TURUN
YLIOPISTO
UNIVERSITY
OF TURKU

UNDERSTANDING NOVICE PROGRAMMER BEHAVIOR ON INTRODUCTORY COURSES

Learning analytics approach

Erno Lokkila

TURUN YLIOPISTON JULKAISUJA – ANNALES UNIVERSITATIS TURKUENSIS
SARJA – SER. F OSA – TOM. 20 | TECHNICA – INFORMATICA | TURKU 2023

University of Turku

Faculty of Technology
Department of Computing
Computer Science
Turku Research Institute of Learning Analytics
Doctoral Programme in Technology

Supervised by

Assoc. professor, Mikko-Jussi Laakso
Turku Research Institute of Learning Analytics

Dr. Athanasion Christopoulos,
Turku Research Institute of Learning Analytics

Professor, Tapio Salakoski
Faculty of Science

Reviewed by

D.Sc. (Tech), Ari Korhonen
Aalto University, Finland

Assoc. Professor, Ingo Frommholz
University of Wolverhampton, The UK

Opponent

Assistant Professor, Brett Becker
University College Dublin, Ireland

ISBN 978-951-29-9147-1 (PRINT)
ISBN 978-951-29-9148-8 (PDF)
ISSN 2736-9390 (Painettu/Print)
ISSN 2736-9684 (Sähköinen/Online)
Painosalama Oy, Turku, Finland 2023

UNIVERSITY OF TURKU
Faculty of Technology
Department of Computing
Computer Science
ERNO LOKKILA: Understanding novice programmer behavior on introductory courses – Learning analytics approach
Doctoral Dissertation, 78 pp.
Doctoral Programme in Technology
February 2023

ABSTRACT

It is not easy to learn programming. This is why increasing theoretical and practical knowledge in programming education benefits both the educators as well as the students. To allow the students to gain maximal benefit from their studies, the educator must be able to recognize the students who are struggling with learning programming. Learning analytics provides a possible solution to this problem.

This thesis demonstrates a novel method to model programmer behavior by using Markov Models. Programming fulfills the Markov property, because the success of the next attempt to compile or execute code is not influenced by the previous attempts; only by the current skill level of the programmer. The model is built using a state machine, which consists of states representing the different phases of the programming process. The state machine contains eight different states and 29 different state transition possibilities. A Markov chain corresponding to a specific student can be computed using this state machine and then used with, for example, machine learning algorithms.

The data for this thesis was collected from a total of five different introductory programming courses, which used either the Java or Python programming languages. The dataset contains 1174 unique students, who made 544 835 total submissions to 411 unique assignments. All programming courses were given in Turku, during 2017-2021.

This thesis provides a theoretical basis for modeling students (Markov Models) and offers a practical method to model students using Markov Models. This thesis only applies unsupervised machine learning methods to the data, specifically the K-Means clustering algorithm. However, supervised methods may also be used. The usefulness of the model is demonstrated by clustering students into three statistically similar clusters: students who perform well, average and poorly. The model is also applied to recognize the programming language used, based only on the transitions within the state machine.

KEYWORDS: Learning Analytics, Machine learning, Programming education

TURUN YLIOPISTO
Teknillinen Tiedekunta
Tietotekniikan laitos
Tietojenkäsittelytiede
Erno Lokkila: Understanding novice programmer behavior on introductory courses – Learning analytics approach
Väitöskirja, 78 s.
Doctoral Programme in Technology
Helmikuu 2023

TIIVISTELMÄ

Ohjelmoinnin oppiminen ei ole helppoa. Tästä syystä ohjelmoinnin opetuksen teoreettinen ja käytännön edistäminen hyödyttää paitsi nykyisin ohjelmointia opettavia, myös opiskelijoita. Jotta opiskelijat voivat saavuttaa maksimaalisen hyödyn opiskelustaan, opettajan täytyy voida tunnistaa ne opiskelijat, joille ohjelmoinnin opiskelu tuottaa hankaluuksia. Oppimisanalytiikka tarjoaa tähän mahdollisuuden.

Tämä väitöskirja esittelee tavan mallintaa ohjelmoinnin opiskelijoiden käyttäytymistä käyttämällä Markovin malleja. Ohjelmoijan käyttäytyminen toteuttaa Markovin ominaisuuden, sillä ohjelmoijan koodin ajoyrityksen onnistumiseen vaikuttaa ainoastaan ohjelmoijan senhetkinen taitotaso; aikaisemmilla yrityksillä ei ole vaikutusta tuleviin kertoihin. Malli rakennetaan käyttämällä tilakonetta, jonka jokainen tila vastaa ohjelmointiprosessin vaihetta. Tilakoneessa on yhteensä kahdeksan eri tilaa ja 29 erilaista tilan muutosmahdollisuutta. Tilakoneesta lasketaan opiskelijaa vastaava Markovin ketju, mitä voidaan käyttää esimerkiksi koneoppimisalgoritmien kanssa.

Dataa tähän väitöskirjaan kerättiin yhteensä viidestä ohjelmoinnin peruskurssista, joissa käytettiin joko Java- tai Python-ohjelmointikieltä. Opiskelijoita kursseilla oli yhteensä 1174. Opiskelijat tekivät yhteensä 544 835 ohjelmointitehtävän palautusta 411 ohjelmointitehtävään. Kaikki ohjelmointikurssit pidettiin Turussa, vuosina 2017-2021.

Tämä väitöskirja tarjoaa teoreettisen pohjan ohjelmoinnin opiskelijoiden mallintamiseen (Markovin mallit) ja tarjoaa menetelmän, jolla Markovin malleja käyttämällä voi mallintaa ohjelmoinnin opiskelijoita. Malliin sovelletaan vain ohjaamattomia koneoppimismenetelmiä, erityisesti K-Means clustering -algoritmia. Tässä väitöskirjassa osoitan myös teoreettisen mallin muutamia käytännön sovelluksia luokittelemalla opiskelijoita samoja ominaisuuksia sisältäviin luokkiin. Malli opetetaan erottelemaan opiskelijat kolmeen ryhmään: hyvin, keskiverrosti ja huonosti pärjääviin. Mallia sovelletaan onnistuneesti myös tunnistamaan käytetty ohjelmointikieli käyttämällä vain tilakoneen tilasiirtymiä.

ASIASANAT: Oppimisanalytiikka, Koneoppiminen, Ohjelmoinnin opetus

Acknowledgemets

The journey to complete a PhD would not be possible without a large support group. I would like take this moment to thank all the people who have helped me on the way.

I would like to thank my supervisors Tapio Salakoski, Mikko-Jussi Laakso and Athanasios Christopoulos for offering me guidance and support throughout this journey. Without them, I would never have started, let alone finished this work.

I also thank my pre-examiners, Ari Korhonen and Ingo Frommholz for their insightful and helpful comments, which helped further improve this work. I would also like to thank Brett Becker for being my opponent in the defence.

I would like to thank the Department of Computing for providing a fertile place for research and thought, especially Filip Ginter, Tapani Joelson and Risto Luukkonen. I must also thank Jari Björne, Antti Hakkala, Erkki Kaila and Riikka Numminen; floorball was fun regardless whether we were on the same side or not. Sanna Ranto, Nina Lehtimäki and everyone else at UTUGS helped me whenever I had questions, big or small. A special thank you goes to Asteriski and all the students at the department; you're the reason I do this work (and also why I have my data set).

ViLLE team, or nowadays the Turku Research Institute for Learning Analytics, was where my PhD journey begun and I had the honor of working with some awesome people, such as Peter Larsson, Teemu Rajala, Petra Enges, Tomi Rautaoja, Erkki Suvila, Juho Kuusinen and Vilho Kivihalme.

I would not have gotten this far without my family. The support from my mother and father has been invaluable and you can now stop asking me for the completion date of my PhD. Marko and Henri, thanks for the nights spent gaming and distracting me just enough from my work to recharge. Last but not least, Tuisku, your encouragement kept me going even through the hard times. Thank you.

Feb 16 2023

Erno Lokkila

Table of Contents

Abbreviations

AI	Artificial Intelligence
AOP	Algorithms and Programming course at University of Turku
EQ	Error Quotient
LA	Learning Analytics
LMS	Learning Management System
MCQ	Multiple Choice Question
ML	Machine learning
MM	Markov Model
HMM	Hidden Markov Model
NMF	Non-negative matrix factorization
NPSM	Normalised Programming State Model
OOP	Object-oriented Programming course at University of Turku
RED	Repeated Error Density

List of Original Publications

This dissertation is based on the following original publications, which are referred to in the text by their Roman numerals:

I Lokkila, E., Rajala, T., Veerasamy, A., Enges-Pyykönen, P., Laakso, M. J., & Salakoski, T. (2016). How students' programming process differs from experts – A case study with a robot programming exercise. In *EDULEARN16 Proceedings* (pp. 1555-1562). IATED.

II Lokkila, E., Kaila, E., Lindén, R., Laakso, M. J., & Sutinen, E. (2017). Refactoring a CS0 course for engineering students to use active learning. *Interactive Technology and Smart Education.*

III Lokkila, E., Christopoulos, A., Laakso, M. J. (2022). Automatically detecting previous programming knowledge from novice programmer code compilation history. *Informatics in Education*.

IV Lokkila, E., Christopoulos, A., & Laakso, M. J. (2022, July). A Clustering Method to Detect Disengaged Students from Their Code Submission History. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1* (pp. 228-234).

V Lokkila, E., Christopoulos, A., Laakso, M. J. (in press). A Data Driven Approach to Compare the Syntactic Difficulty of Programming Languages. *Journal of Information Systems Education.*

The original publications have been reproduced with the permission of the copyright holders.

Contributions of the author

The studies included in this thesis all have more than one author. I am the primary author for all included studies and as such primarily responsible for the study design, data collection and analysis. Next a quick overview of the contributions of the authors regarding the included studies.

I I decided the study methodology. I performed the analysis and data collection from ViLLE. The reporting was done collaboratively by all authors.

II I collected the data from ViLLE and analyzed it. The actual course redesign and reporting results were done collaboratively by all authors.

III I did all data collecting, analysis and reporting of results specifically. Athanasios Christopoulos and Mikko-Jussi Laakso assisted with the study design. Other writing was done collaboratively among authors.

IV I did all data collecting. Athanasios Christopoulos assisted me with data analysis and reporting of results specifically. Writing the article was done together with the other authors.

V I did all data collecting and analysis for the study. The paper was written collaboratively by all authors.

1 Introduction

Even as early as the 1980s, scientists have recognized the importance of computer literacy, or the ability to use computers effectively. Computer literacy has even been dubbed a requirement for citizens (Kemeny, 1983). This computer literacy means all the skills to be able to survive in the modern society. Little did they know in the ’80s how deeply integrated all matters technical would become in only 40 years. Today nearly everything requires the use of a computer; paying taxes, going to school (especially so during the COVID-19 pandemic). Even calling a friend requires the use of a hand-held computer.

Computers make everything (or at least most things) easier. Using ready-made computer programs is an important skill. The ability to program a computer to complete trivial tasks for you is even more important. However, programming is not easy, much less learning to program. Even as recently as 20 years ago, programming courses at universities had a remarkably high drop-out rate, and those who finished the courses were sill not able to effectively program (McCraken et al., 2001).

Since the McCraken report, research has actively sought out means of reducing drop-out rates and improving learning outcomes: Early warning systems for programming courses have been created with methods from machine learning (Costa et al., 2017; Pereira et al., 2019; Veersamy et al.,2020). Laakso (2010) proposes automatic assessment and feedback as one part of the solution to improved learning outcomes. Additionally, course redesigns have been done, which described moderate success in reducing student dropouts and increasing student performance (Lokkila et al., 2017; Kaila et al. 2016; Lokkila et al., 2016; Lokkila et al., 2015; Yadin 2011).

Despite several decades of research, no consensus has been reached on the question: what exactly do students struggle with when programming? The question is difficult, perhaps because the novice student tends to struggle with every aspect of programming: problem solving (de Raadt, 2007), algorithmic thinking (Knuth, 1985), syntax (Shneiderman & Mayer, 1979), and so forth. As with all education, some students seem to learn faster than others and this introduces the need to measure student progress.

In an ideal world, educators would communicate with students face-to-face, assess their skill level and design exercises to help the student reach new heights in programming. This is, unfortunately, often impossible with programming course sizes often reaching hundreds of students. This introduces the need to track student progress automatically. This is most often done using Learning Management Systems (LMS).

When adopting a LMS to a course, interesting opportunities from Learning Analytics (LA) arise. LA is, in a very simplified form, taking data, analyzing it and providing understandable results to students, educators and decision makers (Lang et al., 2022). For students, LA can appear in the form of explanatory statistics: your skills in theme A are matching the learning goals, but more work needs to be done on theme B to meet the goals. Educators may receive an overview of what their students struggle with. Decision makers may be provided learning outcomes from different institutions, courses or other educational instances, to allow for informed decision making.

Several models exist to predict the performance of students on a programming course, such as the EQ (Jadud, 2006), the Watwin score (Watson, Li and Goodwin 2013) and the Normalised Programming State Model (Carter, Hundhausen and Adesope (2015). These models are primarily for predicting course performance and do not directly address the question of student differences. These models provide a single real number as outcome, which makes it very difficult to group students with similar traits together. One could arbitrarily assign ranges, but then no guarantees of student similarity exist within these groups.

In this thesis, I present the scientific background and development process for a model which can be used to objectively measure student behavior in programming courses. The model is data driven and produces consistent results. The model and its products are analyzed as well as applied to real world datasets. The results, performance and applicability of the model are discussed. It is my hope this work plays its small part in improving the programming education community and programming education at large.

1.1 Objectives and Research Questions

This thesis aims to have both theoretical and practical significance. The contributions of this thesis are listed below.
Theoretical contribution:

- This thesis presents a model that is effective at determining differences between novice programming behavior.
- This thesis opens new research possibilities by combining mathematical theory (Markov models), unsupervised machine learning, and educational research into programming.

Practical contribution:

- The model can be used to find groups of similarly behaving students. This allows for future work aimed towards adaptive programming exercises and individual programming study paths.
- The model allows educators to provide differentiated instruction for the students. By providing students with exercises that match their skill, learning can be maximized; the strong programmers can be challenged with more difficult exercises while the weaker programmers can drill down on the basics.

The goal of this thesis is to provide a model for automatically identifying novice programming students of different skill levels. The model is created from data collected mostly from first year programming courses given at the University of Turku. This thesis is built around the research questions described in the following sections.

1.1.1 RQ1: How can a model be built for automatically clustering novice programming students?

Based on my Master's thesis (Lokkila 2016), I concluded that it is possible to automatically cluster primary school students in mathematics' courses based on their performance on mathematical exercises. However, my true passion lies in programming, thus this result in mathematics encouraged me to attempt to create a similar model for programming. This research question is posed to give a starting point to those who wish to create a model of their own, by describing how this model was created.

The aim is to describe the starting point of the model. Then discuss the data collection process. What kind of data was needed and how and where to collect this data? The role of LMSs in data collection is discussed as well as the data collection best practices.

This research question is addressed by publications I and II.

1.1.2 RQ2: How effective is the created model?

If the model can be created, how effective will it be in differentiating between students of different skill levels? To answer this question appropriately, several terms need clarification, such as what is meant by effectiveness. Also, the model should be robust in the sense that repeated applications always produce the same result.

The included publications III, IV and V all address this research question by applying the model to real-world data.

1.1.3 RQ3: What can the model be applied to?

A model without practical applications is worthless. How and where can the proposed model be applied? Publications III and IV present results on applying the model to analyze student behavior. Publication V demonstrates the applicability of the model to analyzing programming languages themselves.

1.2 Structure of this thesis

This thesis has the following structure. In the second section the foundations on which this thesis is built are described. These include the related other models from which this model drew inspiration, but also improves upon. Further, the fundamentals of machine learning are reviewed to prepare the reader for the machine learning approaches used in the model developed for this thesis.

The third section describes the model. Section 3.1 describes how and why the model was developed. Also, data collection best practices are described and the LMS that the data was collected on. Section 3.2 Describes the data collected for this thesis and its included studies. These datasets are further analyzed in section 4. Section 3.3 describes the developed model in detail. The state machine from which the transition matrix is built is described and an algorithm for computing the weights is given. This section also describes how to reduce the matrix to a single value which describes the model in a single number.

The fourth section further analyzes the model and describes its applications in the real world. Section 4.1 performs a normality analysis on the data, which is required for a variety of statistical tests. Section 4.2 applies the dimensionality reduction technique t-SNE on the dataset. Section 4.3 performs a Principal Component Analysis (PCA) on the dataset in order to determine whether the dataset could be represented with fewer variables. The PCA also gives insight into what are the most important features in the models presented. Section 4.4 applies the Non-negative Matrix Factorization (NMF) on the dataset and further determines which features are most prevalent in novice students. Sections 4.5, 4.6 and 4.7 present the application of the model in real world classroom situations.

The fifth section discusses the developed model, its strengths and weaknesses, as well as how it improves upon the existing models. The discussion also touches on the applicability of machine learning models. Section 5.2 addresses the research questions posed for this thesis. Section 5.3 discusses the theoretical and practical contributions of this model. Finally, in section 5.4, the research limitations and their effects are discussed.

The final section summarizes this thesis and provides an overview of the contributions of this model as well as possible research paths made possible by applying the developed model.

2 Foundations

In this section, a revision of the foundations for this work is given. First, in section 2.1, an overview of the related work is given, with special focus to the pre-existing models, on which this thesis is based on. Then, in section 2.2, a quick primer on machine learning, its history and methods, is provided focusing on supervised and unsupervised machine learning techniques.

2.1 Related work

One of the problems in programming education is measuring student skill. Programming is not one skill, but a collection of skills. What these skills actually are, is still under debate. Algorithmic thinking (Denner et al., 2012; McCracken et al., 2001), is one such skill and while its contents vary, common included skills are decomposing problems into modules, analogical reasoning, and systematic planning skills.

Multiple systems have been developed over the years to aid the student in learning programming, but also collect data on the students' progress that the teacher can act on (e.g., Keuning et al., 2018; Toll, 2016; Heinonen et al., 2014; Balzuweit & Spacco, 2013; Vihavainen et al., 2013). These systems work on different levels of granularity of data collection. We can consider storing complete submissions to be the highest, most coarse level of data collection, followed by storing compilation events, and storing individual keystrokes as the finest level of data collection (Toll, 2016). Other methods of data collection from programmers are also available. Surveys are extensively used (Tabanao et al., 2011). Modern technology also allows student eye movements to be tracked in relation to the programming task (Murphy et al., 2009). However, unless the collected data is used to effect change in the students, the act of collecting data is futile.

One possible way to incorporate the collected data into teaching activities is by differentiation. Differentiation is the act of providing different students with different materials, based on the current skill level of the student and their needs (Tomlinson, 2001). Differentiated learning as a concept is not anything new: it started to gain recognition as early as the 1910's (Parker, 1924). Nowadays it is a widely accepted pedagogic approach. Differentiated instruction has also been successfully applied in programming education. It may take such forms as tiered exercises (Mok, 2011), or offering fast students more exercises (Gil Fonseca et al., 2018).

In order to apply differentiated instruction, the skill levels of the students must somehow be discovered. This is not an easy task. As mentioned in the beginning of this section, programming is a collection of skills, but exactly which skills should be measured and how is still under question. Introductory programming courses, which are often called CS1, teach the basics of programming with such topics as variables, conditionals, loops, and methods. However, more learning gains could be gotten by incorporating instruction on problem solving (Dalton and Goodrum, 1992).

Differentiated instruction can be done on four levels: content, process, product and learning environment (Tomlinson, 2000). Content refers to the 'what' is being taught. In programming education, teachers can differentiate for content by, for example, providing programming students exercises of different difficulty: advanced concepts, which may not even be on the syllabus for the difficult level and step-by-step instructions on the easiest level (Mok, 2011). Differentiating by process refers to 'how' the material is accessed. Different, successfully applied methods include auditory, visual, and tactile methods (Taylor, 2015). The product refers to the outcome of the student's learning (Tomlinson, 2000). The student may present their learning either as a presentation, a written report, or even a play or video. Lastly, the learning environment refers to the 'where' aspect of learning (Tomlinson, 2000). Some students prefer learning in silence, whereas other students may enjoy group work or listening to music. Others still may learn best in the familiar environment of their home, while others may prefer to study in a busy cafe.

However, before being able to differentiate students, we must be able to tell who needs differentiation. Modelling students and their learning in programming education is not a new concept. Spohrer & Soloway (1989) created MARCEL, a program that simulates students using a model with three major categories: specification understanding, domain-specific learning and progress criteria. Carter et al. (2015) propose the Normalized Programming State Model, which can be used to generate a model of students based on the time they spend in specific states, such as debugging, editing, and executing.

Once a model of a learner has been developed, it can be analyzed. One way is to cluster students and detect outliers, which is to say those who behave distinctively different from the majority. Detecting outliers in a novice programmer course can be done using a tool called CodeInsight (Fonseca et al., 2018). The tool itself is a simple plugin for an IDE and it collects code snapshots from students at execution time. It then detects students who are significantly slower or faster than the average in completing the exercises on the course. This allows the teacher to act, both in supporting the slow students and giving more tasks to the fast students. The main limitation of this tool is that is uses a fairly simple method to detect the outliers, the mean absolute deviation around the mean.

Other methods of clustering students include analyzing the frequency and error types encountered by the student (Tabanao et al., 2011). While they were not able to derive a model which accurately predicts at-risk students, they do provide insight into specific types of errors that correlate with lowered course performance.

Fwa & Marshall (2018) created a Hidden Markov Model (HMM) to determine the engagement state of students. They used very specific and difficult-to-apply methods, such as head pose tracking and gaze tracking. They also collected keypresses and number of erroneous compilations the students made. Their main finding is that, using their HMM, they were able to cluster students into three states: start, engaged, and disengaged. Students who were in the engaged state tended to stay engaged, while the other states often lead to disengagement.

While it might theoretically be possible to create a single metric to encompass all programming knowledge, this has not been created yet. Instead, it is more reasonable to measure one skill at a time. Regardless of the other skills required for programming, one skill is at its core: the ability to write correct program code, i.e. knowledge of syntax. The next sections describe the mathematical basis for the model described in this study, as well as the existing models from which the developed model is based and improves on.

2.1.1 Markov models

Markov models are named after Andrei A. Markov, who published his results on Markov chains first in 1906. Markov Models are stochastic models where future events do not depend on past events (Ching & Ng, 2006). This property of not depending on the past is called the Markov property.

The simplest Markov Models are Markov Chains. Mathematically a Markov chain could be described as all the values taken by a random variable X from a finite set S, when the next value, X_{n+1} is based solely on some function $\mathrm{F}(X_n)$ (Ching & Ng, 2006). Each value received by X is called a state in the Markov chain. The transitions between states are not random, but determined by each state. Together all the possibilities for state transformations create a transition matrix. This transition matrix is often depicted as a graph or an adjacency matrix. An example of a Markov chain process is given in Figure 1, where $S = [Raining, Cloudy, Windy, Sunny]$.

	Raining	Cloudy	Windy	Sunny
Raining	0	1	0	0
Cloudy	.2	0	.6	.2
Windy	.5	0	0	.5
Sunny	0	1	0	0

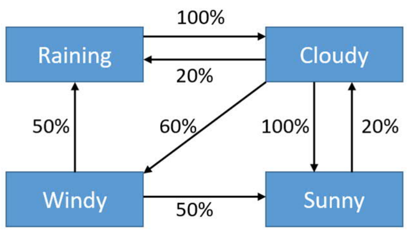


Figure 1 Markov Chain model; The same model depicted as a transition matrix on the left and as a graph on the right.

The transition matrix is notable in the sense, that all its rows sum to one. This is because the state transitions occur on the rows and normally every state has a 100% change of transitioning to another state (including itself). States that do not allow outgoing transitions (i.e., they only transition to themselves) are called absorbing states.

The transition probabilities need not be static. Those Markov chains, where the transition probability is dependent on the current state, are called time inhomogeneous Markov models. Recent work has allowed much larger time-inhomogeneous models on arbitrary state spaces and work is ongoing (Truquet, 2019).

The Markov Chain can be thought of in two ways: one is to think of the model in terms of its transition probabilities, *M* (see Figure 1, left), the other is to think of an agent walking through the model and giving the probability of the agent being in each of the states at the current point in time (Ching & Ng, 2006). The latter is denoted $\boldsymbol{\pi_t}$ as being the *distribution* of the model at time *t*. For instance, a distribution of $\boldsymbol{\pi_t} = [\mathbf{0.25}_r, \mathbf{0.25}_c, \mathbf{0.25}_w, \mathbf{0.25}_s]$ should be understood as the agent having an equal possibility of being in any of the four given states: raining (r), cloudy (c), windy (w), sunny (s).

As the agent then takes their next step according to the probabilities of the transition matrix, the distribution of the model changes. The new distribution is computed by performing a matrix multiplication on the current distribution and the transition matrix (Chiang & Ng, 2006). Mathematically we can define taking a step in the chain with the following formula:

$$\boldsymbol{\pi_t M = \pi_{t+1}}$$

Thus, for our example above, $\boldsymbol{\pi_{t+1}} = [\mathbf{0.175}_r, \mathbf{0.5}_c, \mathbf{0.15}_w, \mathbf{0.175}_s]$.

The *stationary distribution* for the model is the distribution $\boldsymbol{\pi_t} = \boldsymbol{\pi_{t+1}}$, which is to say the distribution does not change as the random walk progresses. The stationary distribution has found many recent applications in e.g., image tampering detection (Wang et al., 2010), COVID-19 prediction (Achmad & Ruchjana, 2021) and optimizing drug doses for multiple-drug treatments (Ma et al., 2021) to name a few.

The Markov Chains where the timestep is discrete, i.e., happens at clear intervals are called *discrete-time Markov Chains*. The alternative is that the transitions happen continuously, in which case the Markov Chain is called continuous-time Markov Chain.

Markov Models can also be used to model *latent states*, or states which are only indirectly observable from the states of the model itself. These states are, in effect, hidden, and thus these models are called Hidden Markov Models (HMM). An HMM has no direct correspondence with the visible state observed from the model and the state the model is actually in. Chiang & Ng (2006) give an example of throwing two 4-sided dice, where one of the dice is loaded, while the other is fair. The observable states of the system are the values from the dice: $\{1,2,3,4\}$. The hidden states of the system are $\{$loaded, fair$\}$. Given any given sequence of observations, it is impossible to directly see which thrown die was loaded and which was fair. However, the probabilities may be estimated efficiently (Chiang & Ng, 2006).

Markov Models have found wide application in modern science, including natural language processing (Almutiri & Nadeem, 2022), modeling cell life cycles (Sazonov et al. 2021), forecasting land usage patterns (Al-Shaar et al., 2021) and malware detection (Sasidharan & Thomas, 2021).

2.1.2 Error Quotient

The Error Quotient (EQ) was first introduced in Jadud (2005) and later revised in Jadud (2006). The purpose of the EQ is to score programming students based on the errors they make. A low EQ means the student made few errors and a high score that the student made many errors. The score is computed on a session-per-session basis, where a session can be of arbitrary length. One session could be a two-hour lab session, or an 8-week course.

The EQ is computed according to the following algorithm (Jadud, 2006):

1. Collate: Create consecutive pairs from all compilation events in a session, e.g., $(e_1, e_2), (e_2, e_3), \ldots, (e_{n-1}. e_n)$.
2. Calculate: Score each pair according to Figure 2.
3. Normalize: Divide the score assigned to each pair by 11 (the maximum value possible for each pair).

4. Average: Sum the scores and divide by n the number of pairs.

The scores in the EQ algorithm are not random. Jadud (2006) spends considerable time searching the parameter space for which parameters have an effect. This search led to the revised EQ algorithm (Figure 2) and the scores involved. The scoring rubric of the original EQ considered a variety of aspects, such as the location of error and proximity of attempted fix to the error. The revised version drops all the variables which have little to no effect and only two remain: Whether both compilation events end in an error and whether both errors are of the same type.

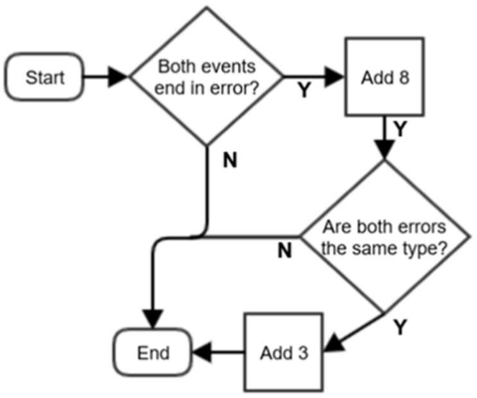


Figure 2. The EQ scoring rubric (Jadud, 2006)

The computation of EQ is illustrated with the following example. Consider a student who creates the following submission history: s, e_1, e_2, e_2, s, where s is an event that does not end in an error. e_1 and e_2 are events which end in different types of errors. The EQ algorithm would be executed by first creating the submission pairs $(s, e_1), (e_1, e_2), (e_2, e_2), (e_2, s)$. These pairs would be scored according to Figure 2 and then normalized. Finally, we average the scores and discover the student has an EQ of about $0.43\overline{18}$.

The EQ model has been found to correlate positively with student achievement when programming in Java (Rodrigo et al. 2009; Tabano et al 2011), but studies with other languages report only mixed success (Petersen et al., 2015).

2.1.3 The Watwin Score

Watson, Li and Goodwin (2013) improved on the EQ by including time spent fixing errors into the scoring rubric. They report better prediction capabilities by as much as 25 percentage units; where the EQ only explained 15%—20% of the variance in data, the Watwin score explained 40%. The better explanation capabilities are attributed to taking into account the time spent by fixing errors and, in case of assignments involving multiple files, the order on which students worked on the files. (Wason, Li and Goodwin, 2013)

The Watwin algorithm is strikingly similar to that of EQ. First some data preprocessing is required in order to discard outliers and other data that would skew the Watwin score. The preprocessing for the Watwin algorithm consists of:

1. *Pair construction*. Create subsequent pairs of code compilation events on the same file (e_1, e_2) ordered by timestamp.
2. *Pair pruning*. Remove all event pairs where the code is identical.
3. *Filtering comment and deletion fixes*. Remove all pairs where compilation success is achieved by either commenting line(s) of code or deleting recently added line(s) of code.
4. *Generalize error messages*. Remove all identifier information from error messages in order to facilitate comparing them. For instance, both the errors “Missing ‘;’” and “Missing ‘)’” would be generalized to “Missing symbol”.
5. *Time estimation*. Estimate time the student spent on changing e_1 to e_2. This may be difficult as students may leave the editor open while taking a break or they may work on other files, to name a few problems.

The actual Watwin algorithm is as follows:

1. **Preprocess** data according to steps above.
2. **Score** each compilation event pair according to Figure 3.
3. **Normalize** scores by dividing by 35 (the maximum score).
4. **Average** normalized scores.

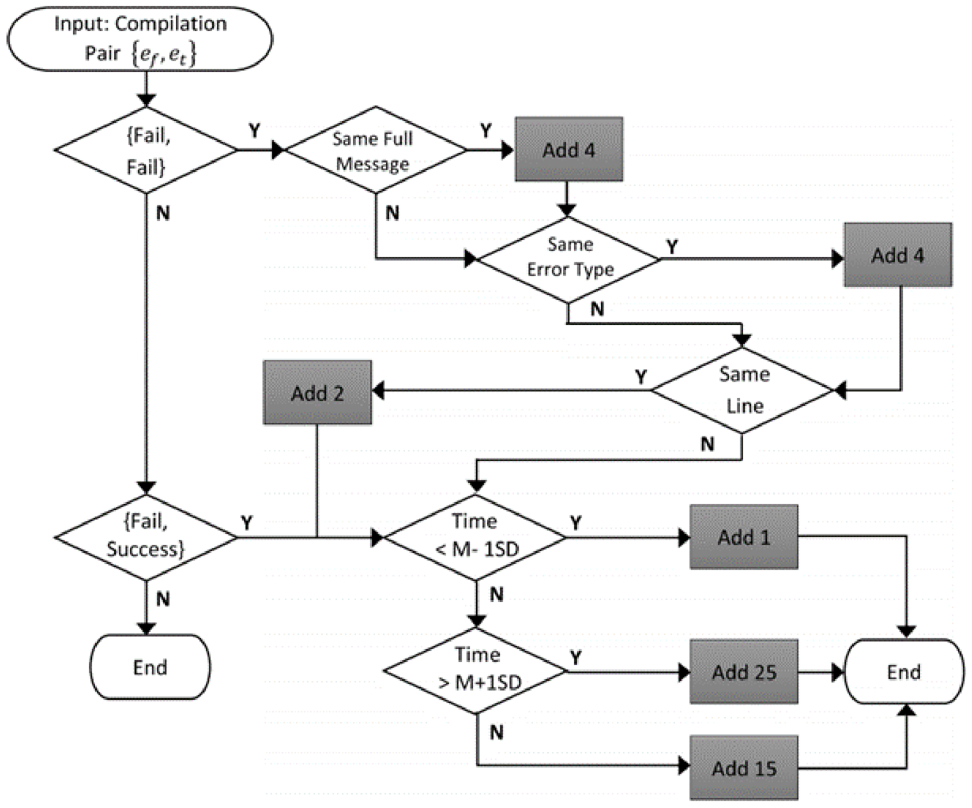


Figure 3. The Watwin scoring rubric

The scores for the Watwin algorithm were not randomly chosen. The scores were determined by a brute force search around the chosen parameters, so as to minimize deviation of the Watwin score between student sessions and spreading the score between students. This came at the cost of reducing the explanatory power of the model (Watson, Li and Godwin, 2013).

The Watwin algorithm has been found to be a better predictor of academic success than previous academic success such as grades in calculus or college physics (Watson, Li and Godwin, 2013). In other contexts, it has performed equally poor as the EQ (Ahadi et al., 2015).

2.1.4 Repeated Error Density

The Repeated Error Density (RED) model was introduced by Becker (2016). Instead of tracking erroneous compilation events, it merely counts how many times an error is repeated throughout a session. The model was created in response to two questions raised by Jadud in his dissertation:

> “If one student fails to correct an ‘illegal start of expression’ error over the course of three compilations, and another over 10, is one student [about] three times worse than the other? What if the other student deals with a non-stop string of errors, and the repetition of this particular error is just one of many?” (Jadud, 2006)

The value for RED is computed according to formula 1, where *r* is defined as a repeated error. The resulting value is always positive or zero and unbounded at the top. This may be an issue, as growing values of RED may skew analysis for very long sessions. Becker (2016) gives a table of precomputed values for RED (Table 1), which demonstrate the properties of RED. The following are the important properties of RED:

- Does not depend on parameter values or predetermined scores.
- It is additive; e.g. the value for sequence 7 is the sum of values for sequences 5 and 3 (See Table 1)
- It is proportional, see e.g., sequences 3 and 4, or sequences 3 and 6.
- It is unbounded.

(Becker, 2016)

$$\mathrm{RED} = \sum_{i=0}^{n} \frac{r_i^2}{r_i+1} \tag{1}$$

Despite its simplicity, Becker (2016) reports good predictive power especially with small datasets. However, RED does not seem to be widely studied.

Table 1 All sequences of RED up to r_4

Number	Sequence	r	RED
1	x	0	0
2	… x … x …	0	0
3	… xx …	1	0.5
4	… xx … xx …	2	1
5	… xxx …	2	$1.\bar{3}$
6	… xx … xx … xx …	3	1.5
7	.. xxx … xx …	3	$1.8\bar{\bar{3}}$

Number	Sequence	r	RED
8	… xxxx …	3	2.25
9	… xx … xx … xx … xx …	4	2
10	… xx … xx … xxx …	4	$2.\overline{3}$
11	… xxx … xxx …	4	$2.\overline{6}$
12	… xx … xxxx …	4	2.75
13	… xxxxx …	4	3.2

2.1.5 Normalised Programming State Model

The Normalised Programming State Model (NPSM) was developed by Carter, Hundhausen and Adesope (2015). It differs from the models described above in that it explicitly uses a finite state machine to collect data on students. The basis for the model is that students' programming activities need to be looked holistically and to this end they developed the finite state machine shown in Figure 4.

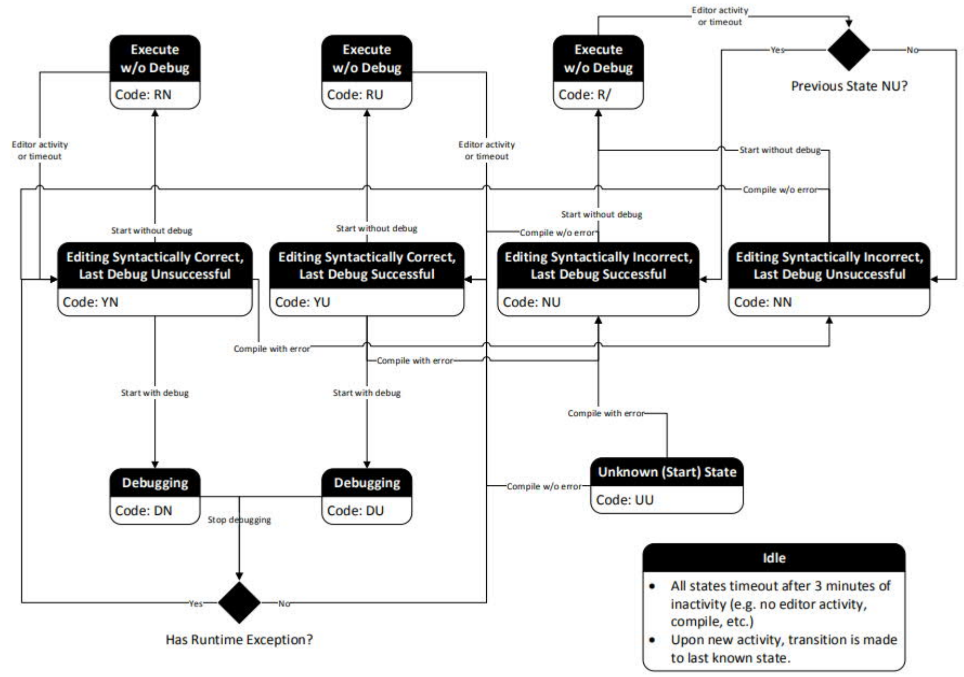


Figure 4. The NPSM state machine

While both the EQ and Watwin algorithm focus on compilation events that end in error, the NPSM focuses on the state the student is in (Carter, Hundhausen and Adesope, 2015). The model collects data on how much time each student spends in various combinations of debugging and editing program code, while also considering the previous compilation attempt. The intuition here is that errors and the ability to fix them are indicative of programming skill; programming, as a whole, consists of more than these two activities. Editing and debugging code play a major part in programming and the NPSM collects how much time a student spends in each of these states and provides a predictive measure based on this data.

The NPSM requires the use of an IDE with the capability of measuring time spent in each of the states, which was done with Visual Studio using an additional plugin to collect data on debugging (Carter, Hundhausen and Adesope, 2015). The requirement for such extensive preparation makes the model hard to apply in other contexts. BNPSM (Richards and Hunt, 2018) was developed to bridge this shortcoming. BNPSM does not require any debugging data and thus, is applicable to wider swath of contexts.

Carter (et al., 2015) determined which states correlate most strongly with course final grades. They reached the conclusion that the time a student spends in the start state and editing incorrect programs after a successful debugging were the most significant predictors. While the NPSM can be used to measure students' skill, a single predictive value is derived from the NPSM by using a weighted average of the unstandardized beta coefficient of each variable.

2.2 Machine learning

There are many different descriptions of machine learning (ML). Kirk (2017, pp. 15) describes it as "the intersection between theoretically sound computer science and practically noisy data." Morgan (2018, pp. 2) defines ML as "[r]ather than programming everything you want a computer to do using strict rule-based code, the ML process works from large datasets, with the machines learning from data, similarly to how we humans and other biological brains process information." Yet another description is that "[m]achine learning is about extracting knowledge from data." (Müller and Guido 2016, pp.1) What all these descriptions have in common is that the computer is presented data, and models are created from this data using algorithms. Some of the most popular algorithms for ML are described in the following sections.

Artificial Intelligence (AI) is a very closely related term, and often used in the same context as ML. These terms are not interchangeable since ML is only a subset of AI. The umbrella term AI refers to efforts to create software that functions as human intelligence does, only better. ML is a part of the quest for general AI, and contains the tools for extracting knowledge from large scale data. ML algorithms contain varied methods, such as clustering, deep learning, and regression. For example, deep learning is a specific approach within the ML algorithms to extract knowledge from big data using neural networks. Figure 5 illustrates the generally agreed upon relationship between the terms (Morgan, 2018).

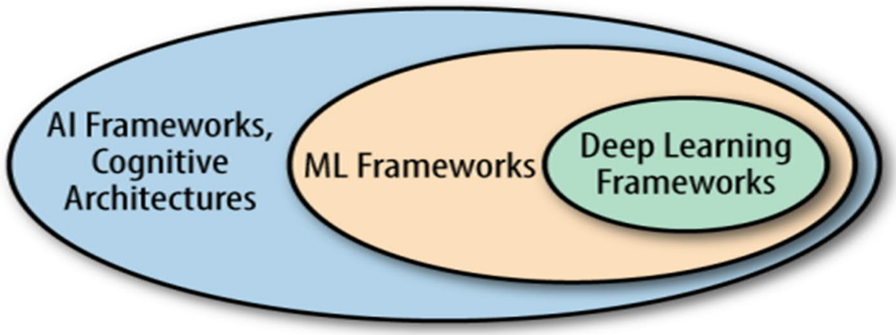


Figure 5. Comparing AI, machine learning, and deep learning (Morgan, 2018 pp.3)

Currently, in computer science, ML methods are applied in speech recognition, computer vision and natural language processing (Sindhu et al., 2020). However, practically each field of research, from biotech to social sciences to the human sciences can apply AI in their research.

Common terms in ML worth noting are *hyperparameter*, *feature* and *class*. Hyperparameters are parameters not found in the data, but are used to tune the learning process. Features are the aspects of a data point the ML is using to create a model. For instance, if an observation is a row in a database, the columns are features of that observation. Class, in ML, means a distinct category observations belong to. Often, especially in supervised learning, observations are a pair of the data point and the corresponding class. For instance, all observations representing a dog would be labeled ‘dog’ and all observations representing a cat would be labeled ‘cat’.

Machine learning algorithms can all be considered to perform one thing: minimize the difference between the learned model and the real data (even data points not yet encountered). A way to measure this difference is often called the cost function, but sometimes loss function or error function (Raschka & Mirjalili, 2019). Mathematically speaking, the model seeks to minimize the cost function.

2.2.1 Brief history of machine learning

Machine learning as a term took off in the 1960s. By the mid ’60s, ML was already established as a research field. Research was discussed in, for instance, the book “Learning machines” by Nils Nilsson (1965). Nilsson’s book was focused on pattern matching and some emerging aspects of ML (which are today known as) unsupervised learning, long short-term memory and fuzzy clustering, were left out completely (Sebestyen, 1966).

The 1970s began a shift away from equating pattern matching to ML (Duda and Hart, 1973). In a conference titled “Pattern recognition and ML”, saw only two papers (out of 30) directly related to pattern matching and classification (Gerhardt, 1974). Progress was still made and the general state of ML research was described by e.g. Nilsson (1980). The ’70s was also when AI and ML reached a body of knowledge massive enough to warrant statements such as: “[…] it is encouraging to discover that AI has progressed to the stage where a course requires a full year to do justice to the subject matter!” (Kant, 1980).

The 1980s saw increased research interest in especially neural nets, as researchers increasingly reported success with neural nets with hidden layers and the invention of backpropagation (Weiss and Kapouleas, 1989). Research efforts begun to include a wide variety of subfields alongside the traditional classification problems. Conferences for ML increased in number (both quantitatively and participant-wise), and the conference of the Association for the Advancement of Artificial Intelligence-87 (AAAI) had over 6700 participants and a wide variety of state-of-the-art AI research (Greiner et al., 1988).

In the 1990s, neural network research received the majority of ML research attention, even though the results were poor. Today, we of course recognize the utility of neural networks and deep learning, but in the ’90s, results were mediocre, most likely due to the lack of available computing power (Cook, 2016). ML research begun to research not only the methods and algorithms by which learning happens, but also what should be learned and researcher bias when developing these models (Provost and Kohavi, 1998).

The 2000s saw the explosive growth of the internet and World Wide Web and ML methods were applied in this context, for instance in intrusion detection (Tsai et al., 2009). Computational resources had also reached new heights and the massive parallel computation power of the graphical processing unit was harnessed to enable ML algorithms on bigger data faster (e.g., Steinkraus et al., 2005).

The 2010s saw widespread adoption of AI methods by real-world businesses, as opposed to being only available to governments, researchers and advanced tech companies (Ongsulee, 2017). Nearly all large businesses utilize some sort of ML. In finance it is used to predict stock prices (to varying degrees of success) but reports as high as 81% success (in making a profit) have been achieved. In transportation self-driving cars are making their first debuts on the streets and flight companies use ML methods to optimize flight routes. The health technology sector has access to massively big data from wearables and other such sources, which is ripe for analysis. (Morgan, 2018)

The 2020s will most likely see the continuation of wide-spread application of ML technologies to real-world problems. More and more jobs for humans will be replaced by automation and societal changes will be needed to accommodate this. Even creative jobs, such as computer programming, may be subject to extensive automation, as Google releases AutoML, which automatically creates AI models while needing little to no expertise in AI from the end-user (Google, 2018).

2.2.2 Supervised learning

Supervised learning is an umbrella term for all ML methods where the aim is to classify new observations using a dataset of previously classified observations. To accomplish this, a training dataset needs to be built which contains observations and the class said observation belongs to. Then, using this training dataset, a model is built which allows the prediction of new, unseen observations into the existing classes. Two main types of supervised ML problems exist: classification and regression. (Müller and Guido, 2016)

In classification problems, the aim is to group observations into distinct categories based on measured attributes. For instance, given a dataset of pictures of digits, the supervised learning algorithm must determine which digit the picture is. As computer do not see as humans do, the algorithm is given the picture in its binary form and the classification must be done based on it. If the model is trained on part of the dataset and predicts the rest, the accuracy of the model is easily computed, as the dataset contains the correct classification data. (Campesato, 2020 pp. 137)

Classification problems are either a multiclass or binary. Binary classification problems involve classifying an observation into one of two categories, whereas multiclass classification involves more than two target categories.

Regression problems involve a continuous variable, usually numbers, instead of a fixed number of categories. For example, given a dataset of observations for age, sex, occupation, and income a regression model could be built to estimate income based on age, sex and occupation.

The difference between regression and classification problems is in the number of possible categorizations. A continuous output variable has an infinite possibilities and a regression model should be used. A discrete output variable does not have an infinite number of possibilities and classification algorithms may be used. However, it is often not as clear cut as this; say we have a huge dataset with one million categories. While one million certainly is bounded, a data-analyst would need to look into the data whether regression or classification is the more appropriate route.

K-nearest neighbors

The K-Nearest Neighbors (KNN) algorithm is one, if not the simplest, machine learning algorithm. Essentially, the algorithm is a heuristic, as little to no mathematical or statistical grounds exist, however the method has been proven effective (Campesato, 2020, pp.177). K stands for the number of neighbors considered when making predictions. No special training phase is needed, as making predictions for new data points is as 'simple' as finding the K nearest data points in the training set and assigning the new observation to the same class as the majority of found neighbors. (Müller and Guido, 2018)

An illustration of how the KNN algorithm works is shown in Figure 6. The new observation is depicted as the star and the solid line indicates its neighbors at K=1, whereas the dashed line indicates its neighbors at K=3. The prediction for K=1 would be a circle, and for K=3the prediction would be a square.

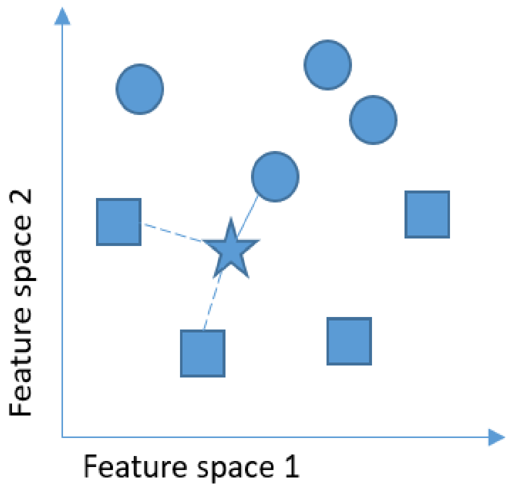


Figure 6. Neighbor selection in KNN with K=1 (solid line) and K=2 (dashed line)

Because the KNN algorithm classifies new observations based on its neighbors, the possibility of a tie exists. Imagine classifying cars and trains; even with K=1, it is possible a new observation has one car and one train at equal distances apart. Several methods for resolving ties exist, as suggested by Campesato (2020 pp. 177):

- Assign higher weights to closer points
- Increase the value of K until a winner is determined
- Decrease the value of K until a winner is determined
- Randomly select a class

There are two hyperparameters: the number of neighbors and the measure of distance. Traditionally the Euclidean distance is used and performs fairly well (Müller and Guido, 2018), however other distance measures exist.

Computational distances are computed directly from data structures, such as graphs or matrices. Often used computational distances are the Manhattan distance and the Levenshtein distance. Statistical distances compute statistical values and determine similarity in terms of statistical values. Popular statistical distances are the Mahalanobis distance and the Jaccard distance. (Kirk, 2017)

Regression analysis

Regression analysis attempts to predict a continuous variable from available data. Model construction is done by fitting a function to the data points such that the difference is minimized.

Linear regression is a subset of regression models. Linear regression attempts to fit the data to a straight line such that the difference between the data point and line are minimized (Müller and Guido, 2018). An example of fitting a line is shown in Figure 7.

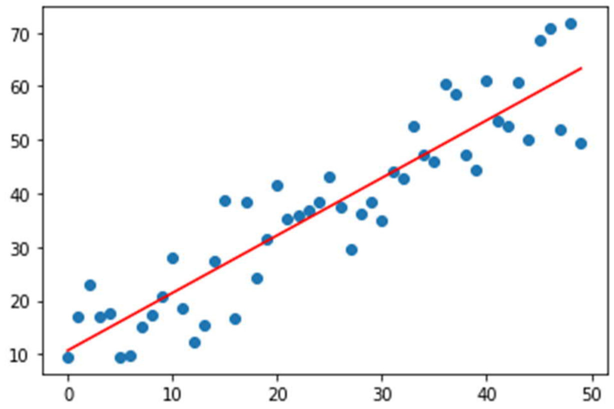


Figure 7. Linear regression of data shown as the line.

Decision trees

The decision tree algorithm is one of the simplest approaches in supervised ML. The decision tree is basically a flow chart that produces a classification for the data point in question. They are created by choosing one feature and choosing the best split for the data according to it a metric (Cook, 2016). A commonly chosen metric is the information gain (for examples in application, refer to Azhagusundari and Thanamani, 2013). Decision trees are called *classification trees* when the leaf nodes of the tree are distinct classes. When the leaf nodes represent a continuous value or a range of values, the decision tree is called a *regression tree* (Müller and Guido, 2018).

A very simplified decision tree model is shown in Figure 8. The hypothetical model was trained on three features to classify into four classes.

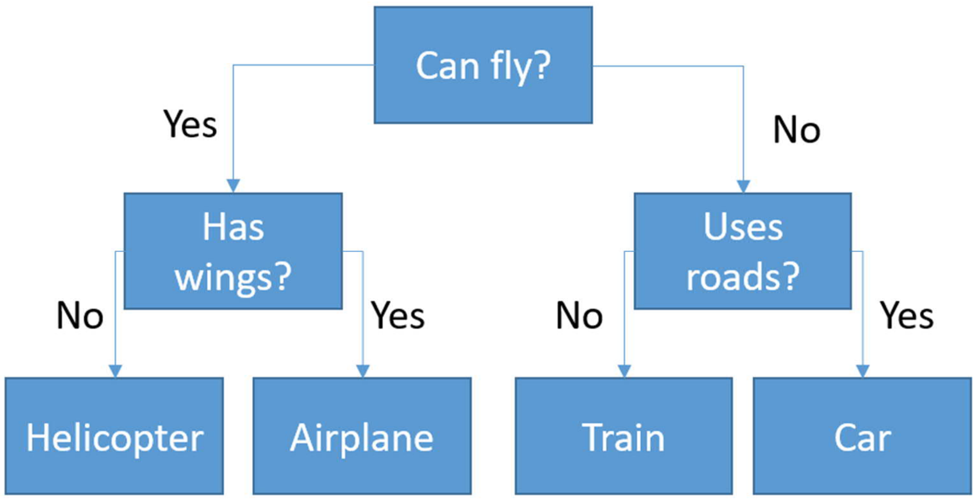


Figure 8. Simple decision tree to classify vehicles

2.2.3 Unsupervised learning

Unsupervised ML involves data without any labels or classes and is aimed at discovering these. The discovery process is often done in terms of clustering, which attempts to group similar datapoints together, or dimensionality reducing techniques, whose purpose is to reduce the number of features while retaining maximal variation in the data. Anomaly detection is an interesting subset of clustering, wherein the ML algorithm is tasked to find typical and untypical usage patterns (Zimek and Filzmoser, 2018). Anomaly detection can be used to identify, for instance cheating on applications (financial, educational, etc.) or denial of service attacks against websites (Campesato, 2020, pp. 138).

Typically, any collected data only contains the features, but no labels or other class information. This is the case, for instance, with e-commerce businesses, who know the purchase history and personal information of their customers. This data can be used to create a model which groups customers with similar purchase histories or interests together, which allows the business to focus similar marketing to these similar groups of customers. (Kirk, 2017)

Unsupervised ML methods are used to discover features about the data (Kirk, 2017). Clustering methods are needed, if the dataset is to be labelled. Unsupervised methods can be used to find clusters, which a human then assigns a label to. Feature extraction, of dimensionality reduction is applied to datasets with hundreds, or even thousands, of features to discover the most meaningful ones or possibly interconnectedness between features.

Hundreds of different unsupervised ML algorithms exist. One possible broad categorization is a division to clustering algorithms and dimensionality reduction algorithms. What follows is a brief overview of clustering and dimensionality reduction techniques.

Clustering

Clustering is the process of dividing a dataset into groups of points that are similar with each other, but different to those from other groups (Müller and Guido, 2018, pp. 170). Clustering is often done when familiarizing with the data; while the clustering algorithms cannot tell what the clusters describe or contain, they do reveal how many and how large they are. This information can then be used to drive further research into the data.

Clustering methods have some weaknesses as well. John Kleinberg (2002) introduced the impossibility theorem, which states that any clustering is always imperfect. He gives three metrics *richness*, *scale invariance* and *consistency*, from which only two may be maximized at any time. Richness refers to the best clustering possible; more richness means more distinct clusters. Scale invariance refers to the relationship of data and clusters; the same clusters should happen even after applying a function to the data. Consistency refers to the distances within clusters; if a function is applied to the distances between points, the same clusters should emerge. K-mean clustering also forces clusters to have a hard boundary and prefers spherical data which is centered around its mean, which cause the algorithm to perform poorly on some data sets (Kirk, 2017).

K-Means clustering is a simple and often used algorithm. It was first used by MacQueen (1967). The algorithm has three steps: first creating clusters around the current chosen cluster centroids (which may be randomly chosen the first iteration).

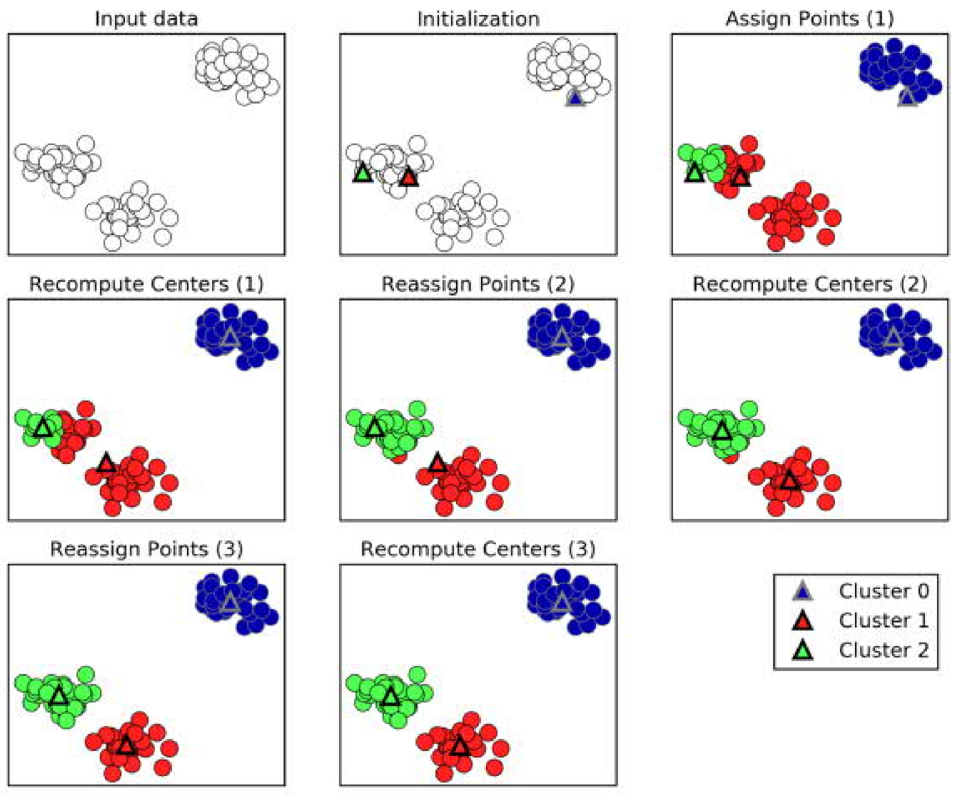


Figure 9. Three iterations of the k-means clustering algorithm (Müller and Guido, 2018, pp.171).

Second the algorithm assigns points to the current cluster centroids that are closest to it. Lastly, it moves the cluster centroids to the center of the newly created clusters. This process is then iterated for as long as the cluster centroids move as shown in Figure 9.

Dimensionality reduction techniques

Sometimes datasets have thousands of features. The sheer number of features gives rise to what Richard Bellman calls the *curse of dimensionality* (Bellman, 1957) and the data needs to be transformed to reduce the number of features and enable computation. Dimensionality reduction techniques are then needed to find a minimal set of features that preserves maximal variance in the data.

A common dimensionality reduction technique is Principal Component Analysis (PCA). The aim of the technique is to rotate the data so that the new data points align with axes that represent important features (Müller and Guido, 2018). The result of PCA is a new vector space with fewer axes that still capture the variance of the original data.

Another often used technique is Non-negative matrix factorization (NMF). It is an unsupervised ML technique similar to PCA, in that it can be used to reduce dimensionality, but it can additionally be used for feature extraction. NMF works by finding a weighted sum of some of the components, as in PCA, but with the difference that all coefficients and components are non-negative. This allows for decomposing data into its constituent 'signals'. This method has successfully been applied to audio tracks and other such sources which are composed of aggregated signals. (Müller and Guido 2018)

A dimensionality reduction technique, especially useful for visualizing multidimensional data, is called t-distributed stochastic neighbor embedding (t-SNE). It is described by Van der Maaten and Hinton (2008) to be superior to a number of previous dimensionality reduction techniques. It differs and improves on the normal SNE in two critical ways: 1) it uses a symmetrical cost function, which is easier to compute and 2) it uses a different distribution to compute similarity between two points.

2.2.4 Model evaluation

Once a model has been created with ML models, it is important to validate it. Ideally, this is done with what is called *ground truth* – a dataset, which is already correctly labelled. Validation is performed by comparing the results from the developed model against the ground truth. Often, especially with unsupervised learning, no such ground truth exists and other methods are needed.

With supervised learning, when some source of ground truth exists, often used metrics are the *Receiver Operating Characteristics curve*, and the *Precision-Recall curve*. They both work by comparing true matches to false matches. (Campesato, 2020, pp. 187)

Unsupervised learning has fewer metrics. One is called the *silhouette coefficient*, which assesses the compactness of the created clusters, but often does not work well in practice. Real world data often does not cluster the data into compact clusters, so cluster inspection by hand is often a valid option (Müller and Guido, 2018, pp.195).

Other useful methods of validating clusters created by unsupervised machine learning algorithms are dimensionality reduction techniques such as Principal Component Analysis and Non-negative matrix factorization. After the dimensionality reduction has been done, the resulting new axes or factors may be manually inspected. Because the axis should share common factors, manual inspection of them often leads to new insight on how the model works.

For regression models which predict a continuous variable, an estimate of accuracy is needed. One possible method to estimate this accuracy is the r*oot mean square error*, which squares the error between the actual value and prediction before computing the mean, and takes the square root of the squared mean. (Combs, 2016, pp.213)

2.3 Learning analytics

Learning analytics (LA) nowadays covers a broad range of topics. For the 1st International Conference of Learning Analytics and Knowledge, it was defined as

> "Learning analytics is the measurement, collection, analysis and reporting of data about learners and their contexts, for purposes of understanding and optimising learning and the environments in which it occurs. " (Conole et al., 2011).

The modern definition of LA is more nuanced: Lang et al. (2022) edited the second edition of the Handbook of Learning Analytics, which contains some 240 pages of what LA is and is used for.

LA is thus aimed at several demographics: The students, whose data is being collected and analyzed, and ultimately whose learning outcomes LA seeks to improve. The teachers, who seek to gain insight into the learning progress of their students. The administrators, who can track changes in learning outcomes after changes in the learning environment. Finally, the researchers, who are given another tool to measure learning.

LA also comes with its own set of challenges. Possibly the most important challenge is that of ethics (Cerratto & McGrath, 2021; Alwahaby, et al., 2021; Tzimas & Demetriadis, 2021). Slade and Prinsloo (2013) give several overlapping categories of the ethical considerations LA should take into account: data and its storage, informed consent in research and privacy.

The model presented in this thesis belongs to the categories of predictive LA and explanatory LA. Predictive LA aims to predict future outcomes by collecting, analyzing and creating a model from data (Clow, 2013). This is to be differentiated from explanatory LA, wherein the purpose is not to predict outcomes, but explain reasons behind the current outcomes (Brooks & Thompson, 2017)

3 The Model and Data

This section describes the developed model, justification for why its creation was necessary and how the data was collected. Section 3.1 justifies the need for creating a new model and gives details on the data collection process and best practices. Section 3.2 describes the collected data and the collection process. Section 3.3 describes the developed model that was used in publications III, IV and V

3.1 The development of the model

This section describes the model and its development. Section 3.1.1 discusses the reasoning behind creating a new model and why the existing models were not sufficient. Research question 1 is addressed by providing a description of what the starting point for this thesis was, and also how the new model came to be and why a new model was needed. Section 3.1.2 addresses research question 1 by providing details on what is needed to collect data, what kind of data and how the data collection was done for this model. The section also describes the LMS used to collect the data.

3.1.1 Publication I

In the mid-2010s, research begun on learning analytics in the Department of Information Technology in the University of Turku. Special focus was given to applications of learning analytics in mathematics and programming. Soon thereafter, a model for clustering students in primary school mathematics was developed (Lokkila et al., 2015). Next, a similar model was needed for programming. Turns out analyzing programming students is more complicated than analyzing primary school mathematics students.

Publication I of this thesis lays the foundation for the model in programming. The study examines differences between novice and expert programmers and proves such differences do exist. However, the differences are more subtle than in mathematics, where a student either arrives at a solution or not. In programming a novice student takes detours around the possible solution space, whereas the expert seems to approach a correct solution in a more linear way. Additionally, experts provide a solution that is more abstract and general; i.e., applicable to multiple similar situations, whereas novices often provided a solution for the specific problem only. The practical problem at this point was how to apply this knowledge to create a model.

Research into the field provided several good candidates for models: the EQ (Jadud 2006), the NPSM (Carter, Hundhausen & Adespoe, 2015) and the Watwin score (Watson, Li and Godwin, 2015). Later, also RED (Becker, 2016) was considered. While these models provided moderately good predictive power for academic success, none of them ultimately offered what interesting capabilities in identifying students who behave similarly in programming. All the existing models provided only a single value to work with. Using this one-dimensional value proved too difficult with too many unanswerable questions: What values should be used as the center point for clusters? How big a difference to the centroid is acceptable? These questions drove me to develop my own model.

The starting point for my model was the findings in Publication I of this thesis. Experts were able to solve the given problems with far fewer (on average) attempts than novices; additionally, the experts attempts were more focused on the problem solving than struggling with the syntax of the language. Thus, modelling this behavior was key in separating programmers of different skill levels. After many years of failed attempts, such as parsers, static and dynamic code analysis, and abstract syntax tree analysis, I turned my efforts to apply ML to this problem.

The key to the application of ML is to create vectors of the data points. To achieve this a combination of the EQ, Watwin, RED and NPSM algorithms was used. The developed model uses a state machine, like in the NPSM, but tracks state changes like in the EQ and Watwin algorithm. The RED algorithm gave me the idea to evaluate the tracked state changes only at the end. Thus, the transition matrix saw light of day. Instead of the single value offered by the other models, the transition matrix had 29. This number of variables is enough for any unsupervised machine learning algorithm.

The starting point for the discovery process for the model for this thesis was publication I. The next section delves deeper into the data collection process and tools used to collect the data.

3.1.2 Publication II

As research begun in learning analytics in the mid-2010s, at what is currently known as the Turku Research Institute of Learning Analytics (TRILA) at the University of Turku, the first task was to begin collecting data. This was done in the form of course reforms, in which technology assisted learning and active learning were hugely incorporated into the course curriculums. Several existing courses were reformed during this time. In addition to publication II from this thesis, the reformed courses included a databases course (Lokkila et al., 2017), object-oriented programming course (Kaila et al., 2016), and an introduction to computer science course (Lokkila et al., 2016). All future programming courses were also created following the same reform guidelines.

The main tool with which learning analytics data was collected and applied to research was a LMS called 'ViLLE'. It was created in the University of Turku in 2005, and development has been ongoing ever since. An extensive description of the system can be found in (Laakso et al., 2018). The main features of ViLLE, in terms of programming education, are its automatic assessment system for programming exercises.

Automatic assessment was heavily used in the course reforms and as each automatically assessed exercise had to be stored in a database, massive amounts of data were gathered as an artefact. The collected data was then used to further the research of learning analytics. The relevant data collected, that is also used in the forming of the model of this thesis, is described in Section 3.2.

The models described in Sections 2.1 (excluding the Markov models) were considered to be applied to the courses' automated assessment. However, they were not suitable for our needs, as most of them only provides us with a single value describing skill. NPSM offered us multiple values as times spent in the states, but we were not able to implement the algorithm due to technical requirements. Specifically, we were not able to collect accurate enough time usage data on when the students were editing, executing, and writing code in ViLLE. Simply put, the existing models did not provide us enough information to allow the effective application of ML models to gain deeper insight into the novice programmer.

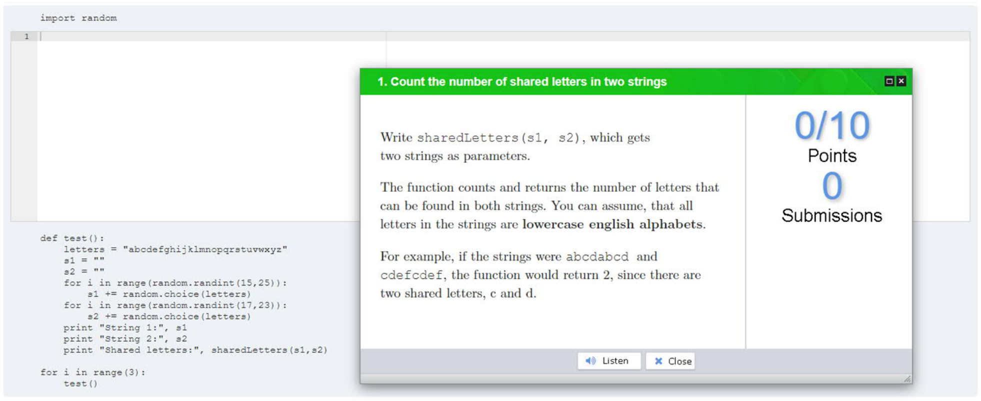
In publication II we introduced active learning to an introductory course to CS for engineering students. Instead of purely lecturing the course, students were to complete small assignments in ViLLE to test their understanding of the topic. The incorporated changes resulted in decreased failure rates and a higher average grade for the course. This was a clear indicator that active learning and technologically assessed learning was the way forward. The incorporated exercises included Parson's problems, programming exercises and questions with either multiple-choice or short open-ended answers.

Parson's problems are exercises wherein the student is provided a scrambled set of lines of code, which must be assembled in a specific order to complete the programming task given in the description (Parsons & Halden, 2006). Not all provided lines of code are necessarily required in the answer, which allows the instructor to provide the students with options for common misconceptions. Parson's problems on the course in question in publication II were given in Python. The Parson's problems were modified slightly by parametrizing specific lines of code;

specifically, the for-loops and if-statements. For instance, the Python range-method takes an exclusive second parameter, so the for-loop line in question was parametrized with options for both the correct and several incorrect distractors for students. Providing students with lines for their common misconceptions is an effective way of correcting their misconceptions (Parsons & Halden, 2006).

The course reforms also included incorporating a multitude of multiple-choice questions (MCQ) into the course. All MCQs were automatically assessed using ViLLE. The term MCQ here is used to include the traditional MCQs, as well as the open-ended question type, where students are not given a list of options, but must type in their own. Well-designed MCQs are excellent in testing students for factual knowledge (Scouller, 1998). However, when used as an assessment tool it may give students the wrong signals on what learning is important (Scouller, 1998). Hence, the exam itself contained only a few MCQs and was more geared towards practical assignments, wherein students had to apply what they had learned. The MCQs on the reformed course were not all multiple choice in the traditional sense; some were given as an open-ended question wherein students had to type in the correct answer. The automatic assessment employed in these cases was a simple regular expression check: the instructor gave the correct answer as a regular expression and if the students' answers matched this expression, they were given points. This approach effectively made the MCQ to have an infinite number of possibilities to choose from.

The programming exercises in ViLLE are typically short exercises, wherein the student is given a template and asked to complete the program by writing one or more methods according to specification (Figure 10). The programs are then executed and the output of the student and teacher code are compared. Even though only the output is compared and points are awarded based on it, this does not mean the written methods are not tested individually. Indeed, often the template itself contains unit-test-like methods specifically written to test the students' method on various inputs. The tested inputs often include edge cases, erroneous values, and random values. Edge cases are tested because these situations are the typically the most error-prone; such as off-by-one errors in indexing an array or list. Erroneous values are tested, as these force the student to write error-handling code and thus think about the problem in a more wholesome way. Random values are included in virtually every test, as this forces the student to write more general algorithms, instead of those that work on a specific value.



```
import random

def test():
    letters = "abcdefghijklmnopqrstuvwxyz"
    s1 = ""
    s2 = ""
    for i in range(random.randint(15,25)):
        s1 += random.choice(letters)
    for i in range(random.randint(17,23)):
        s2 += random.choice(letters)
    print "String 1:", s1
    print "String 2:", s2
    print "Shared letters:", sharedLetters(s1,s2)

for i in range(3):
    test()
```

1. Count the number of shared letters in two strings

Write `sharedLetters(s1, s2)`, which gets two strings as parameters.

The function counts and returns the number of letters that can be found in both strings. You can assume, that all letters in the strings are **lowercase english alphabets**.

For example, if the strings were `abcdabcd` and `cdefcdef`, the function would return 2, since there are two shared letters, c and d.

Figure **10**: A programming exercise in ViLLE

The data collected from submissions to ViLLE exercises consists of four data points: 1) the user identifier 2) the submission data (e.g., the code for programming exercises) 3) the timestamp 4) the awarded points. These data points alone are enough for simple learning analytics tasks such as learning tracking and activity monitoring. The data used specifically in generating the model will be described in the next section.

3.2 Data used in developing the model

The model itself has been machine learned using the k-means algorithm. The quality of the model depends on the quality of the data. As such, this section describes the data and the data collection processes. In total, submission data from 5 introductory programming courses spanning 3 educational institutes were included in this thesis. The courses are described in detail in Table 2. All institutes are located in Turku, Finland.

Table 2: Description of courses used to create datasets

Code	Course name	Language	Institution	Academic year
A	Algorithms and Programming	Java	University of Turku	2017 – 2018
B	Algorithms and Programming	Java	Open University of Turku	2017 – 2018
C	Programming	Python	Turku University of Applied Sciences	2020 – 2021
D	Introduction to Programming	Python	University of Turku	2020 – 2021

E	Introduction to Programming	Python	University of Turku	2021 – 2022

In total, 3 datasets created from the 5 courses were used in creating this model. The details of the datasets are indicated in Table 3. The total combined number of analyzed unique student code submissions is slightly over half a million (544 835). Submissions were received from a total of 1174 unique students to a total of 411 assignments. An overview of the data sets is shown in Table 3. The data collected is presumed to contain very little noise in terms of students writing and submitting code other than what the exercise requires. It is worth noting that Datasets 1 and 2 also includes answers to weekly surveys, which are not counted in the total code submission counts.

Table 3 Overview of all datasets used in this thesis.

Publication	Dataset	Courses	Students	Assignments	Total submissions
III	1	A,E	757	238	450 101
IV	2	E	467	152	325 404
V	3	A,B,C,D	715	259	219 454

3.2.1 Publication III

The dataset for this study was unlabeled. The data for this study was collected from two courses over two academic years 2017-18 and 2021-2022. The courses were given at the University of Turku. Both courses were given over 8 weeks, with the last week reserved for the final exam.

Data collection was performed using the LMS ViLLE on both courses. ViLLE was used to deliver both surveys and programming exercises to students. The programming exercises were automatically assessed by ViLLE and saved to a database, form which the dataset was then collected.

The dataset contains student answers to a questionnaire given to the students on the first week of the course, which asked for an estimate of their previous programming experience in two ways. One question was a Likert scale of 1 to 5, where 1 meant no programming experience and 5 meant expert. The other question asked for a list of programming languages the student felt proficient with.

The dataset also contains automatically assessed submissions to programming exercises. Each attempt the student gave to solve the exercise is called a submission. The following data points were collected for each submission:

- the program code
- submission timestamp
- submission correctness
- user identification

From these four data points, further data points can be inferred by compiling and executing the code. This gives data points such as whether or not the code executes completely, faced a compiler error or a runtime error. Each submission is allocated two seconds of real time for code execution. If the code never finishes during this time, the code execution is halted and the submission is labeled as timed-out.

3.2.2 Publication IV

The dataset for this study was partially labeled, due to the open-ended answers provided by the students themselves. The data for this study was collected from the course 'Introduction to Programming' given at the University of Turku in fall 2021. The course is aimed at first year computer science students and is the first programming course they take. The programming language on the course was Python. The dataset was collected using ViLLE as the LMS on the course. The data consists of code submissions to programming exercises given on the course as well as answers to a weekly survey. The survey explored the following:

- Open ended: How do you feel about programming thus far?
- Likert scale 1-7: How difficult do you estimate this week?
- Multiple choice: What programming IDE did you use this week?

A total of 325 404 coding submissions were collected from 467 students answering 152 assignments. Of all collected submissions, 173 155 (53%) did not contain compilation or runtime errors, 6753 (2%) were timed-out after two seconds of program run time, and the rest of code submissions had either a compilation or a runtime error.

A total of 2392 answers were collected to the weekly survey, averaging 341 answers per week. Signs of survey fatigue are visible, as the first weeks received substantially more answers to the survey than the latter weeks. Regardless, a total of 1459 answers were received in the open answer field, averaging 208 answers to the open-ended questions every week.

Answers to the open-ended question were codified following grounded theory (Strauss & Corbin, 1997), wherein the coding categories are not predetermined, but discovered in the data. Two categories were discovered: engagement and difficulty. Labels were created for a three-point scale for both categories. The following labels were created for engagement, its score in parenthesis: *Disengaged* (0), *Neutral* (1), *Engaged* (2). The following labels were created for the difficulty category, its score in parenthesis: *Easy* (0), *Neutral* (1), *Difficult* (2). After coding was complete the mean values for each category were computed and they are shown in Table 4 with the number of answers in parenthesis.

Table 4 Overview of survey data for dataset 2

	Average Engagement	Average Difficulty
Week 1	1.95 (341)	0.40 (48)
Week 2	1.95 (275)	0.87 (76)
Week 3	1.86 (209)	1.15 (108)
Week 4	1.83 (192)	0.98 (86)
Week 5	1.59 (169)	1.59 (49)
Week 6	1.66 (138)	1.59 (56)
Week 7	1.88 (107)	1.39 (39)

3.2.3 Publication V

The data set for this study was unlabeled. The data for this study was collected from three institutions: a) University of Turku b) Turku Open University c) Turku University of Applied sciences. Data was collected between the years 2017 and 2020 from a total of 4 courses (Table 5). All institutions used ViLLE as the LMS of choice. The data collection process was handled by ViLLE. The students were given programming exercises in ViLLE, and they submitted their answers to ViLLE, which then automatically assessed the correctness of the submitted programming code. In total, 219 454 total submissions were collected from 715 students to a total of 259 programming exercises. The following data points were collected for each submission:

- the program code
- submission timestamp
- submission correctness
- user identification.

All courses lasted eight weeks with the last week reserved for the final exam. The courses were all introductory programming courses given to first year students. Structurally they were remarkably similar: All started by introducing variables, then conditionals, followed by loops on a weekly basis. Next methods were introduced, followed by data structures like lists and hashmaps/dictionaries. Other topics, such as error and file handling were also discussed briefly.

Table 5: Overview of data from Dataset 3

Course	Institution	Year	Students	Assignments	Submissions
Algorithms and Programming	University of Turku	2017	294	86	124 697
Algorithms and Programming	Turku Open University	2018	58	73	24 291
Introduction to programming	University of Turku	2020	92	55	21 164
Introduction to programming	Turku University of Applied Sciences	2020	273	45	49 302

3.3 The developed model

The developed model is heavily based on Markov Chains. The idea is simple: Model each student as their own Markov Chain. Each student produces a chain of submissions, where the next submission is always independent of the previous ones, thus fulfilling the Markov property. The Markov property is fulfilled, because the student always has, before resubmitting, a chance to revise the submission according to their skill level. Therefore, as the student performs their random walk on the model, we are able to collect the Markov Chain from which the transition probabilities are then computed.

The developed model consists of two parts: 1) a state machine describing the Markov model and 2) the transition matrix, to which all transitions in the state machine are aggregated. There is no minimum session length for collecting submissions. However, results become more accurate with more data. Thus, a session could be all submissions to a single assignment or all submissions collected during a course.

The model is assumed time-homogenous, as the data is collected during one course of seven weeks. While the student (hopefully) learns programming during the course, the learning is not significant enough to drastically affect the transition probabilities. This assumption may not hold when collecting longitudinal data, i.e., same students over multiple courses.

The state machine contains eight states and all possible state transitions are described in Table 6. Each exercise begins in any of the start states and may end in any of the given states. The start states are state 1, state 2 or state 8, depending on

the first submission. State 8 is an absorbing state, which means it has no outgoing transitions.
The purpose of each state is as follows:

1. Met success; the student succeeds in executing the code without a compilation or runtime error after one or more erroneous submissions.
2. Met error; the student fails to execute the code successfully and encounters a compilation or runtime error after one or more successful submissions.
3. Confused; the student receives a compilation error after one or more compilation errors.
4. Repeated success; the student succeeds in executing the code without compilation or runtime errors after one or more successful submissions.
5. Runtime error; the student encounters a runtime error.
6. Unmodified error submission; the student resubmits an erroneous submission with no code modifications.
7. Unmodified success submission; the student resubmits a successful submission with no code modifications.
8. Completion on first attempt; the student succeeds in the assignment on the first attempt (i.e., receives full points). Stored as a percentage of exercises completed on the first try.

Because we do not initially know the transition matrix for the student, we let them do a random walk on the matrix and collect the results. The random walk through the Markov chain here is the student performing submissions to programming exercises.

The transition matrix is then built using the state machine. As the student creates submissions, each state transition is added to the matrix. Once all submissions are made, the transition matrix is normalized to a right stochastic matrix, which means the sum of each row equals one. This is achieved by dividing each cell on the row by the sum of the row. The end result is the transition matrix, where each cell has a value between 0 and 1, inclusive. A vector of these matrices can then be used, for example, as input to unsupervised ML algorithms such as K-means clustering.

The transition matrix can also be used to derive a single value representing the student's current state. To compute this value, the states need to be labeled into either an error state or a non-error state. Thus, the following four categories are created (Table 6):

- Non-error to error (NE2E)
- Non-error to non-error (NE2NE)
- Error to error (E2E)
- Error to non-error (E2NE)

Table 6: The state machine. Transitions happen along the rows.

	1	2	3	4	5	6	7
1		NE2E		NE2NE	NE2E		NE2NE
2	E2NE		E2E		E2E	E2E	
3	E2NE		E2E		E2E	E2E	
4		NE2E		NE2NE	NE2E		NE2NE
5	E2NE		E2E		E2E	E2E	
6	E2NE		E2E		E2E	E2E	
7		NE2E		NE2NE	NE2E		NE2NE

The reduction can now be computed using the categories: subtract the sum of error transitions from the sum of non-error transitions and add the percentage of assignments wherein the student succeeded on the first attempt (Formula 1). The value is typically a value between -1 and 1, where low values mean high error rates and high values low error rates. Values over 1 are possible when the proportion of assignments completed on the first submission is high and very few errors were done on the assignments which were not completed on the first attempt. When applying the method, and in the included studies, this value was always bounded between -1 and 1; values over 1 were set to 1.

$$\sum NE - \sum E + FirstAttempt \qquad (1)$$

Next an example of the computation of the transition matrix and the reduction. Suppose the following submission history from a student to one exercise: S,E,E,E,S; where S stands for a successful submission and E, for simplicity's sake, a submission with a compile error. The first step in the algorithm is to create submission pairs. The following pairs are created: (S,E), (E,E), (E,E), (E,S). The next step is to aggregate the state transitions into the transition matrix (Table 6, left). To complete the transition matrix, all rows are normalized by dividing each cell on the row by the sum of the row. Using the same states as described in Table 6, the final transition matrix is shown on the right in Table 7 (left).

Table 7: Aggregated state transitions on the left; final transition matrix on the right

	1	2	3
1		1	
2			1
3	1		1

	1	2	3
1		1	
2			1
3	0.5		0.5

The reduction can then be computed from the transition table (Table 7, right) as the sum of all columns subtracted by the sum of all transition probabilities to success states. Specifically, this means we sum column 1 and subtract the sum of columns 2 and 3. The final score for this student is then computed as 0.5 – (1+1+0.5) = -2

4 Model analysis

This section analyzes the model and proves the model succeeds in meaningfully clustering students. First, in Section 4.1 the distribution of data is proven to be normally distributed. Second, in section 4.2, the model is shown to perform similarly to other metrics through a correlation analysis. Lastly, dimensional reduction techniques are applied to the transition matrix created by the model in Sections 4.3, 4.4 and 4.5. These techniques verify the model does indeed contain valuable insight into novice programmer behavior.

4.1 Normality analysis

The transition matrix is, in itself, a 29-dimensional space, which consists of all the non-zero states in the transition matrix (see Table 6) and the separate state 'success on first attempt'. In order to visualize transition matrix data, a dimensionality reduction needs to be performed; common methods are t-distributed stochastic neighbor embedding (t-SNE), Principal Component Analysis (PCA), and Non-negative Matrix Factorization (NMF).

The clustering method used to cluster the data is K-Means clustering. When clustering students into similar skill groups, the number of clusters is chosen to be k=3. The reasoning behind k=3 is two-fold: first, results from Moubayed et al. (2020), determined the suitable number of clusters to be 3 for their programming MOOC data. Second, when using the transition matrix reduction to estimate student programming skill, it has a Gaussian distribution. As the K-Means clustering algorithm divides the variable space into k-equal parts, the result of the clustering effectively divides the cohort into the best and worst groups (the two tails of the distribution) and the average group (Figure 11). The normality analysis was done using the Shapiro-Wilks test for normality, which revealed that the data is normally distributed in terms of the reduced value (Shapiro-Wilks-test: Python, W=0.96, df=1540, p<.001; Java, W=0.916, df=645, p<0.001). The distributions of transition matrix reduction within the clusters are also normally distributed as tested by the Shapiro-Wilks test for normality and visual analysis of Figure 1.

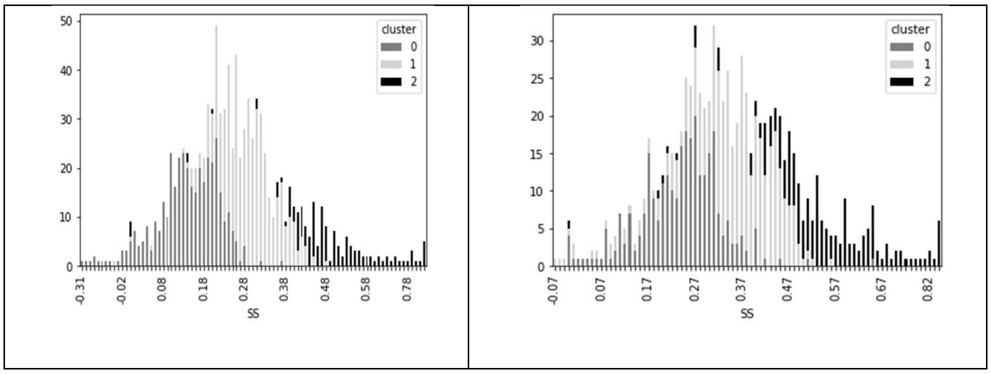


Figure 11: Histograms of the distribution of the value from transition matrix reduction. Java data on the left and Python on the right.

4.2 Comparison to existing metrics

A correlation analysis is done to verify whether the method proposed in this study is at least as valid as the existing metrics described in sections 3.1.2, 2.1.4 and 2.1.5. The validity of the model is analyzed by comparing the reduced value to the existing metrics.

The correlation analysis was done using Pearson's correlation coefficient. The data from each course, as described in section 3.2, was used to compute the correlation coefficient. All correlations were found to be negative and mostly moderately correlated. The smallest correlation was found to be with RED, whereas the other two metrics were more strongly correlated. The EQ had, on average, a stronger correlation. The coefficients can be found in Table 8.

Table 8: Correlation coefficients between the matrix reduction (section 3.3) and other metrics

Course	EQ	Watwin	RED
A	-0.72	-0.675	-0.546
B	-0.868	-0.821	-0.673
C	-0.591	-0.546	-0.297
D	-0.664	-0.554	-0.426
E	-0.779	-0.686	-0.556

4.3 t-SNE analysis

The dimensionality reduction technique t-SNE can be used to visualize datasets with more than three dimensions. The dimensionality reduction is performed by computing a transformation on the data which moves similar points closer to each other and dissimilar points further apart.

A t-SNE dimensionality reduction revealed the transition matrices produced by the two languages are distinctly different (Figure 12). This is further demonstrated in publication V, that Python is a more suitable first language in terms of student errors and ease of syntax. The two languages differ in one major aspect: traditionally Java is categorized as a compiled language whereas Python an interpreted language. This indicates that the model captures differences in student programming behavior when using these languages.

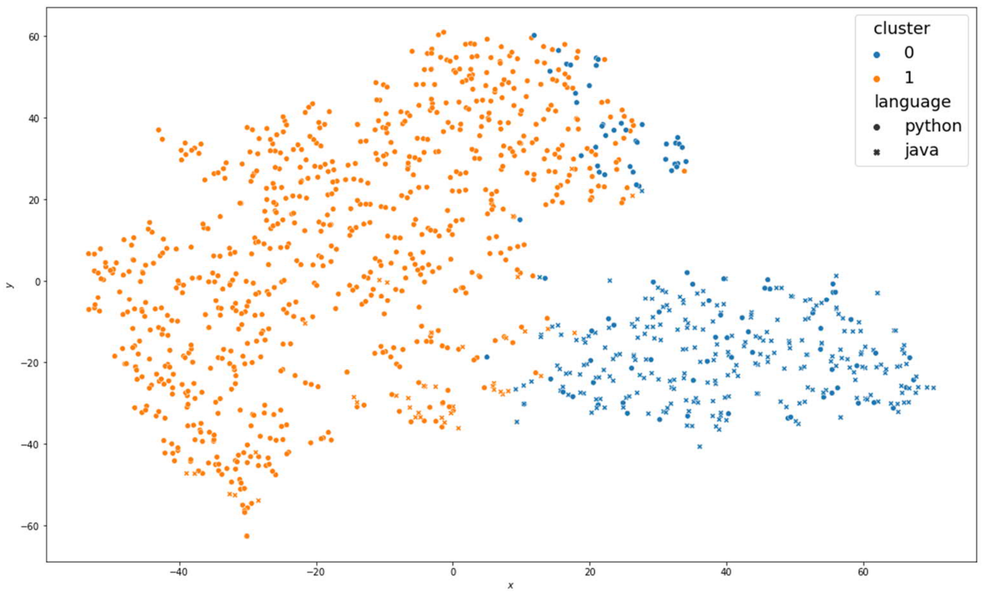


Figure **12**: The transition matrix variable space reduced to two dimensions. Clustering was done on original data before dimensionality reduction.

To determine how accurately the model is able to differentiate between Java and Python programmers, an error matrix (also known as a confusion matrix) was computed in Table 9. The model was very successful in separating between Java and Python programmers. The total accuracy of the model is 0.876, and its F_1-score is 0.774. However, as the dataset is unbalanced with over twice as many Python students as Java students, the Matthews correlation coefficient (MCC) may be a more suitable metric. The MCC takes into account the ratios of all cells in the error matrix and produces a value between -1 and 1, similarly to any correlation coefficient (Chicco, 2017). To confirm the model’s ability to recognize Java programmers from Python programmers, a chi-square test of independence was performed. The model was found to separate the two languages extremely well (X^2 (1, N = 1174) = 563.759, $p < .0001$).

Table 9: Error matrix of clustering all datasets into java and python programmers

		Real condition	
		Java	Python
Predicted condition	Java	249	100
	Python	45	780

4.4 Principal Component Analysis

The transition matrix contains 29 independent variables, which together describe the student's behavior when programming. In other words, the 29 variables create a 29-dimensional search space. However, some of the 29 variables may correlate with one another and thus, the same dataset could be described with fewer dimensions by dividing the search space using new axis.

The search for these new axes is called a Principal Component Analysis (PCA). Each principal component (PC) is an axis in the search space, which has been drawn so that the axis describes as much variance in the data as possible. Each axis can thus be understood as a combination of the original variables. Because the t-SNE analysis showed distinctly different transition matrices for Python and Java, PCA will be performed separately on the two languages. PCA requires the data be normalized to give valid results. Fortunately, the transition matrix is already normalized (all values between 0 and 1).

Because PCA is a dimensionality reduction technique, some loss in information is inevitable. Figure 13 shows the explained variance of each PC after performing the PCA.

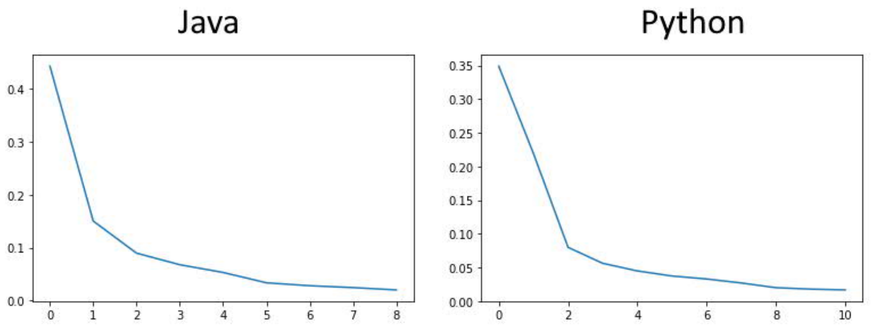


Figure 13: The explained variance (in percentage) for each principal component for Java data (left) and Python data (right).

For the Python data, 11 PCs were needed to capture 90% of the variance in data. For Java, 9 PCs can be used to capture 91% of the total variance in the data. However, since as much as 60% of the variance could be explained with only three PCs for both datasets, the analysis was focused on them (Figure 14).

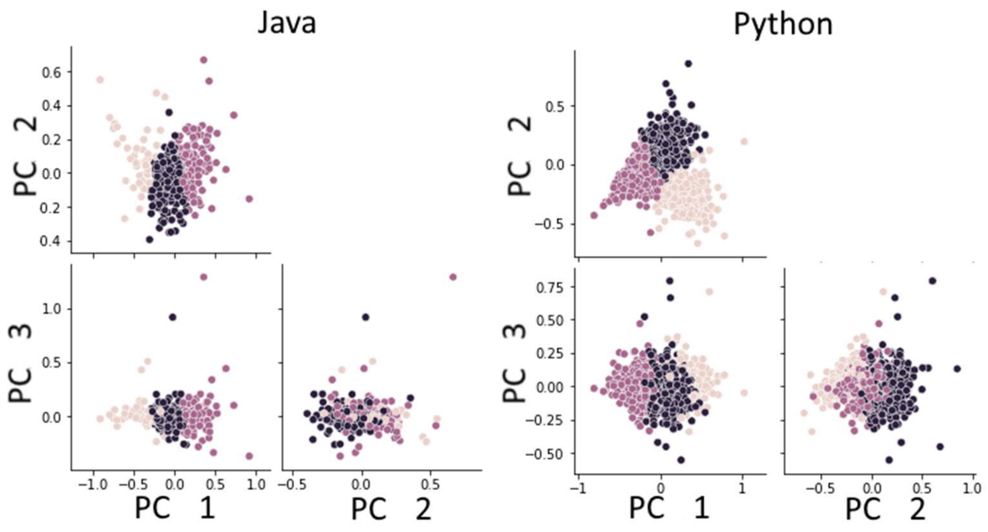


Figure 14 A scatterplot of the first three principal components for Java (left) and Python (right) datasets

For the Java dataset, the k-means clustering algorithm seems to have divided the data primarily along the first PC. It might thus be possible to achieve similar results to k-means clustering by merely plotting students on this axis. The second and third PCs were not considered by the k-means clustering algorithm.

Not surprisingly, the first PC is loaded very highly on one end with succeeding on the first attempt and on the other end, transitioning to the confused state. The second PC is more interesting, albeit had little effect on the clustering.

The second PC is easily interpreted as maintaining a successful state: one end of the axis is highly loaded with state transitions ending in success and the other end of the axis is highly loaded with state transitions ending in the confused state. However, the state successOnFirstAttempt is grouped with the confused state changes on this axis. It would seem the second PC differentiates the students who maintain working code (few syntax errors) from those students who normally fail to create working code, but also often finish exercises on the first attempt. This could conceivably be the case for students working outside the offered IDE, so their errors are not captured in the data. It may also be the struggling students who work in groups and only offer the group's answer or alternatively, students who cheat or relay heavily on copy-paste code from the internet. The third PC only explains around 10% of the variance in the data and is largely not involved in the clustering.

For the Python dataset, the k-means clustering algorithm seems to have divided the data along the first two of the most meaningful components (Figure 14). The PCs are quite similar to those from the Java data, except run time errors play a more prominent role. Similarly to Java, the first PC is highly loaded with S8, success on first attempt on one end, and confusion at the other. The second PC, however, discriminated students based on the type of error made. On one end was compile time errors, and run time errors on the other. As with the Java data, the third PC only explains around 10% of the variation in data and has virtually no effect on the clustering.

4.5 Non-negative matrix factorization

Similarly to PCA, a Non-negative Matrix Factorization (NMF) is a dimensionality reduction technique. Where the PCA attempted to redesign the search space so that maximal variance is retained with the least number of orthogonal axis (dimensions), the NMF attempts to factorize the transition matrix into smaller matrices which, when multiplied, produces the original transition matrix (Müller & Guido, 2016). Unlike the PCA, NMF requires the matrix to only have non-negative (i.e., zero or larger) values.

An NMF was performed separately on the Java and Python data, since the two languages were discovered to perform distinctively different transition matrices using a t-SNE dimensionality reduction (section 5.1). To determine the optimal number of factors, the reconstruction error for all values between 1 and 29 components was computed. A sharp drop in the reconstruction error was noticed at 5 factors. Thus, both datasets were factored into 5 factors. Figure 15 displays the found factors for both languages, with 10 most contributing components.

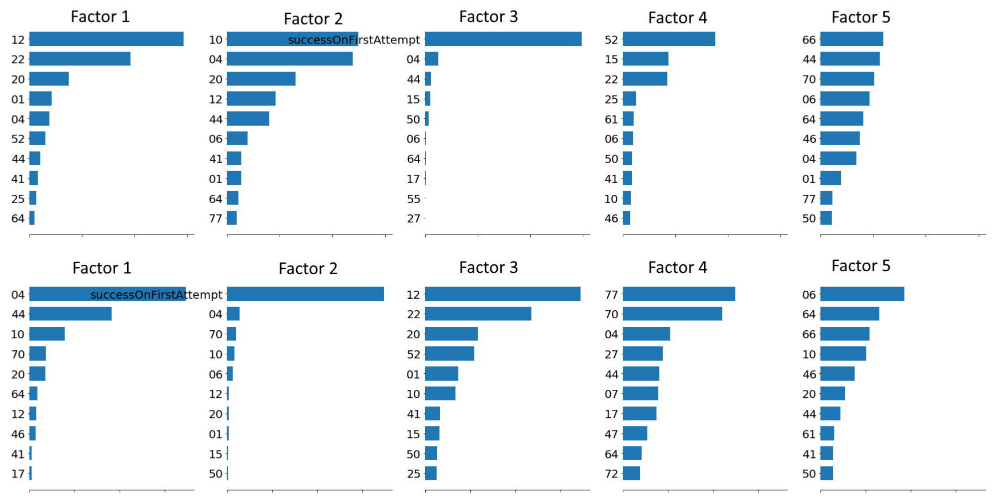


Figure 15 NMF factorization of Java (top) and Python (bottom) data to 5 factors. The X-axis is the loading for each variable. The Y-axis are state transitions; e.g. 04 is read as the transition from state 0 to 4.

The created factorizations could be interpreted as novice programmer archetypes. For instance, Java's factor 3 is highly loaded on the state successOnFirstAttempt and as an archetype is a strong programmer, who is likely to complete assignments on the first attempt. Python's factor 4, on the other hand, is the type of programmer who either receives a runtime error or no error, but does not manage to complete assignments in one attempt.

By creating linear combinations of the factors, all possible programmer types can be described. For instance, combining Java's factor 3 (programmer mostly completes assignments in one attempt) and factor 4 (programmer is mostly receiving errors) in a 50/50 ratio results in a programmer, who either completes assignments on the first attempt or not at all.

5 Model application

This section summarizes the applications of the model that have so far been published. The publications in question are publications III, IV and V.

Section 5.1 summarizes publication III, in which the model was applied to a programming course to show the model can be used to determine the previous programming experience purely from the code submission histories of students without pre-course surveys or quizzes.

Section 5.2 summarizes publication IV, in which the model was applied to a programming course to detect students who are engaged with the course material. The model was trained on the submission histories of students and clustered into three groups based on the model. The clusters were correlated with results from a weekly survey on engagement.

Section 5.3 summarizes publication V, in which the model was applied to Java and Python programming courses in order to determine whether the model can tell the difference between the two and also which language is easier.

5.1 Publication III: Automatically discovering prior programming experience

The purpose of publication III was to determine whether the model can be successfully applied in the classroom to determine the prior programming experience of the students on the course. Prior programming knowledge of students affects multiple aspects of the course, especially in mixed level groups often found in introductory programming courses (Smith IV, et al., 2019). If the exercises are geared towards the students with no prior experience, the more skilled students find the exercises too easy and get bored and disengage with the course, believing they already know all the course has to offer. On the contrary, offering exercises suited for those with previous programming experience risks losing the interest of the population with no experience.

The model was applied on two courses retroactively. The first course was given in fall 2017 and the programming language was Java. The second course was given in fall 2021 with the programming language being Python. Both courses contained programming exercises to which the students answered. The submissions were then used to form the Markov chain and transition matrix. In addition to programming exercises, students were given a pre-course survey asking their previous programming experience on both a Likert scale of 1-5 and as a list of programming languages used.

The model for each student was created from all submissions made to the programming exercises and were clustered using the K-Means clustering algorithm with a chosen K=3. Three clusters were chosen after an analysis of the data, but also as recommended in other literature (e.g., Moubayed et al., 2020).

The main research question of the study was whether or not the model has enough predictive power to differentiate between those with prior knowledge and those without. Thus, the null hypothesis is that there is no difference between the three generated clusters. However, statistical analysis showed a significant difference between the clusters, thus the null hypothesis was rejected. The difference between the clusters gradually vanished as the course advanced so that there were no significant differences between the groups during the last two weeks. The largest differences were found to be in the clusters for Python; for the first five weeks, the effect of prior programming knowledge in the strongest and weakest cluster was extremely significant (Tukey's HSD, $p<.01$). The last two weeks had no significant statistical difference in prior programming knowledge. The same pattern emerged for the Java course as well, the effect of prior programming knowledge for the first five weeks were deemed statistically different, albeit not as strongly as the Python course (Tukey's HSD, $p<0.05$) with no difference during the last two weeks.

The significance of this result is that the model can be used to discover students with prior programming experience solely from the actions on a programming course; in this case the submissions to coding exercises. No surveys or questionnaires are required. More importantly, this detection of prior programming skill can be done automatically with no teacher intervention or disruption to the student.

The major implication from this study is that the difference in student prior programming knowledge can be measured through the actions of the student during the course. It is perhaps unsurprising that previous programming experience affects students' behavior on a course, but the model presented in this thesis allows teachers to measure and quantify this difference in prior programming knowledge. Once this difference has been quantified, measures can be taken to begin diminishing the difference through pedagogical approaches, such as differentiated learning.

5.2 Publication IV: Automatically detecting engaged students

The purpose of this study was to apply the developed model to the brick-and-mortar classroom and determine whether it can be used to differentiate between students who are disengaged from the course material and those engaged with the course material. Engagement is also often called ‘commitment’ or ‘motivation’ (Schaufeli, 2013). Publication IV defines engagement as the willingness and enjoyment in working through the course’s programming exercises.

The null hypothesis was that disengaged and engaged students program in similar ways. Engagement was measured with weekly surveys asking an open-ended question: “How has programming felt thus far?” which were codified and mapped onto an integer scale of 1 to 3 in the analysis phase. Publication IV also measured student self-reported level of difficulty. The students were, in a weekly survey, asked how difficult they felt the week was, both on a Likert scale of 1 to 7 and as an open-ended question. The open-ended difficulty questions were processed similarly to the open-ended questions about engagement.

After the data coding phase, the transition matrices for all students were then computed for all students and an unsupervised ML algorithm, K-Means clustering, was used to create clusters from the created data points. The number of clusters was chosen to be K=3, according to previous findings from literature (Moubayed et al. 2020) and as explained in Section 4.1.

The results of the study were that the difference in programming behavior can be detected using the developed model. The formed three clusters had a statistically significant difference in means on both the self-reported difficulty (ANOVA, F(2,380)=32.715, *p*<.01), as well as self-reported levels of engagement with the course material (ANOVA, F(2,383)=3.697, *p*=.026). A post-hoc Tukey’s Honestly Significant Difference test indicated a significant statistical difference between all the groups in terms of difficulty (*p*<.01) for the Likert scale, but a slightly weaker significance in terms of the coded difficulties (*p*<.05). For engagement, a post-hoc Tukey’s HSD reported statistically significant differences only between the weakest and the strongest clusters (*p*<.026).

Publication IV also reports on the accuracy of the model and how the accuracy evolves over the course. Accuracy was computed for each week as the data points correctly identified in the cluster they ended in at the end of the course. The model accuracy started at 60% after week one, which means 60% of students were in the same cluster as they would be at the end of the course. The accuracy steadily increased, reaching an accuracy of 70% after week two and 90% at week five and 100% at week eight, the end of the course.

The implications of the findings of the study are that engagement breeds engagement, which is in line with the findings from Fwa & Marshall (2018). The findings of the study further imply that students find exercises more suitable to their skill level engaging. This is further evidence that the model is very applicable for incorporating differentiated learning into the classroom.

5.3 Publication V: Java or Python as first programming language

The purpose of publication V was to determine whether Python or Java is a more suitable first programming language in terms of how novice programmers behave when programming. Their behavior was measured as error rates and by training a K-Means clustering model on two datasets: Python programmers and Java programmers. The question of first programming language to teach to students is currently hotly debated (Jain et al., 2020; Khoirom et al., 2020). However, if we are to achieve the level of computer literacy discussed, for instance by Wing (2006), we should have the barrier for starting programming as low as possible. The choice of a (syntactically or conceptually) difficult language as the first language may have a deterring effect and many potentially proficient programmers may be turned away from programming.

The study was performed by first collecting two datasets: one of Python code submissions and other of Java code submissions. The two datasets show immediate differences in terms of error rates: nearly 60% of submissions in the Java group were had a compilation or runtime error, whereas only slightly over 50% had errors in the Python group. This difference was statistically significant ($\chi 2(1219454) = 1070.2719, p<.001$),

In addition, the two groups were used to train a K-Means clustering model, with K=3. The number of clusters was chosen based on existing literature. The distribution of students on the Java course was 159 students in the weakest cluster, 151 students in the average cluster, and 43 students in the best performing cluster.

Then, in order to determine what would have happened, had the Python programmers been on the Java course, the models were used to cluster the other group. That is to say the Java-trained model was used to cluster the Python group. The outcome of this was that had the Python students been on the Java course, only 9 students had been in the lowest performing group and the remaining students would have been split roughly evenly among the two other clusters.

The conclusions drawn in the paper are that we can say that students in Python are less likely to create erroneous code. The Python programmer group was also considered stronger programmers by the model in general. Thus, when given a free choice between Python and Java as the first programming language, Python is recommended. However, other factors, such as instructor capability, restrictions by the educational institute must also be taken into account when choosing the first language to be taught to the students.

The main theoretical contributions of publication V lie in the findings that the model found differences between Python and Java students. Publication V also provides practical contributions in the form of lists of typical errors students generate in both languages. The benefit of these error lists is two-fold: firstly, they give indication into what instructors should pay attention to when instructing debugging strategies and course content. Secondly, they can be used to cross-reference existing studies which provide such lists in order to discover what change, if any has happened in the errors novice programmer produce, in their first programming class.

6 Discussion

This chapter offers discussion about the model and its implications to the current scientific body of knowledge. In Section 5.1 I relate the model to the related literature and discuss the application of the model. In Section 5.2 I discuss how the model answers the research questions posed in the introduction. In Section 5.3 I describe the theoretical and practical contributions offered by the model in this thesis. Lastly, in Section 5.4 I dissect the limitations directly affecting the model and the attached studies.

6.1 Discussing the model

Programming is not one skill, but a composition of many. Computational thinking is not programming in itself, but a major component of programming. Computational thinking is solving problems efficiently, using abstraction and problem decomposition; computational thinking is transforming problems into ones that can be computed (Wing, 2006). On a practical level, programming skills are those of planning, developing, testing, and debugging (du Boulay, 1989). Programming is also a highly hierarchical skill, where concepts are built on top of other concepts (Venables, 2019). If a student does not know a concept, learning concepts dependent on the unlearned one becomes an insurmountable task.

It seems unreasonable to expect one single model or metric to be able to represent such a wide spectrum of behavior and knowledge. That is why the model presented in this thesis focuses primarily on the behavior of novice students; how they program and what happens during their programming sections. This model does not attempt to capture student knowledge, their misconceptions about specific programming concepts or rank students according their skill (even though it seems strong programmers do behave similarly).

The reasoning behind choosing to model student behavior instead of specific skills is mainly that of practicalities; the submission history of a student is an objective truth of how the student has performed. Measuring mastery of a single concept in programming is very complicated, as everything is interconnected. Measuring the knowledge of simple for-loops requires a test that controls for variables such as programming variables, assignment and usage, control flow, conditionals, and sequential execution of statements. Also the application of the loop-structure needs consideration. For instance have all repeated code sections been turned into a loop structure. . The submission history of the student has no such complications.

The developed model is capable of grouping similarly behaving students together. Thus, if one student in such a group can be identified as having difficulties, the rest of the group can also be assumed to. Publication IV proves that the groupings do indeed have statistically different perceptions of programming difficulty. This enables educators to target those students who feel programming is difficult with extra assistance via differentiated instruction. This extra assistance might be of extreme importance especially in the beginning of the course, where the 'computer shock' (Kreitzberg & Swanson, 1974) of programming is most prevalent and potentially leads the student to drop out of the course.

Differentiating instruction this way could be of great benefit to CS1 courses. Currently, drop-out rates are still high, even after decades of work in attempting to alleviate the problem (McCracken et al., 2001). Learning outcomes on CS1 courses fall below teacher expectation; albeit this may just be due to excessive expectations from the teachers' part (Luxton-Reilly, 2016). Both the aforementioned problems might be alleviated by providing students with exercises tailored to their current skill level. By giving easier exercises to the struggling students they are given a chance to catch up, while simultaneously giving the strong students more difficult exercises to help them become even stronger programmers.

Publication IV studied whether the experience of difficulty varies according to the generated clusters. The implications of Publication IV are that engagement breeds engagement, which is in line with the findings from Fwa & Marshall (2018). The findings of publication IV further imply that students find exercises more suitable to their skill level engaging. By providing engaging exercises and learning experiences to students of all skill levels would likely lead to increased retention and higher learning outcomes.

The findings from publication IV, where engagement breeds engagement, combined with the results from publication III create an interesting implication. Since the cluster with the highest prior programming experience also experienced highest engagement and indicated programming to not be difficult, we can tentatively argue, that the more students program (on their skill level), the more engaging and interesting it becomes. It may also be that this result shows survivorship bias; those who felt programming is not for them quit before gaining enough prior programming experience to be included in these studies.

The model itself builds on top of the works of Jadud (2006), Wason, Li and Godwin (2013), Carter, Hundhausen and Adesope (2015) and Becker (2016). It combines to power of the state machine from NPSM with the—simple to compute—

scoring rubric of the EQ and the Watwin score. Measuring programmer behavior in terms of repeated behaviors came from RED. This, combined with the mathematical theory from Markov models proved to be able to cluster students into distinct groups.

While not the first study to apply Markov models to analyzing novice students (see for instance, Fwa and Marshall, 2018), it is the first to do so with a simple protocol for data collection, to my knowledge. The implications of being able to model student behavior as a Markov chain still leave a lot to be researched, but opens a way to combine proven mathematics with educational research into novice programmers.

6.2 Addressing the research questions

After the analysis presented in this study alongside with those presented in Studies I to V, the research questions presented in the beginning are answered. The first research question is mainly focused on what it takes to create a model for clustering students. The second question is mostly concerned with how to define the efficiency of the model as well as how well it performs in light of this definition. The last research question aims to discover the applications of the model thus far.

6.2.1 RQ1: How can a model be built for automatically clustering novice programming students?

As discussed in the previous section, no single model is likely to capture the full variation and scope of programmer skill. Creating such a model would end up to be a gargantuan task, as programming is composed of a multitude of aspects. Thus, a likely future is one where multiple models work in tandem to determine how well novice programmers are learning to program.

Publication I is crucial in demonstrating that novices and experts behave differently and that qualitative measures of code do give us great insight into the novice programmer mindset. However, the amount of work required to do that becomes unsustainable for a large number of submissions. Thus, the other observation that quantitative metrics are able to divide strong programmers (the experts) from the novice programmers (the students).

Publication II demonstrates how e-learning was applied to an engineering course to collect data for analysis. It also demonstrates the types of assignments that are suited especially well for automatic assessment and thus, collecting data for analysis.

The quantitative approach, however, presents other challenges. Firstly, and most importantly the question of data collection must be addressed. For any quantitative study, one needs data; the more the better, usually. This is why the data collection for this model was done with ViLLE, a LMS developed in-house in the University of Turku. ViLLE enables automatic grading of programming tasks, as well as giving the student immediate feedback on their performance. After a student submits their programming code for the given task, it is first graded and the user is given feedback, then it is stored in a database. This database will then, over time, contain hundreds of thousands and even millions of such code submissions, all merely waiting to be analyzed.

All in all, it is firmly recommended that data collection be brought as an integral part of the course via continuous assessment, as done in publication II. This allows data to be collected from between the end points of student starting a course and the end point of the formal end-course exam. While sufficient to (sometimes) measure learning on a course, it is not enough to assess who needs help during the course and at which point in time.

Masses of data obviously benefit the researcher, but what is the benefit to the student, or teacher of the course? Publication II demonstrates the automatic assessment and feedback capabilities of ViLLE, where the student receives almost instant feedback on their progress through the tasks. Additionally, the variety in exercise types offered by ViLLE removes the feeling of repetition from students, as they are continuously offered different types of activities. The above features are of benefit to the teacher as well: automatic grading of small programming exercises frees up time to both guide the students as well as improve the quality of the course in terms of, for instance, lecture content, handouts or exercises.

Researchers collecting data should not forget the rights of the students; the student may not wish to have his or her collected data used in research. All students must thus be given the option to opt-out of the research, as the data is primarily collected and stored for educational purposes.

Once the question of data collection is solved, the next question that arises is that of analysis. More specifically, how to analyze the masses of collected data. Three major choices are available: purely statistical, ML or a combination of the two. Purely statistical methods are applicable to topics where answers are binary; the student is either right or wrong, such as K-12 mathematics (Lokkila et al., 2015). This model, described in Section 3.3, was created with ML. More specifically applying Markov chains to the submission history generated by students.

Good results were obtained with the simplest Markov model, which begs the question: could even better results be obtained with Hidden Markov models? The answer is probably yes, as the premise of HMMs is that we have observable states (the transition matrix from the submission history) and unobservable states that control which observable state is emitted. These unobservable states could be the likes of (from Perkins et al., 1986): moving, tinkering or stopping. Alternatively, student emotions could be revealed: confused, engaged, angry. This remains as future work.

More complicated models may, however, need more involved data collection processes. While ViLLE (currently) can only collect data at the coarse level of submission attempts, other researchers have applied a variety of finer grain data collection processes, such as storing student edits performed on the code (Heinonen et al., 2014) or even keypresses (Toll et al., 2016). This finer level of detail in the data may prove fruitful when combined with a HMM. However, a meaningful clustering of students can be achieved using only a state machine, like one described in section 3.3. As students transition from state to state within the state machine, a transition matrix is formed which uniquely identifies each student.

6.2.2 RQ2: How effective is the created model?

Publications III, IV and V all answer, in part, the question of how effective the model is. However, before the results are discussed, the term 'effectiveness' should be defined. The Cambridge dictionary (2022) defines it as "the ability to be successful and produce the intended results". This definition however, is not very helpful, as we still need definitions for success and the intended results of the model.

The model is intended to be able to group students of similar programming behavior. Thus, if a student produces a certain kind of transition matrix, other students with similar matrices presumably behave similarly. It is worth noting, that the term behavior does not equal skill, necessarily.

Success, in terms of clustering students, is when the created clusters do hold the above property of being similar in one or more ways. If the model is successful, then the created clusters are distinct enough so that when given a new data point (i.e., a student's transition matrix), placing the data point in the model's latent variable space and applying the K-nearest neighbors algorithm will tell us how the student behaves when programming. In other words, it should repeatedly create similar transition matrices for similarly behaving students.

Section 4.2 of this study shows that the model is indeed capable of separating the Java programmers from the Python programmers. First the model variable space was clustered into two clusters, then a t-SNE dimensionality reduction was applied. The result was a clear distinction for the two groups, as seen in Figure 12. What, then, is the separating feature? The most obvious explanation is how the two languages are built: Java is a (mostly) compiled language, whereas Python is a (mostly) interpreted language. This distinction creates a radically different error profile for students programming in these languages: a higher proportion of compile errors for Java programmers, but a higher proportion of runtime errors for the Python programmers. The results of publication V also indicate that novice programmers in Python create less errors altogether.

In publication III, the model was employed to detect and group those students with previous programming experience. The students were given a pre-course survey in which they were asked for their previous programming experience on a Likert scale as well as a list of programming languages. Each week during the course, the transition matrices were created for each student. After the course had ended, the data was tabulated and a significant difference between prior programming knowledge between clusters was discovered for the majority of the course. Interestingly, the first five weeks of a 7 week-course were clustered such that they contained statistically significant differences in prior programming knowledge. During the two last weeks, no statistically significant differences were found. This effect is presumably due to the course material becoming more and more difficult, thus decreasing the significance of prior programming experience; that is to say, after the fifth week, all presented material was new to the majority of students.

These findings have practical implications to programming educators. The difficult weeks' topics were existing libraries and revision for Java, and Files, exceptions and libraries for Python. Notably the theme 'libraries' appears on both these lists. These topics, based on the results from publication III, are such that previous programming experience does not affect them.

The intent in publication IV was to use the model to group the students into three groups, each with differences in either experience difficulty or engagement to the course material. That is to say, our null hypothesis is that the model is unable to create clusters with meaningful differences in these attributes. The students were given a weekly feedback survey asking them two open-ended questions: 'how difficult has programming been thus far' and 'How has programming felt thus far". Additionally, an estimate of that week's difficulty was asked on a Likert scale.

The results from publication IV give grounds to discard the null hypothesis. This is to say that the model successfully created clusters which had a statistically different mean in the self-reported feeling of both difficulty in programming as well as engagement. These findings raise the question: Why is it, that the same group of students who consider programming difficult, also is the least engaged in the material? One explanation could be the reverse of the finding from Fwa and Marshall (2018); disengagement breeds disengagement. It is likely these students have not learned some fundamental concepts in programming and as a result struggle to understand the concepts later on in the course. This result would be in line with those of Venables et al. (2009), who state that programming skills form a hierarchy, where one must learn the basics before being capable of learning the more difficult concepts.

In summary, if effectiveness is measured as the successful capability to group similarly behaving students together, then the model described in this thesis fulfills this requirement, as shown in publications II, IV and V. Successful clustering being that they do indeed share a similar trait, and then the model certainly can do this. In the analysis in Section 4.2, it grouped Python programmers together and Java programmers together. In publication III it successfully created groups of students with similar previous programming experience. In publication IV it successfully created clusters of students who considered the exercises to be of similar difficulty.

6.2.3 RQ3: What can the model be applied to?

The previous sections discuss how the model was created and that it is successful in creating clusters of similarly behaving students. But without practical applications, the model would not be very useful. This section discusses how the model can be applied to real world situations and the possible benefits of such applications.

Publication III demonstrates the application of the model in detecting previous programming knowledge of students on introductory programming courses. It is important for the teacher of an introductory programming course to know how much, if any, previous programming knowledge the students have. This knowledge allows the teacher to adapt course material and teaching practices to better suit the needs of the students. While other methods of detecting previous programming knowledge exist (Leinonen et al., 2016, Strong et al., 2017; Smith IV et al., 2019), they all require self-reflection on the part of the student. These methods include interviews and surveys. These are problematic, because the data received with these methods is inherently objective and often hard to quantify. Additionally, interviews take time to conduct (which the teacher may not have) and not everyone answers surveys. Publication III demonstrates a method to gain this insight automatically from assignments normally done on the course, in an objective manner.

Publication IV demonstrates the application of the model in a classroom situation to detect those students who the teacher should target for extra assistance. The model is able to determine, with acceptable accuracy, whether a student feels the exercises are too difficult and also whether the student begins to feel unmotivated. While the clustering generated by the model showed that those who feel the exercises are difficult also felt more disengaged, no causation is confirmed. That is to say, it is unknown whether the difficulty of exercises causes disengagement, or the disengagement feeds into students' experiences of difficult assignments. This information revealed by the model is crucial in helping the teacher direct resources to those students who truly need it. In the best case, well-timed assistance can prevent students from dropping out of the course and continuing on to becoming professional programmers.

Publication V demonstrates the applicability of advancing the theory of programming languages by analyzing not only the student but the programming language itself. Publication V used this model to compare two programming languages, Java and Python, with the conclusion that Python is easier for students to work in. This result can be extended to follow-up studies where similar comparisons are used for different languages in order to, for example, create lists of easy languages and difficult languages. This naturally leads to analysis of what aspects of the languages affect the difficulty of the given language. In publication V the distinctive feature between these two languages is that one is a (mostly) compiled language, whereas the other is a (mostly) interpreted language. This model was able to discriminate between these two languages. Exactly what features were used in creating this distinction requires further work, but presumably runtime errors were more common in Python, whereas compile time errors were more prevalent in Java.

When combining these results from all studies, the most obvious benefit is in enabling differentiated learning for students. It is possible, using the model, to determine which student experiences difficulties and offer additional assistance. For instance, if a student has been advancing through the course with no problems, but on the week whose topic is ‘external libraries” the student begins to struggle, they should be offered additional material regarding specifically on how external libraries work. Additional instruction may also be offered to those who begin to show signs of struggling and disengagement with the hopes of enabling the students to move back into a zone of effective learning.

Publications III, IV and V each determined features of the clusters generated by the model. Logically, if the model is able to find these distinctions, it should be able to find other distinctions as well. However, determining which other properties can be determined using this model is left as future work.

6.3 Theoretical and practical contributions

This work offers both theoretical contributions to the researchers studying novice programmers and their behavior, and practical contributions to the educators on how to further improve programming education in their class.

The theoretical contribution of this work is that novice student behavior can be modelled as a Markov chain, using only the student submission history to coding exercises, which to my knowledge is a novel approach. Leaning to mathematical theories offers great benefits. Namely the mathematical theory has often been extensively researched. Markov models have been researched for nearly 100 years. Combining this cumulated body of knowledge from mathematics with the development of the model presented in this thesis offers numerous interesting research paths to explore further.

The practical contribution comes from the learning analytics perspective and the application of this model to the classroom. In publication III the capability of the model to differentiate between students with prior programming knowledge and those without is shown. Publication IV demonstrates the capability to differentiate students who feel the exercises are difficult and those who feel they are easy.

The results from Publications III and IV combined provide educators with opportunities to provide differentiated instruction in their class. Thus, students who struggle with programming can be offered additional support and those who are doing well, may be offered more challenging material to further develop their skills.

For educators, the findings from publication III provide insight into what topics are challenging to students and the typical errors students generate during introductory programming courses. Notably, the difficult weeks' topics were existing libraries and revision for Java, and Files, exceptions and libraries for Python. Interestingly the theme 'libraries' appears on both these lists. These topics, based on the results from publication III, are such that previous programming experience does not affect them. Indeed, students across the board consider them difficult.

6.4 Limitations of the study

All studies have limitations. Theofanidis and Fountouki (2018) define a limitation as something external, outside the researchers influence that may influence the results. They give, as examples, the sample population, the used data analysis methodology and the research tool used. Some limitations are internal to the study and stem from the boundaries of the chosen methods, or decisions made by the researcher. For example, the decision to remove outliers may have an effect on the results.

There are also external limitations for such studies. The studies in this thesis have a very limited sample diversity. Data was collected only from Finnish students from educational institutes in Turku. Different educational institutions from different cities in Finland or from other countries may have produced different results. Further studies with more diverse data sets should be collected and analyzed. One possible such dataset is the blackbox dataset (see Brown, et al., 2014).

All data for this thesis was collected from the LMS ViLLE. Only four data points (although more can be then inferred from them) were available. Having more fine-grained data, such as keypresses, could provide interesting affordances for identifying struggling students, when combined with this model. Using a LMS may introduce noise to the data: some students may choose to program outside the LMS and submit only near-working code, so that their errors are not captured in the data set.

Internal limitations exist as well. The data in the studies comprising this thesis was only quantitatively analyzed. It is possible some findings were left undiscovered because the code submissions were only automatically analyzed. However, due to the sheer number of code submissions, qualitatively analyzing every submission is not a reality. Qualitative analysis was performed in small scale in this thesis to validate the clusters. Manual qualitative analysis is also required for labeling the datasets, which in turn allows the application of supervised machine learning algorithms.

Internal limitations are also found in the chosen ML methods. Because the studies comprising this thesis use only the k-means clustering algorithm, it remains unknown whether other unsupervised learning algorithms perform better. For instance, the expectation maximization algorithm is known to creates different clusters than k-means. Further studies are required which compare different clustering methods.

7 Conclusions And Future Work

This thesis presented a model for modeling student behavior using learning analytics. The model consists of a state machine, which represents the different states the student can be at any given point in time during a programming exercise. The transition probabilities of the state machine are unknown at the beginning and represent the current student. The model collects a random walk through a Markov process from the student in the form of a submission history to coding exercises on a programming course. The weights for the state machine are then computed from this random walk. As more data is collected on the student (i.e., the student continues to do programming exercises), the weights become more and more accurate. As such, the presented model is completely data driven.

The presented model fits snugly into the category of predictive LA; the created model can be used to predict the future outcomes of new students on introductory programming courses. The model also offers tools to understand the different types of novice programmers through dimensionality reduction techniques such as t-SNE and NMF.

This thesis also presents an additional metric, computed by reducing the transition matrix to a single value. It is computed directly from the transition matrix and is a single-dimensional real value in the range [-1,2]. Typical values of the metric are in the range [-1,1]. The less errors a programmer makes, the higher the value. This value can be used to approximate student skill; the higher the value the stronger the programmer, whereas low values indicate a weak programmer.

The model described in Section 3.3, based on Markov Chains, demonstrates constituency in the obtained results. The applicability of the model to detect programming students with prior programming knowledge is demonstrated in Section 4.5. The model's capability of detecting students who are disengaged or experience difficulties is demonstrated in Section 4.6. The model can also be used to study programming languages. Section 4.6 demonstrates the model's capability to determine whether Python or Java is the easier language for novice programmers.

This thesis also analyzed the described model and found that both Principal Component Analysis (PCA) and Non-negative Matrix Factorization (NMF) were able to reduce the dimensionality of the latent space of the model. The PCA analysis was performed in Section 4.3. The NMF analysis was performed in Section 4.4.

The major theoretical contribution of this thesis is that novice programmers' programming behavior can be represented as a student's random walk through a Markov process. This creates a multitude of future research opportunities. The Markov model created by this random walk can then be used to discern specific properties of the students, such as engagement, prior programming skill and experienced difficulty.

The model described in this thesis demonstrates a variety of favorable properties, however, much is still unknown. While the clusters do show statistically significant differences, a qualitative analysis is needed. A qualitative analysis would further give insight into how the clusters differ and what difficulties the students face in practice.

While publications III, IV and V demonstrate the capability of discerning some properties from the created clusters, it is likely the clusters share even more properties. These properties could be used to create an even more accurate profile of the novice programmer and their programming behavior.

This model is planned to be used as a basis for an intelligent tutoring system, which would automatically detect not only the struggling students, but exactly *what* the student is struggling with. This would enable the system to offer targeted assistance with precision based on the actual student needs.

Other future use cases for this model is comparing different programming courses. By first computing the model for each course individually, the courses can be compared is a similar way as was done in publication V for programming languages. Once the models have been computed, the models can be swapped, so that the students on course 1 are evaluated using the model from course 2. If the courses are similar, no statistical changes should occur in the distribution of students in each cluster. However, if the courses differ significantly, then the clustering of students will change as well. Having a data-driven comparison point, allows for more objective analysis of differences between courses, for example for intervention studies, where the model would first be trained before the intervention (or for the control group), then another model trained after the intervention (or for the intervention group).

This model leans heavily into the mathematical theory of Markov models. However, only the simplest case of a Markov chain is studied. The model currently studies only bigrams of the random walk. It is possible that higher order Markov models that take into account arbitrarily large n-grams, would have a better accuracy or predictive capability.

Hidden Markov models could also be applied. Hidden Markov models are those which contain unobservable states, and these models could conceivably be used to model the student's mood or other hidden attributes.

The presented model offers many interesting research paths to be taken in the future by combining the mathematical theory of Markov models, ML algorithms and educational research

List of References

Achmad, A. L. H., & Ruchjana, B. N. (2021). Stationary distribution Markov chain for Covid-19 pandemic. In Journal of Physics: Conference Series (Vol. 1722, No. 1, p. 012084). IOP Publishing.

Ahadi, A., Lister, R., Haapala, H., & Vihavainen, A. (2015, August). Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 121-130).

Al-Shaar, W., Adjizian Gérard, J., Nehme, N., Lakiss, H., & Buccianti Barakat, L. (2021). Application of modified cellular automata Markov chain model: forecasting land use pattern in Lebanon. *Modeling Earth Systems and Environment*, 7(2), 1321-1335.

Almutiri, T., & Nadeem, F. (2022). Markov Models Applications in Natural Language Processing: A Survey.

Alwahaby, H., Cukurova, M., Papamitsiou, Z., & Giannakos, M. (2021). The evidence of impact and ethical considerations of Multimodal Learning Analytics: A Systematic Literature Review.

Balzuweit, E., & Spacco, J. (2013, July). SnapViz: visualizing programming assignment snapshots. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 350-350).

Becker, B. A. (2016, July). A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 296-301).

Bellman, R. E. (1957). Dynamic programming. Princeton University Press. p. ix. ISBN 978-0-691-07951-6.

Brooks, C., & Thompson, C. (2017). Predictive modelling in teaching and learning. *Handbook of learning analytics*, 61-68.

Brown, N. C. C., Kölling, M., McCall, D., & Utting, I. (2014, March). Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 223-228).

Cambridge Dictionary. (2022, July 2). "The definition of effectivencess". https://dictionary.cambridge.org/dictionary/english/effectiveness

Campesato, O. (2020). Angular and Machine Learning Pocket Primer. Mercury Learning and Information, LLC.

Carter, A. S., Hundhausen, C. D., & Adesope, O. (2015, August). The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 141-150).

Cerratto Pargman, T., & McGrath, C. (2021). Mapping the ethics of learning analytics in higher education: A systematic literature review of empirical research. *Journal of Learning Analytics*, *8*(2), 123-139.

Chicco, D. (2017). Ten quick tips for machine learning in computational biology. BioData mining, 10(1), 1-17.

Ching, W. K., & Ng, M. K. (2006). Markov chains. Models, algorithms and applications.

Clow, D. (2013). An overview of learning analytics. *Teaching in Higher Education*, *18*(6), 683-695.

Combs, A. T. (2016). Python Machine Learning Blueprints: Intuitive data projects you can relate to. Packt Publishing Ltd.

Conole, G., Gašević, D., Long, P., & Siemens, G. (2011, December). Message from the LAK 2011 general & program chairs. In *International Learning Analytics & Knowledge Conference 2011*. Association for Computing Machinery (ACM).

Cook, D. (2016). Practical machine learning with H2O: powerful, scalable techniques for deep learning and AI. " O'Reilly Media, Inc.".

Costa, E. B., Fonseca, B., Santana, M. A., de Araújo, F. F., & Rego, J. (2017). Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. *Computers in human behavior*, 73, 247-256.

Dalton, D. W., & Goodrum, D. A. (1991). The effects of computer programming on problem-solving skills and attitudes. *Journal of Educational Computing Research*, 7(4), 483-506.

de Raadt, M. (2007). A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education*, *17*(3), 201-213.

Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?. *Computers & Education*, 58(1), 240-249.

Duda, R. O., Hart, P. E., & Stork, D. G. (1973). Pattern classification and scene analysis (Vol. 3, pp. 731-739). New York: Wiley.

Fwa, H. L., & Marshall, L. (2018). Modeling engagement of programming students using unsupervised machine learning technique. *GSTF Journal on Computing*, 6(1), 1.

Gil Fonseca, N., Macedo, L., & Mendes, A. J. (2018, February). Supporting differentiated instruction in programming courses through permanent progress monitoring. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 209-214).

Gerhardt, L. (1974). Pattern recognition and machine learning. *IEEE Transactions on Automatic Control*, 19(4), 461-462.

Google (2018, January 17) Cloud AutoML: Making AI accessible to every business. Google blogs. https://blog.google/products/google-cloud/cloud-automl-making-ai-accessible-every-business/

Heinonen, K., Hirvikoski, K., Luukkainen, M., & Vihavainen, A. (2014, March). Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 229-234).

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1), 25-40.

Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ (Doctoral dissertation, University of Kent).

Jain, S. B., Sonar, S. G., Jain, S. S., Daga, P., & Jain, R. S. (2020). Review on Comparison of different programming language by observing it's advantages and disadvantages. *Research Journal of Engineering and Technology*, 11(3), 133-137.

Kaila, E., Kurvinen, E., Lokkila, E., & Laakso, M. J. (2016). Redesigning an object-oriented programming course. *ACM Transactions on Computing Education (TOCE)*, 16(4), 1-21.

Kant, E. (1980). Review of" Principles of Artificial Intelligence by Nils J. Nilsson", Tioga Publishing Co. ACM SIGART Bulletin, (72), 4-5.

Kemeny, J. G. (1983). The case for computer literacy. Daedalus, 211-230.

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1), 1-43.

Khoirom, S., Sonia, M., Laikhuram, B., Laishram, J., & Singh, T. D. (2020). Comparative Analysis of Python and Java for Beginners. *International Research Journal of Engineering and Technology*, 7(8), 4384-4407.

Kirk, M. (2017). Thoughtful machine learning with Python: a test-driven approach. " O'Reilly Media, Inc.".

Knuth, D. E. (1985). Algorithmic thinking and mathematical thinking. *The American Mathematical Monthly*, 92(3), 170-181.

Kreitzberg, C. B., & Swanson, L. (1974, May). A cognitive model for structuring an introductory programming curriculum. In *Proceedings of the May 6-10, 1974, national computer conference and exposition* (pp. 307-311).

Laakso, M. J. (2010). Promoting programming learning. engagement, automatic assessment with immediate feedback in visualizations.

Laakso, M. J., Kaila, E., & Rajala, T. (2018). ViLLE–collaborative education tool: Designing and utilizing an exercise-based learning environment. *Education and Information Technologies*, 23(4), 1655-1676.

Lang, C., Siemens, G., Wise, A., Gasevic, D. & Merceron, A. (Eds.). (2022). Handbook of learning analytics (2nd ed.). New York: SOLAR, Society for Learning Analytics and Research.

Leinonen, J., Longi, K., Klami, A., & Vihavainen, A. (2016, February). Automatic inference of programming performance and experience from typing patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 132-137).

Lokkila, E., Kurvinen, E., Kaila, E., & Laakso, M. J. (2015). Automatic recognition of student misconceptions in primary school mathematics. In *Proceedings of the 7th International Conference on Education and New Learning Technologies* (pp. 2267-2273).

Lokkila, E., Kaila, E., Karavirta, V., Salakoski, T., & Laakso, M. (2016). Redesigning Introductory Computer Science Courses to Use Tutorial-Based Learning. *EDULEARN16 Proceedings*, 8415-8420.

Lokkila, E., Kurvinen, E., Larsson, P., & Laakso, M. J. (2017). Redesigning an introductory database course to utilize tutorial-based learning. In *Conference paper presented at EDULEARN17.*

Luxton-Reilly, A. (2016, July). Learning to program is easy. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 284-289).

Ma, S., Dang, D., Wang, W., Wang, Y., & Liu, L. (2021). Concentration optimization of combinatorial drugs using Markov chain-based models. *BMC bioinformatics*, 22(1), 1-19.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., ... & Wilusz, T. (2001). 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* (pp. 125-180).

Mok, H. N. (2011). Student usage patterns and perceptions for differentiated lab exercises in an undergraduate programming course. *IEEE transactions on Education*, 55(2), 213-217.

Morgan, P. (2018). Machine Learning is Changing the Rules: Ways Business Can Utilize AI to Innovate. O'Reilly Media.

Moubayed, A., Injadat, M., Shami, A., & Lutfiyya, H. (2020). Student engagement level in an e-learning environment: Clustering using k-means. *American Journal of Distance Education*, 34(2), 137-156.

Müller, A. C., & Guido, S. (2016). Introduction to machine learning with Python: a guide for data scientists. " O'Reilly Media, Inc.".

Murphy, C., Kaiser, G., Loveland, K., & Hasan, S. (2009, March). Retina: helping students and instructors based on observed programming activities. In *Proceedings of the 40th ACM technical symposium on Computer Science Education* (pp. 178-182).

Ongsulee, P. (2017, November). Artificial intelligence, machine learning and deep learning. In *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE)* (pp. 1-6). IEEE.

Parker, S. C. (1924). Adapting instruction to differences in capacity. *The Elementary School Journal*, 25(1), 20-30.

Parsons, D., & Haden, P. (2006, January). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education*-Volume 52 (pp. 157-163).

Pereira, F. D., Oliveira, E., Cristea, A., Fernandes, D., Silva, L., Aguiar, G., ... & Alshehri, M. (2019, June). Early dropout prediction for programming courses supported by online judges. In *International conference on artificial intelligence in education* (pp. 67-72). Springer, Cham.

D. N. Perkins, C. Hancock, R. Hobbs, F. Martin and R. Simmons, "Conditions of learning in novice programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37-55, 1986.

Petersen, A., Spacco, J., & Vihavainen, A. (2015, November). An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (pp. 77-86).

Provost, F., & Kohavi, R. (1998). Guest editors' introduction: On applied research in machine learning. *Machine learning*, 30(2), 127-132.

Raschka, S., & Mirjalili, V. (2019). Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2. Packt Publishing Ltd.

Richards, B., & Hunt, A. (2018, November). Investigating the applicability of the normalized programming state model to BlueJ programmers. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (pp. 1-10).

Rodrigo, M. M. T., Tabanao, E., Lahoz, M. B. E., & Jadud, M. C. (2009). Analyzing online protocols to characterize novice java programmers. *Philippine Journal of Science*, 138(2), 177-190

Sasidharan, S. K., & Thomas, C. (2021). ProDroid—An Android malware detection framework based on profile hidden Markov model. *Pervasive and Mobile Computing*, 72, 101336.

Sazonov, I., Grebennikov, D., Meyerhans, A., & Bocharov, G. (2021). Markov chain-based stochastic modelling of HIV-1 life cycle in a CD4 T cell. *Mathematics*, 9(17), 2025.

Schaufeli, W. B. (2013). What is engagement?. In *Employee engagement in theory and practice* (pp. 29-49). Routledge.

Scouller, K. (1998). The influence of assessment method on students' learning approaches: Multiple choice question examination versus assignment essay. *Higher education*, 35(4), 453-472.

Sebestyen, G. (1966). Review of'Learning Machines'(Nilsson, Nils J.; 1965). *IEEE Transactions on Information Theory*, 12(3), 407-407.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*(3), 219-238.

Sindhu, V., Nivedha, S., & Prakash, M. (2020). An Empirical Science Research On Bioinformatics In Machine Learning. *Journal of Mechanics of Continua and Mathematical Sciences*.

Smith IV, D. H., Hao, Q., Jagodzinski, F., Liu, Y., & Gupta, V. (2019, May). Quantifying the effects of prior knowledge in entry-level programming courses. In *Proceedings of the ACM Conference on Global Computing Education* (pp. 30-36).

Spohrer, J. C., & Soloway, E. (1989, January). Simulating Student Programmers. In *IJCAI* (Vol. 89, pp. 543-549).

Steinkraus, D., Buck, I., & Simard, P. Y. (2005, August). Using GPUs for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition* (ICDAR'05) (pp. 1115-1120). IEEE.

Strauss, A., & Corbin, J. M. (1997). Grounded theory in practice. Sage.

Strong, G., Higgins, C., Bresnihan, N., & Millwood, R. (2017, July). A survey of the prior programming experience of undergraduate computing and engineering students in ireland. In *IFIP World Conference on Computers in Education* (pp. 473-483). Springer, Cham.

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011, August). Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research* (pp. 85-92).

Taylor, B. K. (2015). Content, process, and product: Modeling differentiated instruction. Kappa Delta Pi Record, 51(1), 13-17.

Toll, D., Olsson, T., Ericsson, M., & Wingkvist, A. (2016). Fine-grained recording of student programming sessions to improve teaching and time estimations. In *International journal of engineering education* (Vol. 32, No. 3, pp. 1069-1077). Tempus Publications.

Tomlinson, C. A. (2000). Reconcilable differences: Standards-based teaching and differentiation. *Educational leadership*, 58(1), 6-13.

Tomlinson, C. A. (2001). How to differentiate instruction in mixed-ability differentiated instructions. classrooms. *Carol Ann Tomlinson. Alexandria, VA: Association for Supervision and Curriculum Development.*–2001.–124 p.

Truquet, L. (2019). Local stationarity and time-inhomogeneous Markov chains. *The Annals of Statistics*, 47(4), 2023-2050.

Tsai, C. F., Hsu, Y. F., Lin, C. Y., & Lin, W. Y. (2009). Intrusion detection by machine learning: A review. *expert systems with applications*, 36(10), 11994-12000.

Tzimas, D., & Demetriadis, S. (2021). Ethical issues in learning analytics: a review of the field. *Educational Technology Research and Development*, *69*(2), 1101-1133.

Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of machine learning research*, 9(11).

Veerasamy, A. K., D'Souza, D., Apiola, M. V., Laakso, M. J., & Salakoski, T. (2020, October). Using early assessment performance as early warning signs to identify at-risk students in programming courses. In *2020 IEEE Frontiers in Education Conference (FIE)* (pp. 1-9). IEEE.

Venables, A., Tan, G., & Lister, R. (2009, August). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop* (pp. 117-128).

Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013, July). Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 117-122).

Wang, W., Dong, J., & Tan, T. (2010, September). Image tampering detection based on stationary distribution of Markov chain. In *2010 IEEE International Conference on Image Processing* (pp. 2101-2104). IEEE.

Watson, C., Li, F. W., & Godwin, J. L. (2013, July). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies* (pp. 319-323). IEEE.

Watson, C., Li, F. W., & Godwin, J. L. (2014, March). No tests required: comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 469-474).

Weiss, S. M., & Kapouleas, I. (1989, August). An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In *IJCAI* (Vol. 89, pp. 781-787).

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33-35.

Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM inroads*, 2(4), 71-76.

Zimek, A., & Filzmoser, P. (2018). There and back again: Outlier detection between statistical reasoning and data mining algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(6), e1280.

Lokkila, E., Rajala, T., Veerasamy, A., Enges-Pyykönen, P., Laakso, M. J., & Salakoski, T. (2016)
How students' programming process differs from experts – A case study with a robot programming exercise.
EDULEARN16 Proceedings (pp. 1555-1562)

I

Lokkila, E., Kaila, E., Lindén, R., Laakso, M. J., & Sutinen, E. (2017) Refactoring a CS0 course for engineering students to use active learning.

Interactive Technology and Smart Education.

II

Lokkila, E., Christopoulos, A., Laakso, M. J. (2022)
Automatically detecting previous programming knowledge from novice programmer code compilation history.
Informatics in Education.

III

Lokkila, E., Christopoulos, A., & Laakso, M. J. (2022)
A Clustering Method to Detect Disengaged Students from Their Code Submission History.
Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1 (pp. 228-234)

IV

Lokkila, E., Christopoulos, A., Laakso, M. J. (in press)
A Data Driven Approach to Compare the Syntactic Difficulty of Programming Languages.
Journal of Information Systems

V