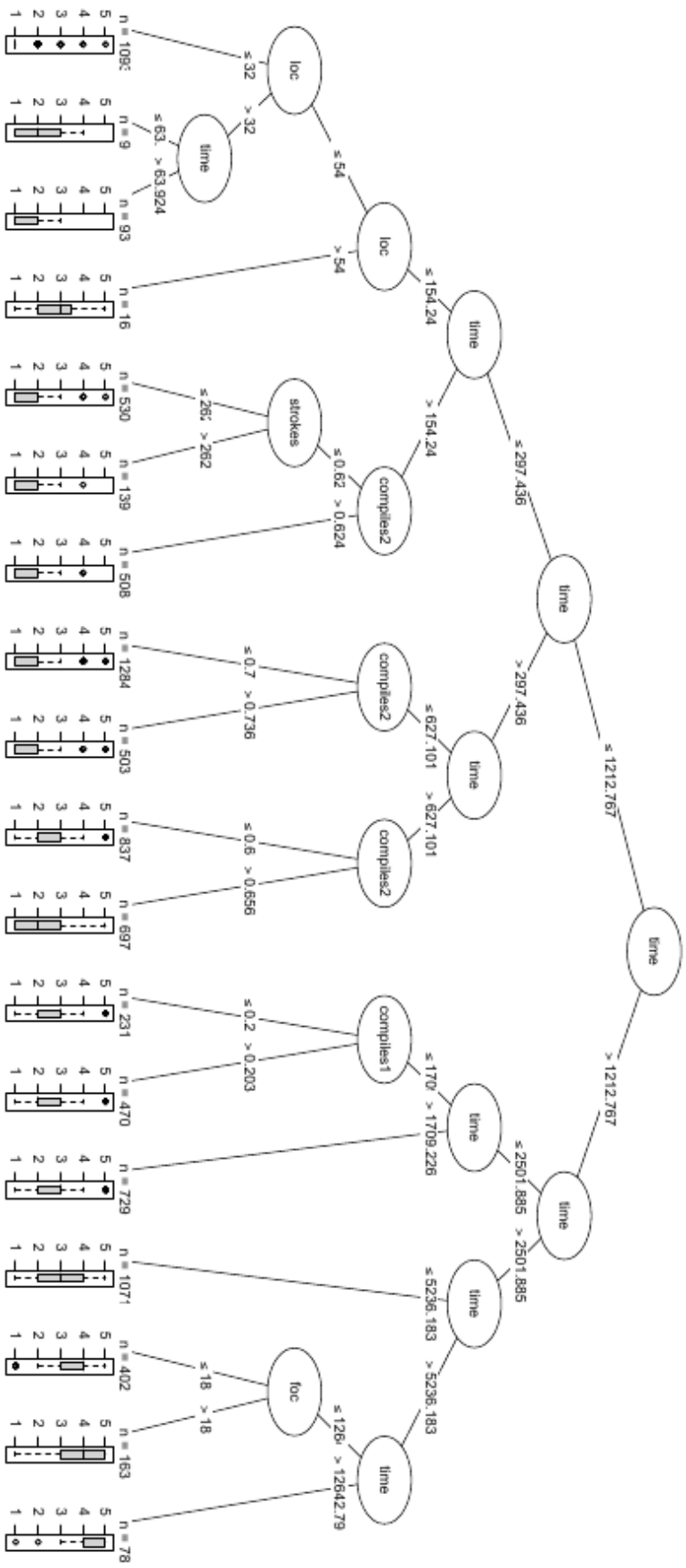

Figure 5.2: A decision tree from Ihantola, Sorva and Vihavainen [62] figure 2.

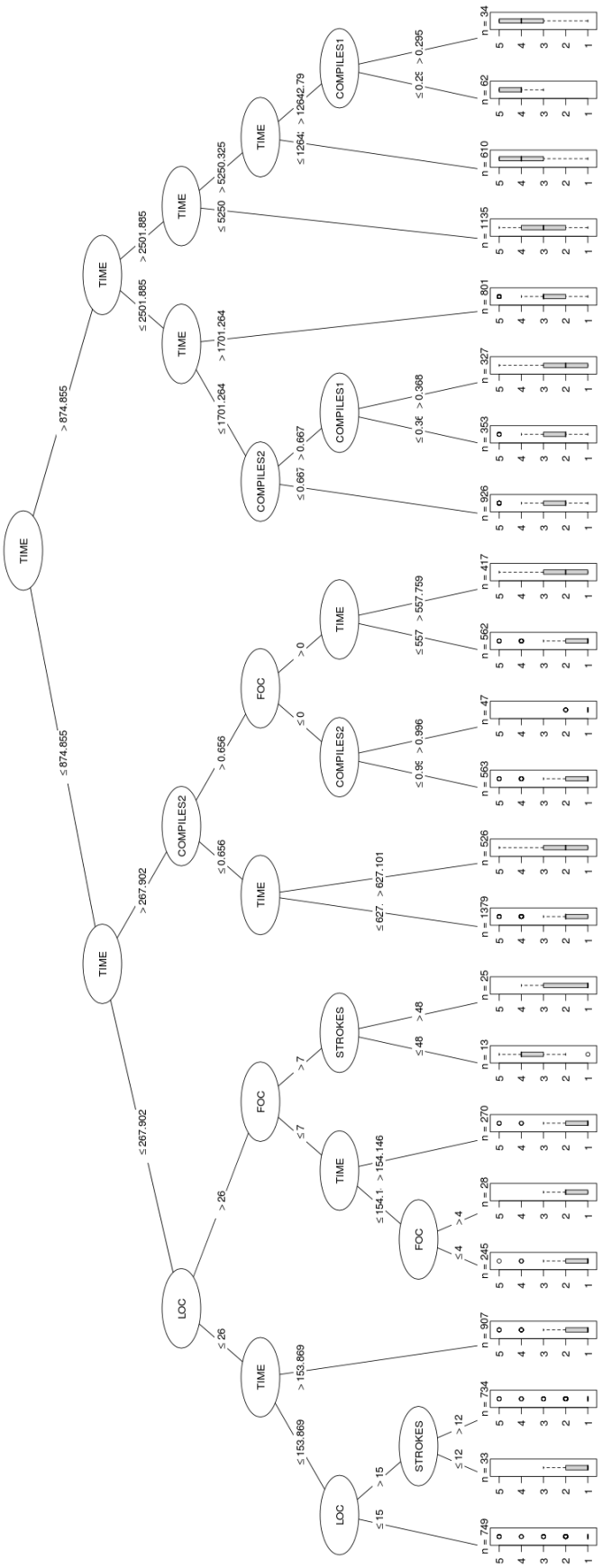

Figure 5.3: Decision tree obtained in re-analysis when utilizing all assignments data.

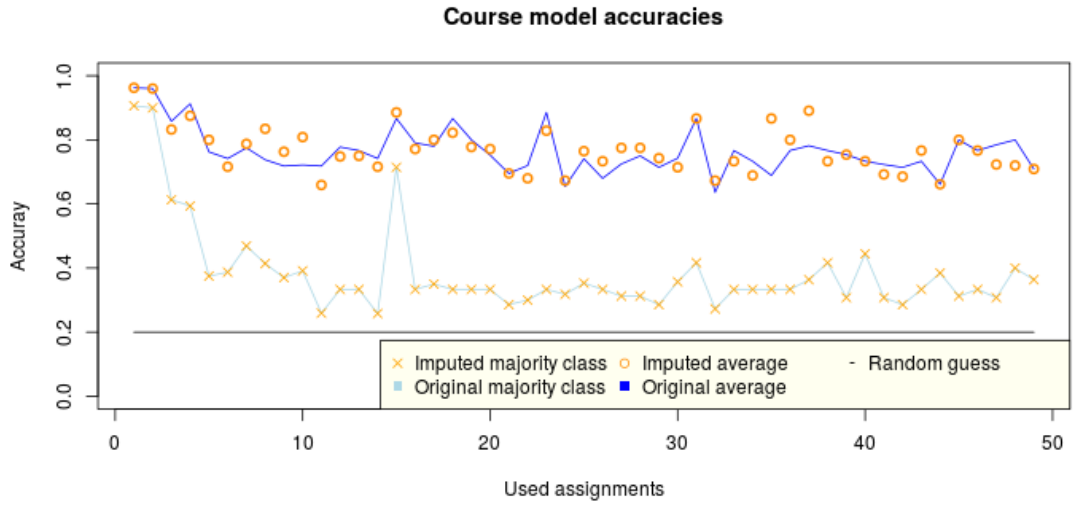

Figure 5.4: Course model average, majority classifier and random guess accuracy. Refer to text for more information on *used assignments*[ua].

5.2 Course model

In order to answer RQ2 and RQ3 we need a model, which is based on an idea from the paper by Ihantola, Sorva and Vihavainen and can be thought of as a course model. The dataset for each assignment model is created by selecting that assignment related features that are used in ctree, see Table 4.1, including difficulty value and adding all other assignments' features to this except for their difficulty value. Only those assignments that contain each difficulty value at least three times are included. Currently handled assignment data is used to make a split for training and validation datasets with 80% going to former and remaining 20% for latter. The model is trained using 10-fold cross-validation. Upon obtaining the model it is used to make prediction on validation dataset in order to obtain the model performance metrics. This same model training is done using both original and imputed data, so there are two different models obtained, two sets of predictions and two sets of results.

The course model accuracy is shown in Figure 5.4. It is good to keep in mind that while figure appears to show continuous line it is due to fact that those assignments that were unused are omitted from the graph in order to make it more readable, so x-axis is not assignment number but a count of assignments that were used[ua]. Random guess classifier accuracy is shown as a one baseline since number of difficulty values is unbalanced in a lot of exercises. The other baseline is majority class classifier, which few peaks indicate rather skewed distribution of difficulty values in the datasets. Imputation does not have effect here since majority class is taken from the validation dataset instead of training dataset, which is where imputed values are. The thicker line shows actual model average accuracy, which is mostly noticeably better than majority class classifier accuracy. On few occasions' majority classifier gets rather close to the model since amount of difficulty values in the dataset is dominated by single value. In other cases, there is somewhat more distribution in difficulty values. Imputed data appears to have only slight effect to the model performance and in most cases that is

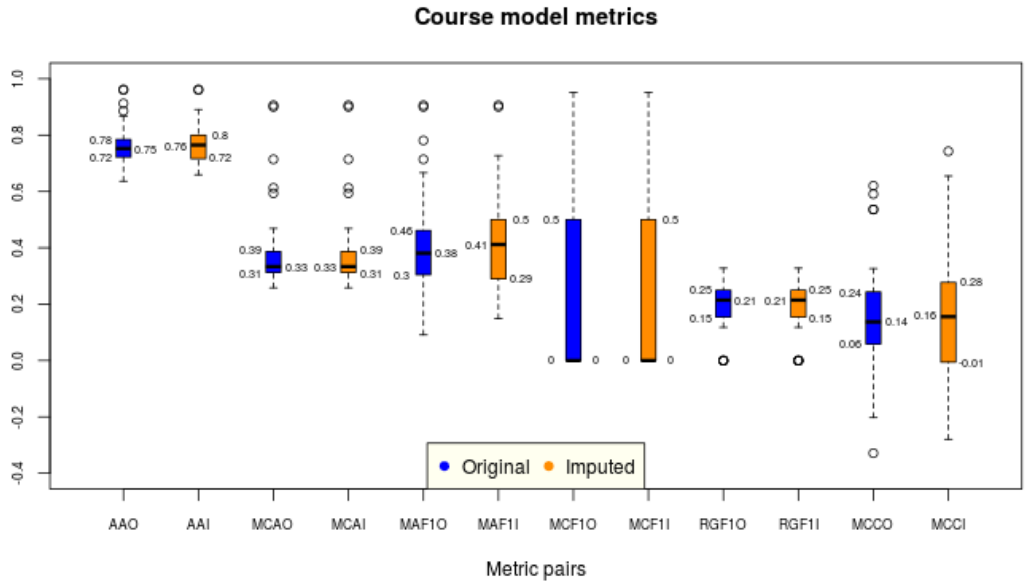

Figure 5.5: Summary of course model statistics. In all acronyms last letter indicates whether it is for original data (O) or for imputed data (I). Acronyms for metrics are AA = average accuracy, MCA = majority class classifier accuracy, MAF1 = micro-averaged F1 score, MCF1 = majority class classifier F1 score, RGF1 = random guess classifier F1 score and MCC = Matthews correlation coefficient.

small nudge towards better average accuracy.

To get a better idea of these models we will look at the summary metric of them. The course models summary, Figure 5.5, reveals that average accuracy of the models is noticeably better than that of major class classifier models, thus they outperform their baseline. The micro-averaged F1 score indicates that both precision and recall are on poor side on the models. On the other hand, when average accuracy is on high side at the same time, it is an indication of imbalance in class distribution, which is known to be the case here. From majority class classifier and random guess classifier F1 scores we can get confirmation that these models perform better than the baselines. Finally, the models MCCs indicates that these models leave a room for improvement. While MCCs are indicating some positive correlation, it is quite meager result. From all the metrics we can observe that the models that utilize imputed data perform marginally better than those without this data. This would also suggest that amount of data is rather low and having more of it might improve results.

5.3 Exercise model

In order to answer to the RQ2 we construct a model that we call an exercise model, which concentrates on a single assignment on its own, and compare it to the original model. The dataset for each assignment model is created by selecting a single assignment related features including difficulty value and leaving all other assignments data out. Remaining part of the process is same than in the course model case. Figure 5.6 shows same two baseline classifiers and the model average accuracy. Similar comparison to the baselines of the exercise model average accuracy as done in the course model

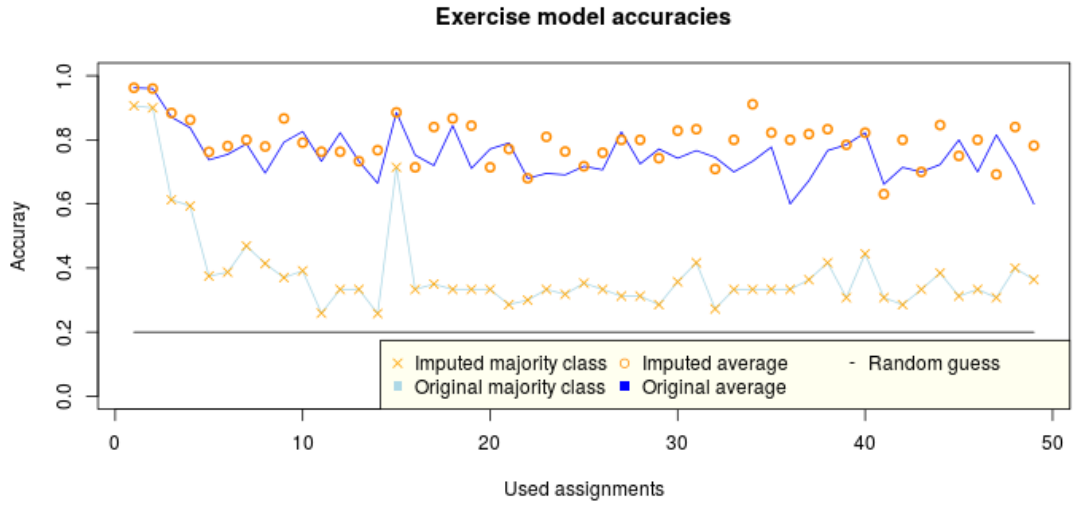

Figure 5.6: Exercise model average, majority classifier and random guess accuracy. Refer to text for more information on *used assignments*[ua].

case reveals it to better than either one of those baselines. The model using imputed data appears to gain more benefit from additional data compared to the course model. This might be due to the exercise model having less data available for training than the course model and since imputation brings more training data in that improves the exercise model more. The course model appears to be a bit better than the exercise model on occasions while overall the exercise model appears to be the better one. This might be explained by course model finding better split points from the larger dataset than the exercise model although latter model should have more focused data for given assignment, even though there is less of it for training the model. In addition, exercises that are closer to one another bear more similarity to each other than those that are further apart, thus, in theory, having more exercises to find split points should help the course model to some extent.

The exercise models summary depicted in Figure 5.7 tells almost identical story to that of the course models. The models outperform their baselines but appear to be only slightly better than the course models. In all metrics using only original data these model's performance are slightly lower than the course models but when adding imputed data to the models they outperform the course models. The exercise models with imputed data MCCs indicates a slightly improved results over the models using original data and they also perform better overall than the course models.

5.4 Exercise model with history

To answer to the RQ3 an exercise model with history is constructed and that is compared to the original model. The dataset for each assignment model is created by selecting an assignment related features including difficulty value and adding all other assignments', which are before current assignment, features for it except for their difficulty value. Rest of the process is same than used with other models. The exercise model with history accuracy is shown in Figure 5.8. Its performance appears to be

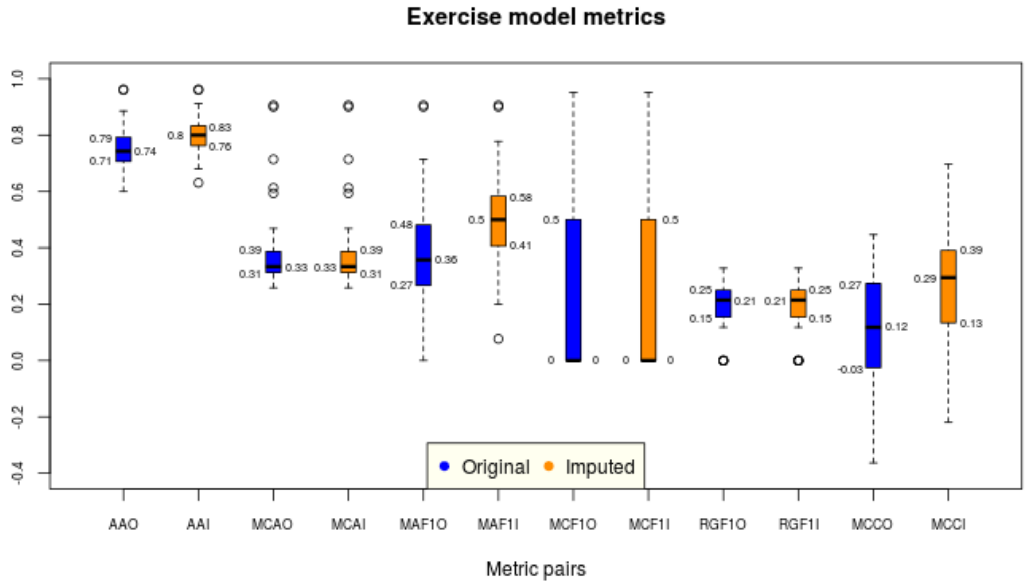

Figure 5.7: Summary of exercise model statistics. In all acronyms last letter indicates whether it is for original data (O) or for imputed data (I). Acronyms for metrics are AA = average accuracy, MCA = majority class classifier accuracy, MAF1 = micro-averaged F1 score, MCF1 = majority class classifier F1 score, RGF1 = random guess classifier F1 score and MCC = Matthews correlation coefficient.

like that of the course model although a bit better overall. This might be due to the exercise model with history using only past and current assignments features rather than looking into to the future for split points. When compared to the exercise model this model performance appears to be somewhat better overall than that model save for some individual cases, thus additional data from beginning to given exercise appears to help the exercise model with history to find those split points a bit better. Imputation seems to affect favorably to this model like the exercise model although to certain extent less than in that model case.

The exercise models with history summary, as seen in Figure 5.9, is quite like the exercise or the course models. Again, the models outperform their baselines and are to some degree better than the course models. Differences are a tad more noticeable when the models are using imputed data compared to that of using original data. The exercise models with history outperforms the exercise models when using original data but appears to be faintly less performing when using imputed data.

5.5 Models metrics compared

To help comparing created models and estimate their performance as relating to the RQ2 and the RQ3, we combine all the model's average accuracies and MCC's together using original, Figure 5.10, and imputed data, Figure 5.11, respectively, in a single figure. When looking at average accuracies of the models there does not appear to be any clear difference between them compared to what earlier summaries already pointed out. MCCs show that there are few models using original data that appear to have some promise in them and they are mainly from the exercise with history models

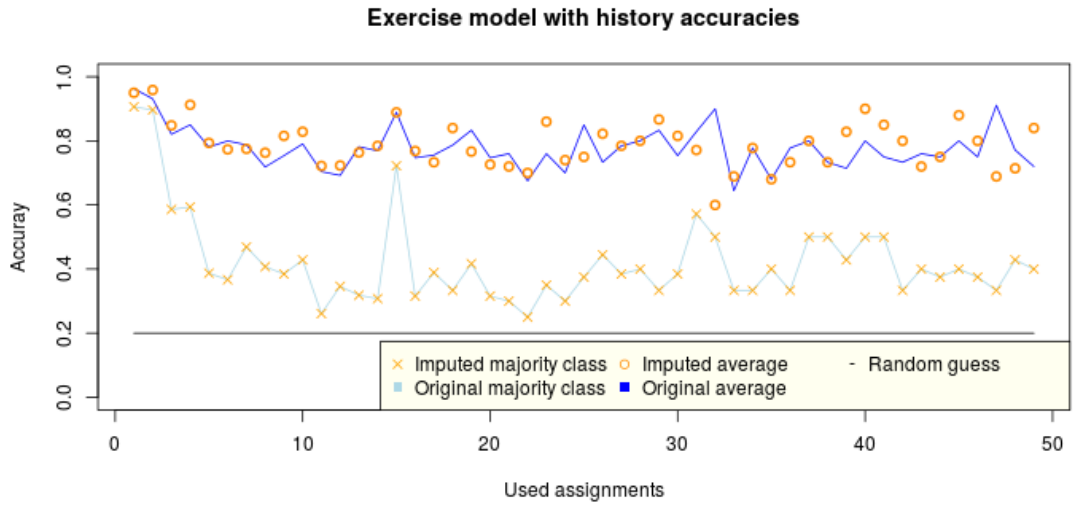

Figure 5.8: Exercise model with history average, majority classifier and random guess accuracy. Refer to text for more information on *used assignments*[ua].

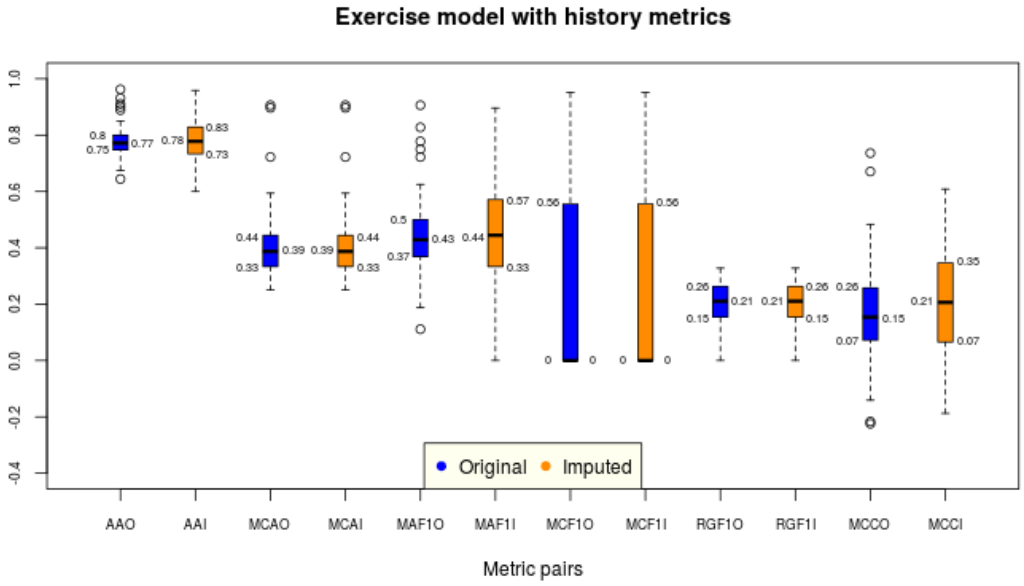

Figure 5.9: Summary of exercise model with history statistics. In all acronyms last letter indicates whether it is for original data (O) or for imputed data (I). Acronyms for metrics are AA = average accuracy, MCA = majority class classifier accuracy, MAF1 = micro-averaged F1 score, MCF1 = majority class classifier F1 score, RGF1 = random guess classifier F1 score and MCC = Matthews correlation coefficient.

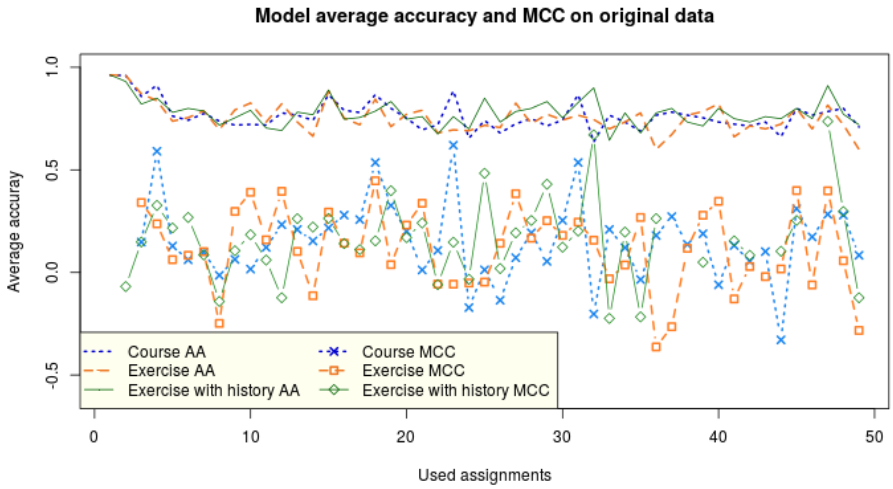

Figure 5.10: Course, exercise and exercise with history models average accuracies and MCCs using original data.

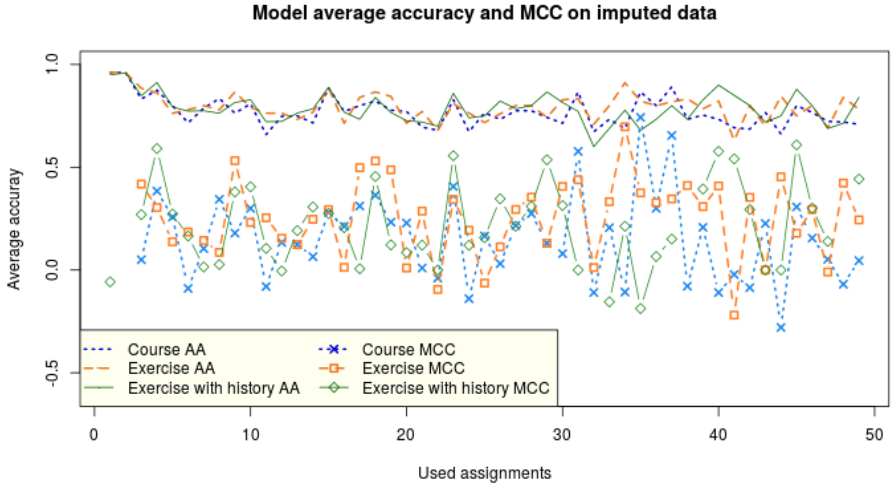

Figure 5.11: Course, exercise and exercise with history models average accuracies and MCCs using imputed data.

except for couple from the exercise models. When using imputed data MCCs appear to gain boost over those using original data just as summaries already pointed out. We can also observe more models showing promise over those using only original data. In addition, very few models appear to lose a bit from their MCC value when using imputed data.

6. Discussion

In this chapter we will discuss obtained results, why they are interesting and what do they offer or could offer in a broader picture. We will also cover issues observed during this process and limitations of results that concern what has been obtained here.

6.1 Combined factors re-analysis

In the original paper by Ihantola, Sorva and Vihavainen combined factors indicate that the more time given assignment completion requires the more difficult it is perceived. They note that also other factors like LOC, FOC and a program being in state that it compiles affects to that perception. In re-analysis performed we can confirm noted observation that time is being the most dominant factor and other factors like LOC, keystrokes and a program being in state that it compiles affects additionally in student's perception of an assignment difficulty. Interestingly time is dominating factor here as it makes a reflection to early days of cognitive load theory measurements where indirect measurements were used, and time was one such measurement. Ihantola, Sorva and Vihavainen note that if student spends more time on average in a given exercise it might be an indication of him struggling in his learning progress, thus assignment may be perceived to be more difficult. Similarly, Chandler and Sweller hypothesized time to be an indication of increase in cognitive load, thus affecting in perceived difficulty of a task as described earlier in section 2.1.2.

Purpose of this re-analysis is to verify if there were errors made during the original data analysis or whether some technique has effect on the outcome of the study done as explained by Gómez, Juristo and Vegas [50]. Re-analysis is a one of ways for validating earlier obtained experimental results while the other two, according to same authors, are replication, which attempts to verify that whatever findings of a study were they are so stable that they can be refound, and reproduction, which attempts to ensure that whatever were found on a study was not due to experimental method used in that study. Juristo and Gómez [68] comment that without reproduction of results it is not possible to identify whether obtained results were due to chance, results are artificial in that they occur only in experiment but not in reality or results are such that they are present in what is examined.

On sciences other than exact sciences, which include mathematics, physics or chemistry, validation is an important part of ensuring that obtained results hold and, as argued by Basili, Shull and Lanubile [7], systematic replication of experiments is a required piece for building knowledge. Despite of being important it does not appear to be valued as much as producing something novel. In a study by Ihantola et al. [63]

relating to computer science education research only 7% of all studies in their literature review were such that someone attempted to replicate or reproduce work done by others previously. They note that exact reasons are unknown although suspect some like field being new or issues relating to privacy or ethics. In another study by Ahadi et al. [2] about replication in computing education they noted that based on query on the Association for Computing Machinery (ACM) digital library only 91 articles out of 21 706 concerning term 'education' contained also word 'replication' in them. It is worth to note that even in that same study they asked computer education researches about whether there is difference between terms replicate, reproduce and repeat and over half, 62%, of respondents indicate that there is no difference in between them, so there is possibility that other terms have been used in the articles as well.

Without validation of results we might end up having quite interesting studies published that have issues with them. For example, some time ago two researchers claimed to have found a way to find out accurately whether a person has a chance at learning to program or not, Dehnadi and Bornat [36] and Dehnadi [35]*. Obviously, this stirred quite a bit of interest as such a test would be very valuable and wanted. According to Lung et al. [77] about six groups decided to replicate these results, them being one of those groups. They were unsuccessful at obtaining similar results but found out various issues on their way that had or might have some effect on the results and that are more general such as population and their background, instruments performing the test and ethics that put some limitations to their experiment but ensure that what researches do are ethical. Similarly, another group concluded that "Our results is an unequivocal rejection of the hypothesis", which was "that there is a positive correlation between a student's mental model and the student's ability to learn programming." (Caspersen, Bennedsen and Larsen [22]). They also noted rather long list of issues that might explain different results including things like course material, course work, resources and the degree of alignment, which means agreement with syllabus, course exam and content.

6.2 Assignment models

We created three models in a manner described in section 3.1 to predict perceived difficulty of a programming assignment. Given perceived difficulty is provided by students using subjective measure, which is described in section 2.1.2. The course model utilizes all available data, the exercise model utilizes only data relevant to a given assignment and the exercise with history model utilizes all data leading to a given assignment including that assignment data. We then evaluated the models using various metrics like mentioned in section 3.3. We observed that all the models outperform their baseline metrics indicating that there is some promise in them. All the models summarized MCC scores indicate barely positive correlation leaving a space for improvements. The exercise model with imputed data shows improvement over the course model, the exercise model with history shows improvement over the course model and the exercise model with history using original data shows improvement over the exercise model.

*The first paper has not been published as is while the second one has been. Bornat later retracted much of claims done in that first paper in his paper publicly available via his home page [12].

The most dominating issue here is the data. Typically, a machine learning algorithm requires good amount of data to work and preferably such data is of high quality. While there are machine learning algorithms that work with small amounts of data, e.g. comparison done by Forman and Cohen [42], it typically means that these have only few parameters, which means of low complexity and high bias, there is a strong prior when looking at things from a Bayesian point of view or both. If the whole available data here could have been distributed over all assignments evenly it had meant roughly one hundred samples per assignment. Additionally, data distribution is typical in that majority of the data comes from the early part of the course and that leaves the latter part of the course with considerably less data. The other issue, quality, is notable as there are quite a few missing values and difficulty ratings are skewed, so that in some assignments only some ratings are given and even if all ratings are given the amount of them is as low as one. If there had been suitable amount of quality data it might change the models and obtained outcomes, hopefully for better but more likely it would improve the models.

It is interesting to see if the models can find some sort of patterns, so to get an idea of this we took a look at the model's performance divided into the model groups. We looked at the top performing models based on their MCC scores and in what assignments they were achieved. Then we looked at basic statistics of these assignment features. Majority of the best course models come from procedural programming paradigm in both original and imputed data case but beyond that there do not appear to be any indications of similarities or dissimilarities between the assignments in the models on this category. The exercise model category appears quite like that of the course model as majority of the best models come from procedural paradigm assignments. The only indication comes from the models using original data that show a faint sign of performing better with assignments that consist of less lines of code but other than that nothing else is noticeable. The exercise with history models have nearly all the best models from procedural programming paradigm as well. Those using original data show a faint sign of performing better with assignments that have fewer keystrokes than models in general and are done in shorter time. Those using imputed data show similarly faint sign of better performance with shorter time used in an assignment. Beyond that there does not appear to be any other signs observed.

From the opposite end are cases where the model performs worst. We did same process than in the best model's case except now we check the bottom performing models. This time majority of the worst course models come from object-oriented paradigm assignments in both cases. In cases the models use original data there is a faint indication with models' performance when used time for assignment is high. In cases the models use imputed data both high number of keystrokes and used time shows similar issues with the model performance. The exercise models using original data worst models appear to be nearly evenly divided between the two paradigms but those models using imputed data seems to be dominated by object-oriented programming paradigm assignments. In both model categories longer time seem to be a faint indicator of the model having issue in its performance. The exercise models with history follow similar division between two paradigms as the exercise models. In this case those models using original data appear to have a faint indication of performing worse if student has used a lot of keystrokes and used more time when doing the assignment.

Nothing similar is noted on those models using imputed data.

Overall signs of any sort of patterns are at the best faint and no real conclusions are feasible to draw. For example, the division between model's performance on two paradigms could be explained with amount of data favoring the procedural paradigm, which is taught on the first half of the course, compared to that of object-oriented paradigm. Having a strong indication of some sort of pattern would be very interesting to look at further if such could be used to create better models but at least in this case no such sign is noticeable. We will shift our view to why this kind of work, as done in this thesis, is interesting and what it could offer.

Programming is mostly learned by writing programs like in XA method as mentioned in section 4.1. In an ideal world we would have a small number of students present in a suitable classroom where an instructor would guide them and provide feedback for everyone based on their needs. In contrast to real world courses, whether arranged in traditional format or something like a MOOC, such guidance and feedback is quite difficulty or nearly impossible to do. Due to this additional help and feedback is often tried to add to the course material and exercises but also into tools that are used for working. Interest into these kinds of models such as developed in this thesis come among other things from research done on tools that support teaching programming like e.g. Ihantola et al. [61] review on automated programming assignments assessment systems or Keuning, Jeuring and Heeren [70] review on tools providing formative feedback.

When thinking about a student doing a programming assignment and running into a problem, he may try to search help for it from the course material, discussion channel if available for the course, from internet or from the system used in the course. The last one is typically such that student submits his work to it and receives some feedback such as certain test case pass or fail. Normally such system is unaware of what kind of student is, i.e. whether he is just learning the ropes or someone who has a decent knowledge already. Similarly, that system does not know whether this assignment is difficult or not for this student, thus feedback is based on what submission consist of. If the system would have background information of this same student and additionally whether student perceives this assignment easy or difficult, the system could potentially provide much more useful information to given student and assignment that he is working with. Even without background information if the model could only predict difficulty for an individual in binary way, easy or difficult, it would be good improvement and addition to the teaching systems.

In case there were a model that could predict difficulty either in easy-difficult axis, or in a more fine-grained fashion as done here, it could provide feedback automatically to a student in some non-intrusive way. This would require more from used system, but it would help and ease up student's learning experience. In addition, such a model could be used even better to provide customization of exercises for a given student. If the course provides three exercises of loop construct to a student and he does them all properly without bigger issues, then the system could move him forward by teaching something new. On the other hand, if the student is able to solve only one of such exercise in some way then the system could provide additional help in form of extra material, extra exercises or both. Once a student masters given concept, he would move forward to something new. Such a system would provide guidance to a student at his

own pace, which is much more useful than forcing everyone into group of an average student mold, which exists only in a paper. Apparently, such course material and aid provided by the system should be carefully designed, so that e.g. their extraneous cognitive load, cf. definition of a square in section 2.1.1, is kept very low since a student is trying to learn something else than how to read given material.

6.3 Limitations

As mentioned in results section 5.1 the most notable issue with this re-analysis is lack of documentation of what and how things were done in the original study. The paper describes in general level of steps carried out, but it omits details. The package given by authors provides somehow processed data along with some scripts but what processing is done and what those scripts do are unknown and undocumented. When thinking about doing re-analysis where only experimenters are changed but data and analysis should be same as in original study without full knowledge of how things are done re-analysis will be difficulty and obtained results, if any, might or might not confirm original study results. It is unclear as what caused results and if proper re-analysis was done in the first place. Lack of complete information might also affect one's interest in attempt to perform replication of some work.

When thinking about other ways of validating the work one needs to be aware of context. For example, data in this case is obtained from the university that has certain way of teaching programming, student population has quite similar backgrounds, tooling system to obtain data is specific for the university although it is openly available to others and follow certain ethics*. While other factors might be easier to understand as how they can affect obtained results, the last one, ethics, might be a bit difficult to spot. An example of varying ethics can be observed from a study by Lung et al. [77]. They mention that one team institutional ethics review board prohibited them to provide monetary incentive for attracting participants while Lung et al. were able to offer a chance to win CAD$50 gift certificate for participants. It might affect participants interests in taking part of some study and is considered good habit to declare such thing clearly on a study done. However, it raises questions such as what kind of people attended to study without such incentive compared to those who had an incentive and are they comparable groups.

Used metrics needs to be also considered. For example, time is noted to be a potential indicator of perceived difficulty but there are issues on its use in current form. If time is running when student has an integrated development environmnet (IDE) open but does something else, then it is possible to come up with a question whether this time should be counted or not. In the paper authors note that if student did not make any keypresses within five minutes period then any time excess of that is cut away from total time. In case student is doing something completely different, such as having a lunch, it does make sense to cut this extra time out but if he is e.g. getting aid from teaching assistant then it might make sense to include that time for total time that the student is using for given assignment. Another issue that relates to this one is uncertainty of obtained data. For example, when a student is requested to

*The Finnish National Board on Reasearch Integrity (TENK) sets bases for them.

create a program in assignment #41* where it selects a random number from certain range and ask user to guess it, one would tend to think that creating such a program takes few minutes. Based on data from that assignment we can observe that median time to do such a program is slightly over 16 minutes (979 seconds), lower quartile to be just under 10 minutes (573 seconds) and the minimum time being 3 seconds. This quickest submission consisted of 37 lines of code and it had 6 flow-of-control structures. This could be an indication of an issue with data collection system or that student has copied his solution from somewhere and pasted into the IDE for submission.

When we think about created models, we observe that the data is an issue but how to solve that is a challenge of its own. Having more students in a course who would allow data collection would be rather clear solution. It is unlikely that this kind of course is going to have large crowds, so other ways to increase amount of data needs to be considered. This could include collecting data over multiple courses or over multiple institutions. While both are valid ways, they would require planning since things like changing course contents between terms or institutions having differences in how teaching is done might affect the resulting data.

As used data for our models came from another study, it was defined by that study requirements and needs. This include also collected and used metrics. Although all metrics do appear to have a good and plausible reason to be in used set it leaves open questions like were they the best ones in prediction or would other metrics help to improve models created here. As noted by original authors their results differ from those observed by Alvarez and Scott, for which they compared their results, but as both studies shared only two metrics their comparison is not direct but more as indicative as what might or might not work on prediction.

Merely having more data would not solve all issues. Used data difficulty rating is based on student's voluntary submission, so they can decide when and what to provide. It could be felt by some as obligatory albeit unwanted thing, thus selecting default or single value for all assignments is possible or, like observed here, just omitting it altogether. If students just omit their response, we run into the non-response bias in which we cannot know whether those responding to question form a different group from those who do not respond at all.

Another issue with difficulty is that it is subjective measurement. For example, a difficulty value of three for one student might be two or four to another student. It also tends to be dynamic rather than being constant. When someone begins to learn programming and is encountered with the first tasks beyond something that one can simply copy or type few lines of code, it might cause such a task to be much more difficult than the earlier ones. Once that person progresses in his studies those earlier deemed difficult tasks become easier and difficulty ratings gain new meaning, thus comparing difficulty value of two in early assignment to same value in later assignments might not mean same. It is also good to remember that evaluation of difficulty is asked only after programming task is done and submitted and that there are no provided instructions as what it means. It is possible that a student is reporting difficulty of a given assignment, but he could be as well reporting assignment being complex. Similarly, it is unknown whether such perception is due to how assignment is given,

*Assignments and material, which are in Finnish, can be found at http://cs.helsinki.fi/group/java/k14-materiaali.

thus more related to extraneous cognitive load, which could be solved by instructional design, or is it more due to intrinsic cognitive load related matters.

In addition, we face issues with various other things like used machine learning algorithms, goal settings and interpretation of obtained results. While used machine learning algorithm in this thesis is based on comparison of algorithms for selected purposes there are others available. What kind of results they might provide is unknown, so comparison of such algorithms might reveal some more suitable one in this context. Here the goal setting was to try to predict difficulty rating from a certain scale, but another type of goal setting could be to try to predict only binary case, i.e. whether given assignment is easy or difficult. That would mean dividing difficulty rating into two and probably utilizing another algorithm.

The last thing, interpretation of results, hides into used metrics. As explained briefly in section 3.3 different metrics carry out different meanings and information. While accuracy may sound a good choice, it may not be the best suited to tell whole picture and similar comment can be done for F1 score as well. MCC might be good choice as a single score indicating the quality of a confusion matrix context as noted by Chicco [28] in binary classifier case but it is only a single value that might not provide information that we are interested in. Due to this it might be good to look at other metrics to get better and more comprehensive view of things before settling down to some metrics and even then it would be good to clearly indicate what, how and why things were done. This relates to larger issue of selecting only suitable metrics for given study in order to obtain good looking results that might be found out when someone else performs replication of such study.

Other useful performance metrics, similarly, in binary case, are the precision-recall curve and AUC as pointed out by Chicco [28]. When dealing with multiclass case AUC has no generalization in curve form but utilizing its probabilistic form it can be generalized like done by Hand and Till [55]. However, when this is done, we are dealing with mean AUC, which is not equivalent to its binary version and instead of plotting it one should utilize its numeric form to make interpretations. Another possible measurement is Cohen's kappa, which measures the agreement between two raters that classify items into mutually exclusive categories as presented by Cohen [30]. While its values should be comparable between two models its interpretation appears to be difficult and it has received a lot of criticism over the years as pointed out by e.g. Pontius and Millones [95].

In many cases it would be interesting to look at what would happen to obtain results in a different context but how to do that in controlled fashion since there are many things that affect to the whole. As noted in section 2.2.2 the introductory course teaching varies a lot in a single country, so how can we be sure that the models here would be useful in those institutions or elsewhere where more context related parameters change. There is research done in machine learning that attempts to find out how to store knowledge learned while solving some problem and then how to apply it to related but different problem. It is called transfer learning and it could be one way to start working this issue.

7. Conclusions

We performed re-analysis of combined factors presented by Ihantola, Sorva and Vihavainen as part of our process to create models that predict programming assignment difficulty. We were able to replicate original findings to some extent. Our findings indicate that main results are comparable although there are some differences observed. Based on idea from that study we created the course model. It uses all exercises and their features to make predictions of a selected assignment difficulty. We used this model as a baseline.

We then created two other models. The exercise model uses a single exercise's features in making its prediction. The exercise model with history uses all exercises' features leading to current exercise and including it in its prediction making. The exercise model outperforms the course model when using imputed data but does not quite match on its performance when using original data. The exercise model with history outperforms the course model regardless of used data. Between the two exercise models the one with history indicates better performance when using original data whereas the exercise model performs better when using imputed data.

Results, which are based on values from Figures 5.5, 5.7 and 5.9, show following outcomes. The exercise model median average accuracy is -1.33% lower than the course model when using original data but 5.26% higher when using imputed data. Similarly, the exercise model median F1 score is -5.26% lower using original data but 21.95% higher when using imputed data compared to those values on the course model. Finally, the exercise model median Matthews correlation coefficient (MCC) score is -14.29% lower using original data but 81.25% higher when using imputed data as compared to the course model values. Same values for the exercise model with history compared to the course model tells us following. Median average accuracy is 2.67% higher using original data and 2.63% higher when using imputed data. F1 score median improvements are 13.16% and 7.32% for original and imputed data, respectively. Finally, MCC score median improvements are 7.14% and 31.25% for original and imputed data, respectively.

The models here provide a starting place for envisioned difficulty prediction in real time. Based on them it would appear to be possible to create a model that could do such a job although further work is required to get there. Hopefully one day something along the lines presented here will become useful addition to a tool palette of the educators. It could be used in providing automated help or helping new students in their struggle of learning some concept well before moving forward, thus avoiding them progressing with only partial grasp of concepts that they will need in their future assignments and life as a programmer.

Although none of the models appear to provide an exact answer to a question

of how student perceives given assignment, easy or difficult, they do indicate ways into proceeding towards answering that question. In order to get more conclusive answer, it would be good to obtain more of quality data and then re-check these models' performance. Additionally, selecting other metrics might be fruitful as checking whether they themselves can provide better results or in combination of what has been used in here would do the job. This needs care in order to avoid just fishing some wanted results.

Other easier ideas for future would be creating models such as done here using different machine learning algorithms and comparing them. Since there are several algorithms available and all of them have a set of assumptions behind them, it is plausible that something else might produce better results than what we observed here. Another thing would be changing goal setting from multiclass prediction to binary prediction, which is easier than predicting multiple choices.

More challenge can be obtained by trying to figure out how to eliminate issues with difficulty itself. As noted, it is rather individualized, thus one possible way would be to create a new model that is tailored for individual instead of being generic to all participants. This would bring up new challenges but might prove to be much more successful on its predictions than more generic ones. It might be possible to adjust such models, so that it would succeed in considering mentioned dynamic nature of difficult as the student progresses in his studies.

Perhaps even more challenging would be to create a model that would have least amount of dependency on context in sense that it would be useful on more than single place. Whether such thing is doable and purposeful is something to consider. However, broadening models' adaptability to fit a wider range of contexts is likely to be wanted and searched for in a similar fashion than having a general-purpose machine learning method that can be adapted to various places.

An interesting thought comes to one's mind at the end. Since we know that there are a lot of concepts (2.2.1) that novice needs to learn to become a full-fledged programmer and we know that we combine learned information into schemata (2.1.1) to store more complex things into our limited working memory, then is the order used in this course the optimal one or would some other order be better?

Bibliography

[1] D. Afergan, E. M. Peck, E. T. Solovey, A. Jenkins, S. W. Hincks, E. T. Brown, R. Chang, and R. J. Jacob. Dynamic Difficulty Using Brain Metrics of Workload. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 3797–3806, New York, NY, USA, 2014. ACM.

[2] A. Ahadi, A. Hellas, P. Ihantola, A. Korhonen, and A. Petersen. Replication in Computing Education Research: Researcher Attitudes and Experiences. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 2–11, New York, NY, USA, 2016. ACM. event-place: Koli, Finland.

[3] A. Alvarez and T. A. Scott. Using Student Surveys in Determining the Difficulty of Programming Assignments. *J. Comput. Sci. Coll.*, 26(2):157–163, Dec. 2010.

[4] J. A. Aslam, R. A. Popa, and R. L. Rivest. On Estimating the Size and Confidence of a Statistical Audit. *Proceedings of the Electronic Voting Technology Workshop (EVT '07)*, 7:12, 2007.

[5] P. Ayres. Impact of Reducing Intrinsic Cognitive Load on Learning in a Mathematical Domain. *Applied Cognitive Psychology*, 20(3):287–298, 2006.

[6] W. Barfield. Expert-novice differences for software: implications for problem-solving and knowledge acquisition. *Behaviour & Information Technology*, 5(1):15–29, Jan. 1986.

[7] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, July 1999.

[8] J. Bennedsen and M. E. Caspersen. Failure Rates in Introductory Programming. *SIGCSE Bull.*, 39(2):32–36, June 2007.

[9] M. Bilalić, P. McLeod, and F. Gobet. Expert and "novice" problem solving strategies in chess: Sixty years of citing de Groot (1946). *Thinking and Reasoning*, 14, Nov. 2008.

[10] G. Borg. Interindividual scaling and perception of muscular force. *Kungliga Fysiografiska Sällskapet i Lund*, 12:117–125, 1961.

[11] G. Borg, O. Bratfisch, and S. Dornič. On the Problems of Perceived Difficulty. *Scandinavian Journal of Psychology*, 12(1):249–260, Dec. 1971.

[12] R. Bornat. Camels and humps: a retraction. http://eis.sla.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf. Accessed: 21.3.2019.

[13] S. Boughorbel, F. Jarray, and M. El-Anbari. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *PLoS ONE*, 12(6), June 2017.

[14] R. Brause, T. Langsdorf, and M. Hepp. Neural data mining for credit card fraud detection. In *Proceedings 11th International Conference on Tools with Artificial Intelligence*, pages 103–106, Nov. 1999.

[15] L. Breiman. Bagging predictors - technical report no. 421. Technical report, Department of Statistics University of California, Berkeley, CA 94720, 1994.

[16] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.

[18] R. Brünken, S. Steinbacher, J. L. Plass, and D. Leutner. Assessment of Cognitive Load in Multimedia Learning Using Dual-Task Methodology. *Experimental Psychology*, 49(2):109–119, 2002.

[19] N. Buduma. Curse of dimensionality. http://nikhilbuduma.com/img/dimension_sparsity.png, 2015. Accessed: 12.9.2018.

[20] T. Bylander and D. Hanzlik. Estimating Generalization Error Using Out-of-Bag Estimates. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 321–327, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[21] R. Caruana and A. Niculescu-Mizil. An Empirical Comparison of Supervised Learning Algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 161–168, New York, NY, USA, 2006. ACM. event-place: Pittsburgh, Pennsylvania, USA.

[22] M. E. Caspersen, K. D. Larsen, and J. Bennedsen. Mental Models and Programming Aptitude. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 206–210, New York, NY, USA, 2007. ACM. event-place: Dundee, Scotland.

[23] P. Chandler and J. Sweller. Cognitive Load Theory and the Format of Instruction. *Cognition and Instruction*, 8(4):293–332, Dec. 1991.

[24] P. Chandler and J. Sweller. The Split-Attention Effect as a Factor in the Design of Instruction. *British Journal of Educational Psychology*, 62(2):233–246, 1992.

[25] W. G. Chase and H. A. Simon. Perception in Chess. *Cognitive Psychology*, 4:55–81, 7 1973.

[26] P. Cheeseman and J. Stutz. Bayesian Classification (AutoClass): Theory and results. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Dicovery and Data Mining*, pages 153–180. AAAI Press/MIT Press, Menlo Park CA, 1996.

[27] M. T. H. Chi, R. Glaser, and E. Rees. Expertise in Problem Solving. Technical Report TR-5, Learning Research and Development Center, University of Pittsburgh, PA, May 1981.

[28] D. Chicco. Ten quick tips for machine learning in computational biology. *BioData Mining*, 10, Dec. 2017.

[29] C. Clifton. Data mining. https://www.britannica.com/technology/data-mining, 2017. Accessed: 22.8.2018.

[30] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[31] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: Making thinking visible. *American Educator*, 15:18, 1991.

[32] A. Collins, J. S. Brown, and S. E. Newman. Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. Technical Report No. 403. Technical report, American Federation of Teachers, Jan. 1987.

[33] M. L. Commons, E. J. Trudeau, S. A. Stein, F. A. Richards, and S. R. Krause. Hierarchical complexity of tasks shows the existence of developmental stages. *Developmental Review*, 18(3):237–278, 1998.

[34] A. D. de Groot. *Thought and Choice in Chess*. Amsterdam University Press, 2 edition, 1978.

[35] S. Dehnadi. Testing Programming Aptitude. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, page 17, Brighton, UK, 2006.

[36] S. Dehnadi and R. Bornat. The camel has two humps (working title). School of Computing, Middlesex University, UK, 2 2006.

[37] R. Duran, J. Sorva, and S. Leite. Towards an Analysis of Program Complexity From a Cognitive Perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 21–30, New York, NY, USA, 2018. ACM.

[38] D. E. Egan and B. J. Schwartz. Chunking in recall of symbolic drawings. *Memory & Cognition*, 7(2):149–158, Mar. 1979.

[39] Entropy. https://www.merriam-webster.com/dictionary/entropy. Accessed: 18.9.2018.

[40] INFOGRAPHIC: Coding at school - How do EU countries compare? https://www.euractiv.com/section/digital/infographic/infographic-coding-at-school-how-do-eu-countries-compare/, Oct. 2015. Accessed: 23.4.2019.

[41] K. Fisler. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 35–42, New York, NY, USA, 2014. ACM.

[42] G. Forman and I. Cohen. Learning from Little: Comparison of Classifiers Given Little Training. In J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Knowledge Discovery in Databases: PKDD 2004*, Lecture Notes in Computer Science, pages 161–172. Springer Berlin Heidelberg, 2004.

[43] R. E. Francisco and A. P. Ambrosio. Mining an Online Judge System to Support Introductory Computer Programming Teaching. In K. Porayska-Pomsta and K. Vebert, editors, *Workshops Proceedings of EDM 2015*, volume 1446, pages 93–98, Madrid, Spain, June 2015. ResearchGate.

[44] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM.

[45] M. Garbarino, M. Lai, D. Bender, R. Picard, and S. Tognetti. Empatica e3 - A wearable wireless multi-sensor device for real-time computerized biofeedback and data acquisition. In *Through Innovations in Mobile and Wireless Technologies (MOBILEHEALTH)*, pages 39–42. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 01 2015.

[46] D. C. Geary. Educating the Evolved Mind: Conceptual Foundations for an Evolutionary Educational Psychology. In J. Carlson and J. R. Levin, editors, *Educating the Evolved Mind : Conceptual Foundations for an Evolutionary Educational Psychology*, Psychological Perspectives on Contemporary Educational Issues, pages 1–99. Information Age Publishing, 2007.

[47] D. C. Geary. An Evolutionarily Informed Education Science. *Educational Psychologist*, 43(4):179–195, 2008.

[48] T. v. Gog and F. Paas. Instructional Efficiency: Revisiting the Original Construct in Educational Research. *Educational Psychologist*, 43(1):16–26, Jan. 2008.

[49] A. Gomes and A. J. Mendes. Problem solving in programming. In *19th Annual Workshop of Psychology of Programming Interest Group, PPIG'07, July 2-6, 2007, Joensuu, Finland*, page 13, Joensuu, Finland, July 2007.

[50] O. S. Gómez and N. Juristo. Replication , Reproduction and Re-analysis : Three ways for verifying experimental. In *Proceedings of the 1st international workshop on replication in empirical software engineering research (RESER 2010)*, 2010.

[51] J. Gorodkin. Comparing two K-category assignments by a K-category correlation coefficient. *Computational Biology and Chemistry*, 28(5):367–374, Dec. 2004.

[52] T. Götschi, I. Sanders, and V. Galpin. Mental Models of Recursion. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 346–350, New York, NY, USA, 2003. ACM.

[53] C. Gusain. After China and Japan, Canada is Planning To Teach Coding in Kindergarten. https://fossbytes.com/canada-is-planning-teach-coding-kindergarten, June 2017. Accessed: 23.4.2019.

[54] D. Hand and P. Christen. A note on using the F-measure for evaluating record linkage algorithms. *Statistics and Computing*, 28(3):539–547, May 2018.

[55] D. J. Hand and R. J. Till. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. *Machine Learning*, 45(2):171–186, Nov. 2001.

[56] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Text in Statistics. Springer-Verlag New York, 2 edition, 2009.

[57] E. Heinonen. Asymptotic dirichlet problem for $\mathcal{A}$-harmonic functions on manifolds with pinched curvature. *Potential Analysis*, 46(1):63–74, Jan. 2017. arXiv: 1510.01525.

[58] B. Hoffman and G. Schraw. Conceptions of Efficiency: Applications in Learning and Problem Solving. *Educational Psychologist*, 45(1):1–14, Jan. 2010.

[59] T. Hothorn, K. Hornik, and A. Zeileis. Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, Sept. 2006.

[60] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[61] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM. event-place: Koli, Finland.

[62] P. Ihantola, J. Sorva, and A. Vihavainen. Automatically Detectable Indicators of Programming Assignment Difficulty. In *Proceedings of the 15th Annual Conference on Information Technology Education*, SIGITE '14, pages 33–38, New York, NY, USA, 2014. ACM.

[63] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ITICSE-WGR '15, pages 41–63, New York, NY, USA, 2015. ACM.

[64] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer Text in Statistics. Springer-Verlag New York, 1 edition, 2015.

[65] Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/. Accessed: 8.8.2018.

[66] T. Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual conference of the LTSN Centre for Information and Computer Science*, pages 53–58, Loughborough, UK, Aug. 2002.

[67] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013. 999133.

[68] N. Juristo and O. S. Gómez. Replication of Software Engineering Experiments. In B. Meyer and M. Nordio, editors, *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 60–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[69] N. Kasto and J. Whalley. Measuring the Difficulty of Code Comprehension Tasks Using Software Metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13, pages 59–65, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[70] H. Keuning, J. Jeuring, and B. Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 41–46, New York, NY, USA, 2016. ACM. event-place: Arequipa, Peru.

[71] D. Klahr, P. Langley, R. Neches, and R. Neches. *Production System Models of Learning and Development*. ACM Distinguished Dissertations. MIT Press, 1987.

[72] M. V. Kosti, K. Georgiadis, D. A. Adamos, N. Laskaris, D. Spinellis, and L. Angelis. Towards an affordable brain computer interface for the assessment of programmers' mental workload. *International Journal of Human-Computer Studies*, 115:52–66, July 2018.

[73] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on*

Innovation and Technology in Computer Science Education, ITiCSE '05, pages 14–18, New York, NY, USA, 2005. ACM.

[74] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998.

[75] R. Lister. Concrete and Other neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 9–18, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc. event-place: Perth, Australia.

[76] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, pages 119–150, New York, NY, USA, 2004. ACM.

[77] J. Lung, J. Aranda, S. M. Easterbrook, and G. V. Wilson. On the Difficulty of Replicating Human Subjects Studies in Software Engineering. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 191–200, New York, NY, USA, 2008. ACM. event-place: Leipzig, Germany.

[78] A. Luxton-Reilly. Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 284–289, New York, NY, USA, 2016. ACM.

[79] A. Luxton-Reilly, B. A. Becker, Y. Cao, R. McDermott, C. Mirolo, A. Mühling, A. Petersen, K. Sanders, Simon, and J. Whalley. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ITiCSE-WGR '17, pages 47–69, New York, NY, USA, 2017. ACM.

[80] K. Magel, R. M. Kluczny, W. A. Harrison, and A. R. Dekock. Applying software complexity metrics to program maintenance. *Computer*, 15(9):65–79, Sept 1982.

[81] S. Marsland. *Machine Learning: An Algorithmic Perspective*. Machine Learning & Pattern Recognition. Chapman & Hall/CRC, 2 edition, 2014.

[82] B. W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, Oct. 1975.

[83] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[84] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in*

Computer Science Education, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

[85] S. R. MD Derus and A. Z. Mohamad Ali. Difficulties in learning programming: Views of students. In *1st International Conference on Current Issues in Education (ICCIE 2012)*, pages 74–79, Sept. 2012.

[86] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

[87] G. A. Miller, E. Galanter, and K. H. Pribram. *Plans and the Structure of Behavior*. New York: Holt, Rinehart and Winston, 1960.

[88] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. The MIT Press Cambridge, MA and London, England, 1 edition, 2012.

[89] E. Owen and J. Sweller. What do students learn while solving mathematics problems? *Journal of Educational Psychology*, 77(3):272–284, 1985.

[90] Oxford English Dictionary. http://www.oed.com/. Accessed: 27.7.2018.

[91] F. G. W. C. Paas. Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach. *Journal of educational psychology*, 84(4):429–434, 1992.

[92] F. G. W. C. Paas and J. J. G. V. Merriënboer. The Efficiency of Instructional Conditions: An Approach to Combine Mental Effort and Performance Measures. *Human Factors*, 35(4):737–743, 1993.

[93] A. Painsky and S. Rosset. Cross-Validated Variable Selection in Tree-Based Methods Improves Predictive Performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(11):2142–2153, Nov. 2017.

[94] M. Piteira and C. Costa. Learning Computer Programming: Study of Difficulties in Learning Programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, ISDOC '13, pages 75–80, New York, NY, USA, 2013. ACM.

[95] R. G. J. Pontius and M. Millones. Death to Kappa: birth of quantity disagreement and allocation disagreement for accuracy assessment. *International Journal of Remote Sensing*, 32(15):4407–4429, 2011.

[96] T. Roos, P. Hoyer, J. Kivinen, et al. Introduction to machine learning [Class handout on the second lecture]. https://www.cs.helsinki.fi/webfm_send/1886/, 2016. Helsinki, Finland: Department of computer science, University of Helsinki, Accessed: 12.9.2018.

[97] A. Schmeck, M. Opfermann, T. v. Gog, F. Paas, and D. Leutner. Measuring cognitive load with subjective rating scales during problem solving: differences between immediate and delayed ratings. *Instructional Science*, 43(1):93–114, Jan. 2015.

[98] T. A. Scott. Size and complexity metrics and introductory programming students. *The Journal of Computing in Small Colleges*, 11(2):165–173, 1995.

[99] O. Seppälä, P. Ihantola, E. Isohanni, J. Sorva, and A. Vihavainen. Do We Know How Difficult the Rainfall Problem is? In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 87–96, New York, NY, USA, 2015. ACM.

[100] R. Shaikh. Cross validation explained: Evaluating estimator performance. https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85, Mar. 2018. Accessed: 2.4.2019.

[101] C. E. Shannon. A Mathematical Theory of Communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, Jan. 2001.

[102] H. A. Simon and K. Gilmartin. A Simulation of Memory for Chess Positions. *Cognitive Psychology*, 5(1):29–46, July 1973.

[103] J. Sweller. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12(2):257–285, 1988.

[104] J. Sweller, P. Ayres, and S. Kalyuga. *Cognitive load theory*. Number 1 in Explorations in the Learning Sciences, Instructional Systems and Performance Technologies. Springer-Verlag New York, 2011.

[105] J. Sweller and G. A. Cooper. The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction*, 2(1):59–89, Mar. 1985.

[106] J. Sweller, J. J. G. v. Merriënboer, and F. G. W. C. Paas. Cognitive Architecture and Instructional Design. *Educational Psychology Review*, 10(3):251–296, Sept. 1998.

[107] Z. Usiskin, E. Willmore, D. Witonsky, and J. Griffin. *The Classification of Quadrilaterals: A Study of Definition*. Research in Mathematics Education. Information Age Publishing, Charlotte, NC, 2008.

[108] A. Vihavainen, M. Luukkainen, and J. Kurhila. Multi-faceted Support for MOOC in Programming. In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 171–176, New York, NY, USA, 2012. ACM.

[109] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme Apprenticeship Method in Teaching Programming for Beginners. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 93–98, New York, NY, USA, 2011. ACM.

[110] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding Students' Learning Using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, New York, NY, USA, 2013. ACM.

[111] N. Vlako. The countries introducing coding into the curriculum. https://jaxenter.com/the-countries-introducing-coding-into-the-curriculum-120815.html, Sept. 2015. Accessed: 23.4.2019.

[112] C. Watson and F. W. Li. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 39–44, New York, NY, USA, 2014. ACM. event-place: Uppsala, Sweden.

[113] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[114] L. E. Winslow. Programming Pedagogy - a Psychological Overview. *SIGCSE Bull.*, 28(3):17–22, Sept. 1996.

[115] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers, 4 edition, 2011.

[116] V. Zwass. Expert system. https://www.britannica.com/technology/expert-system, 2016. Accessed: 22.8.2018.