Machine Learning Applications Supporting Large Scale Programming Education

Sami Sarsa
Supervising professor
Professor Lauri Malmi, Aalto University, Finland

Thesis advisors
Doctor Arto Hellas, Aalto University, Finland
Doctor Juho Leinonen, Aalto University, Finland

Preliminary examiners
Professor Peter Brusilovsky, University of Pittsburgh, USA
Assistant Professor (Sharon) I-Han Hsiao, Santa Clara University, USA

Opponent
Distinguished Professor Tiffany Barnes, North Carolina State University, USA
Abstract

Providing effective individualized education at scale has been a widely explored topic in education research, and the advancement of recent machine learning methods has made it possible to develop increasingly effective adaptive and intelligent learning systems. In particular, the emergence of deep learning models, and most recently large language models, has propelled the educational field forward, providing both new challenges and opportunities for educators. This dissertation addresses some of these challenges and opportunities, focusing on machine learning methods as a means to enhance large scale programming education.

We first present methodological considerations for identifying learners at risk of dropping out, and empirical evaluation of modern machine learning approaches for evaluating student mastery of skills. Then, we analyse features that relate to students continuing in a series of open online courses for introductory programming.

Relating to the constant need to produce new learning materials to keep course content relevant in the rapidly evolving landscape of programming and computer science, and the fact that producing such materials with appropriate quality can be a highly time-consuming task for educators, we propose and evaluate a novel approach that leverages large language models to create learning materials, particularly programming exercises and code explanations, which can be personalized for student needs and interests for increased engagement. The approach shows promising results in generating diverse, coherent, and relevant content. Most of the generated exercises were considered sensible, novel, and adhering to given themes and concepts. Further, we evaluate automatically generated code explanations in real educational settings and show that students tend to rate automatically generated explanations useful for their learning, even higher than those of their peers.

As means to help students, this dissertation looks into improving the timeliness of feedback, a key aspect in the effectiveness of feedback. This is done through proposing a framework that includes a machine learning step for speeding up automated assessment, which consequently speeds up assessment feedback, and constructing annotated datasets of when and how experts provide feedback and hints to learning programmers that can be used as a reference on when and how future machine learning models or other automated methods should provide feedback.

As a whole, the scope of the dissertation encompasses much of the entire educational process, spanning from (1) identifying learners needs and those who would benefit from additional assistance, to (2) educators designing content for learning and practice to (3) helping learners through timely and meaningful feedback for learners. The results in this dissertation showcase both methodological issues as well as new avenues for enhancing large scale computing education through machine learning methods.

Keywords: computer science education, learning analytics, machine learning, large language models, learner modeling, automated feedback, robosourcing

ISSN (printed) 1799-4934  ISSN (pdf) 1799-4942

Location of publisher: Helsinki  Location of printing: Helsinki  Year: 2024
Tiivistelmä

Oppilaiden yksilöllisten tarpeiden huomioiminen laajamittaisen opetuksen toteuttamisessa on haastava ongelma, jota varten on kehitetty erilaisia tekoälyä avulla toimivia työkaluja ja älykkäitä oppimisjärjestelmiä. Viimeaikoina erityisesti syväoppimismallit ja laajat kielimallit ovat vieneet tekoälyn kehitystä eteenpäin avaten sekä uusia mahdollisuuksia että haasteita opettajille ja opetuksen kehittäjille. Tässä väitöskierrasssa tutkitaan koneoppimisenelmiä, jotka tarjoavat keinoja laajan mittakaavan ohjelmointikoulutuksen edistämiseen.

Väitöskierrasssa käydään ensin läpi menetelmiä ongelmi kurssilta putoamisvarassa olevien oppilaiden tunnistamisessa sekä opiskelijoiden taitojen automaattisessa mallintamisessa, minkä jälkeen tarkastellaan miten erilaiset opiskelijoiden piirteet vaikuttavat heidän jatkamisessaan avoimessa aloittelijataso ohjelmoinnin verkkokurssien sarjassa, joka on suunnattu elinikäisille oppijoille.

Tämän jälkeen, oppimateriaalien kehittämisen tueksi, esitetemme uuden laajoja kielimalleja hyödyntävän menetelmän. Menetelmän avulla voidaan tuottaa opiskelijoiden tarpeiden ja mielenkiinnon mukaan räättäöityjä ohjelmointihajoitustesteitä sekä ohjelmakoordin selityksiä. Lisäksi työssä kehitettiin automaattisesti tuotetta, mutta automaattisesti luotuja ohjelmокоordin selityksiä oppimiselleen hyödyllisinä ja arvostelutavat he jopa paremmiksi kuin vertaisopiskelijoiden laatinut selitykset.

Lopuksi väitöskierrasssa tarkastellaan opiskelijan ohjelmointiyöstä annetun palautteen nopeutta. Ohjelmointitehtävien tarkastamiseen liittyen esitettemme koneoppimista hyödyntävän automaattisen tarkastamisen viitekehyksen, joka nopeuttaa palautteenantamista verrattuna perinteiseen tapaan arvioida ohjelmia automaattisesti. Opiskelijat eivät myöskään aina pyydä apu tai palautetta etenemisestään, vaikka avunpyytäminen olisi tarpeellista. Tähän liittyen laadimme asiantuntijoiden kommentointia aineistojärjestelmä, jotka voivat toimia viitteenä koneoppimismalleille tai muille menetelmille siitä, milloin ja miten ohjelmoinnin oppijoille kannattaa antaa palautetta tai vihjeitä heidän ohjelmointitehtävissä etenemisenä perusteella. Yhteenvetona väitöskiirja kattaa koneoppimisenelmiiden tutkimusta opetuksen edistämiseen ulottuen (1) opiskelijoiden tarpeiden tunnistamisesta ja lisäavun oikein kohdentamisesta, (2) opettajan sisällöntuottamisen helpottamisesta (3) palautteen antamiseen. Näihin liittyen väitöskierrassa käsitellään sekä menetelmällisiä ongelmia että uusia mahdollisuuksia edistää laajamittaista ohjelmoinnin koulutusta koneoppimismenetelmien avulla.
Preface

First, I want to give my sincere thanks to my supervisors, Arto Hellas, Juho Leinonen and Lauri Malmi. The guidance and support from each one of them have been indispensable. In particular, I am grateful to Arto Hellas for first being an inspiring teacher when I started studying computer science and recruiting me as a programming course assistant while I was still only a novice in programming, then, for supervising my Master’s thesis and afterwards asking me to start working on computing education research and build together an awesome online learning environment, and last but not least, for being a fantastic mentor.

I would like to thank my colleagues at the Department of Computer Science for all the great and insightful conversations we have had during the four years I have been with the LeTech Research Group - Learning + Technology. Especially, I am thankful to Charles Koutcheme and Lassi Haaranen with whom I’ve also had the joy to collaborate with on research. I am also grateful for the opportunity to begin my research career and collaborate with Eero Hyvönen and the Semantic Computing Research Group during my Master’s studies.

In addition to my supervisors, I have been able to collaborate with many great researchers during the recent years. I thank all my co-authors for our enjoyable collaboration and the invaluable contributions to our joint work, especially Paul Denny and Stephen MacNeil, who have been simply amazing people to work with.

I thank my pre-examiners, Peter Brusilovsky and Sharon Hsiao, for their insightful comments and suggestions. Professor Brusilovsky’s comments in particular were very thoughtful and helpful. I am grateful to Tiffany Barnes for agreeing to act as my opponent at the public defence of my dissertation.

I would also like to thank the people at Solita and the Agile Data Engine team for giving me a great place to grow as a programmer and being very lenient with me pursuing my studies while working part-time.
Finally, I give my thanks to my family and friends. I thank my dear wife for her love, encouragement and for being my best friend. I thank both of my parents for providing a safe and open environment to grow and learn. I have been especially inspired by my father’s enthusiasm for learning and understanding of the world. I thank my brother who has been a great friend as well as a source of inspiration and strength.

Δόξα ἐν ὑψίστοις θεῷ καὶ ἐπὶ γῆς εἰρήνη ἐν ἀνθρώπως εὐδοκίας
Glória in excélsis Deo et in terra pax homínibus bonæ voluntátis

Jyväskylä, February 14, 2024,

Sami Sarsa
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Publications</td>
<td>7</td>
</tr>
<tr>
<td>Author’s Contribution</td>
<td>9</td>
</tr>
<tr>
<td>List of Figures</td>
<td>13</td>
</tr>
<tr>
<td>List of Tables</td>
<td>15</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>17</td>
</tr>
<tr>
<td><strong>I Introduction and Background</strong></td>
<td>19</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>21</td>
</tr>
<tr>
<td>1.1 Scope of This Dissertation</td>
<td>24</td>
</tr>
<tr>
<td>1.2 Research Questions</td>
<td>25</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>28</td>
</tr>
<tr>
<td>1.4 Dissertation Structure</td>
<td>29</td>
</tr>
<tr>
<td>2. Machine Learning for Education</td>
<td>31</td>
</tr>
<tr>
<td>2.1 Dropout Prediction</td>
<td>31</td>
</tr>
<tr>
<td>2.2 Knowledge Tracing</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Evaluation Metrics and Baselines</td>
<td>36</td>
</tr>
<tr>
<td>2.4 Large Language Models for Educational Content</td>
<td>37</td>
</tr>
<tr>
<td>2.4.1 Large Language Models and Programming</td>
<td>38</td>
</tr>
<tr>
<td>2.4.2 Generating Educational Content for Programming</td>
<td>39</td>
</tr>
<tr>
<td>2.5 Automated Assessment and Feedback</td>
<td>42</td>
</tr>
<tr>
<td>2.6 Timeliness of Feedback</td>
<td>43</td>
</tr>
</tbody>
</table>
II Methodology and Results 45

3. Identifying Learners in Need of Additional Help or Practice 47
  3.1 Methodological considerations for predicting at-risk students 47
  3.3 Continuing in a Series of Finnish MOOCs 53

4. Creating Educational Content and Exercises 57
  4.1 Automatically Generating Programming Exercises and Explanations 57
    4.1.1 Exercises 58
    4.1.2 Code Explanations 60
  4.2 Evaluating LLM Generated Code Explanations With Learners 62
    4.2.1 Experiences With GPT-3 Generated Code Explanations 62
    4.2.2 Comparing GPT-3 and Learner Explanations 65

5. Towards More Timely Feedback 69
  5.1 Speeding Up Automated Code Assessment 69
  5.2 When, Why, and How to Give Feedback During the Programming Process 72

III Discussion and Conclusions 77

6. Discussion 79
  6.1 Discussion on RT1 79
    6.1.1 Dropout Prediction and Study Design 79
    6.1.2 Deep Learning Models for Knowledge Tracing 80
    6.1.3 Mind Your Metrics and Baselines 82
    6.1.4 Who continues in MOOCs? 84
  6.2 Discussion on RT2 85
    6.2.1 Pioneering Automated Programming Exercise Generation with Large Language Models 85
    6.2.2 LLMs as Shortcuts for Learning 87
    6.2.3 LLMs for Explanations and Feedback 88
  6.3 Discussion on RT3 90
    6.3.1 Faster and More Efficient Feedback 90
    6.3.2 Towards Automatic Feedback Interventions 92

7. Conclusion 95
  7.1 Revisiting the Contributions 95
  7.2 Piecing strings together 97
  7.3 Limitations of the Work 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4 Future Work</td>
<td>100</td>
</tr>
<tr>
<td>7.4.1 Methodological Research</td>
<td>100</td>
</tr>
<tr>
<td>7.4.2 Persistence in Online Education</td>
<td>101</td>
</tr>
<tr>
<td>7.4.3 Leveraging Large Language Models for Content Creation</td>
<td>102</td>
</tr>
<tr>
<td>7.4.4 Timeliness of Feedback</td>
<td>102</td>
</tr>
</tbody>
</table>

References

Publications

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>105</td>
</tr>
<tr>
<td>Publications</td>
<td>129</td>
</tr>
</tbody>
</table>
This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


VI Leinonen, Juho and Denny, Paul and MacNeil, Stephen and Sarsa, Sami and Bernstein, Seth and Kim, Joanne and Tran, Andrew and Hellas, Arto.


Author’s Contribution

Publication I: “Methodological considerations for predicting at-risk students”

All authors contributed to conceptualizing and designing the study. The dissertation author conducted the experiments jointly with the first author. All authors contributed to the writing of the article.


The dissertation author conceptualized and designed the study jointly with the third author and was responsible for designing and conducting the experiments. All authors contributed to the writing of the article.

Publication III: “Who Continues in a Series of Lifelong Learning Courses?”

All authors contributed to conceptualizing and designing the study. The dissertation author was responsible for designing and performing the experiments. All authors contributed to the writing of the article.

All authors contributed to conceptualizing and designing the study. The dissertation author was responsible for generating the data for the evaluation. The evaluation was conducted by all authors and all authors contributed to the writing of the article.


All authors contributed to conceptualizing and designing the study. The first, second, fourth and seventh author generated the explanations for the study and the dissertation author implemented the functionality to view and rate the explanations in the e-book. The analysis was done by the first, second, fourth and seventh author and the dissertation author. All authors contributed to the writing of the article.

Publication VI: “Comparing Code Explanations Created by Students and Large Language Models”

All authors contributed to conceptualizing and designing the study. The qualitative analysis was done by the first, third, fifth, sixth and seventh author. The quantitative analysis was done by the eighth author and the dissertation author. All authors contributed to the writing of the article.

Publication VII: “Speeding Up Automated Assessment of Programming Exercises”

All authors contributed to conceptualizing and designing the proposed framework. The dissertation author was responsible for the implementation and evaluation of the framework. The second and fourth authors and the dissertation author handled the most of the writing of the article.
Publication VIII: “Towards Giving Timely Formative Feedback and Hints to Novice Programmers”

This article was formed in an ITiCSE working group where a group of researchers work together to write a research report on a predetermined topic. The topic is set by the group leaders who also coordinate the research. The dissertation author participated in a subgroup that focused on creating an annotated dataset of when, how and why experts intervene and provide feedback to students who are in the process of solving programming tasks. The dissertation author participated in the designing of the annotation methodology, searched and provided data for annotation, and participated in annotating the data. The dissertation author contributed in the learning environment analysis by suggesting and describing environments for analysis. The other authors handled the rest. All authors contributed to the writing of the article.
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>On the left: the fields of learning analytics (LA) and educational data mining (EDM) in the intersection of computer science, education and statistics. On the right: the topic area of this dissertation in the intersection of computing education research (CER) and LA/EDM.</td>
</tr>
<tr>
<td>3.1</td>
<td>Performance of the models for accuracy and ROC-AUC (Receiving Operator Characteristic Area Under Curve) metrics for each round to predict. The scores are averaged over all previous round feature sets over all evaluated datasets.</td>
</tr>
<tr>
<td>3.2</td>
<td>Mean and maximum differences in metric scores compared to optimal when using different metrics to select the “best” hyperparameters. The loss indicates how much lower scores for other metrics can be expected when choosing a metric for hyperparameter optimization. The values are aggregates over each dataset and evaluated DLKT model.</td>
</tr>
<tr>
<td>3.3</td>
<td>Learner retention percentages for different groups. The x-axis represents total course completion rate, i.e., sum of introductory programming course (up to 1.0) and any continuation course (up to 1.0), totalling to a range from 0.3 (the minimum to respond to all surveys) to 2.0.</td>
</tr>
<tr>
<td>4.1</td>
<td>A comparison of the explanations generated by two large language models (GPT-3 and Codex) for three code snippets: (1) a ‘hello world’ server, (2) a server that counts POST requests, and (3) an example of object destructuring. The prompts used are shown. GPT-3 tended to produce more concise and consistently helpful explanations than Codex.</td>
</tr>
<tr>
<td>4.2</td>
<td>The interface for viewing GPT-3 generated explanations.</td>
</tr>
<tr>
<td>4.4</td>
<td>Boxplot of explanation usefulness ratings with + indicating mean. Although most viewed among students, line-by-line explanations were rated least helpful.</td>
</tr>
</tbody>
</table>
List of Figures

4.3 A line-by-line explanation generated by GPT-3 and the explanation feedback form. . . . . . . . . . . . . . . . . . . . . 65
4.5 Distribution of student responses on LLM and student-generated code explanations being easy to understand and accurate summaries of code. . . . . . . . . . . . . . . . . . . 66
5.1 Our proposed evaluation sequence. Each step in the sequence is the final step if a match or problems are found in the step.
1. if the solution is in the cache, return cached test results; 2. if static analysis finds errors, return those; 3. if the ML-model is rather certain that there is a problem, return feedback from the model; 4. execute unit tests and return their results. . . 70
5.2 Sankey plot that shows the proportion of (incorrect) submissions captured at each step of the approach. Precision threshold of 95% is used for the machine learning (ML) model. 72
6.1 An example of generating a hiking themed exercise for practicing loops with OpenAI’s ChatGPT (GPT-3.5). The image is from ChatGPT’s online user interface and taken in March 2023. 86
7.1 An illustration of the contributions of this dissertation from the perspectives of the learner and the educator. . . . . . . 97
List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The topics of this dissertation in the context of computing education research (CER) topic areas.</td>
</tr>
<tr>
<td>1.2</td>
<td>The topics of this dissertation in the context of the topics of computing education research.</td>
</tr>
<tr>
<td>2.1</td>
<td>Categorisation of basic knowledge tracing approaches by Pešek [218] with updated category names.</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of evaluated models.</td>
</tr>
<tr>
<td>3.2</td>
<td>Reported ROC-AUC score matrix for DLKT models trained on the ASSISTments2009 updated dataset.</td>
</tr>
<tr>
<td>3.3</td>
<td>Results for IntroProg dataset.</td>
</tr>
<tr>
<td>3.4</td>
<td>Results for ASSISTments 2015 dataset.</td>
</tr>
<tr>
<td>3.5</td>
<td>Surveyed characteristics and their distributions for various learner cohorts. Lower index ( i ) stands for introduction course completion and ( s ) for subsequent course completion.</td>
</tr>
<tr>
<td>4.1</td>
<td>Examples of uncommon code explanations produced in response to the speeding_check (A) and FizzBuzz (B, C) problems. (D) is an example of a nonsensical explanation of the Rainfall problem.</td>
</tr>
<tr>
<td>5.1</td>
<td>Guidelines that identify when and how to intervene.</td>
</tr>
<tr>
<td>5.2</td>
<td>Feedback characteristics of learning environments related to knowledge about mistakes.</td>
</tr>
<tr>
<td>5.3</td>
<td>Legend for feedback coding.</td>
</tr>
</tbody>
</table>
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>BoW</td>
<td>Bag of Words</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Science</td>
</tr>
<tr>
<td>CER</td>
<td>Computing Education Research</td>
</tr>
<tr>
<td>DLKT</td>
<td>Deep Learning (for) Knowledge Tracing</td>
</tr>
<tr>
<td>EDM</td>
<td>Educational Data Mining</td>
</tr>
<tr>
<td>ITS</td>
<td>Intelligent Tutoring System</td>
</tr>
<tr>
<td>KT</td>
<td>Knowledge Tracing</td>
</tr>
<tr>
<td>LA</td>
<td>Learning Analytics</td>
</tr>
<tr>
<td>LLM</td>
<td>Large Language Model</td>
</tr>
<tr>
<td>MAE</td>
<td>Mean Absolute Error</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>MCC</td>
<td>Matthew’s Correlation Coefficient</td>
</tr>
<tr>
<td>MOOC</td>
<td>Massive Open Online Course</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>PR-AUC</td>
<td>Precision-Recall Area Under the Curve</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root Mean Squared Error</td>
</tr>
<tr>
<td>ROC-AUC</td>
<td>Receiver Operating Characteristic - Area Under the Curve</td>
</tr>
<tr>
<td>RT</td>
<td>Research Theme</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Question</td>
</tr>
</tbody>
</table>
Part I

Introduction and Background
1. Introduction

Advances in technology and digitization have revolutionized the societies in which we live and participate. The pace of technological progress also appears to be only accelerating rather than showing signs of slowing down. Within a span of only a few years, the recent developments in artificial intelligence (AI) based text and image processing technologies have empowered us to automate more complex tasks than ever before. We can now automate not only routine tasks, but also the more intricate processes characteristic to white-collar professions.

In our increasingly technology-oriented world, technological literacy and programming are becoming more valued and more important skills, argued to be essential for participating in the modern workforce [126]. The World Economic Forum’s latest report [293] states that technology-related roles comprise the majority of the fastest-growing roles in the labour market, and that the demand for advanced technology professionals is expected to grow 30-35% in the next four years.

This rapid pace of advancement in technology — and consequential need for technology experts — makes the role of educators all the more important. In order to provide the necessary skills for the future workforce, the extent of technology education needs to match the rising skill demand. At the same time, the new and increasingly advanced topics that emerge need to be covered, further complicating this educational challenge. Simple — yet not necessarily easy — solutions such as opening more teaching positions by themselves are not enough on their own; only a limited number of skilled and motivated teachers and teaching assistants are available at a given time.

One way to tackle this is to teach more with less resources through the help of the same technology that needs teaching. Online education has been on the rise in the past decade, providing means for educators to provide education at a large scale. Massive Open Online Courses (MOOCs) have become a popular form of education, and the number of available MOOCs and learners taking them have been growing rapidly [63,256,280] — even before the recent pandemic which forced schools and universities to shift from traditional to online education.
However, the number of learners dropping out of online courses is a persistent challenge, and has been the focus of much education research [32, 244]. This has led to much research on the factors related to dropout rates to understand the problem [146], and on tools that can predict whether or when a particular student will drop out of a course or a curriculum [25, 231]. With accurate predictions, teachers can focus their efforts on the learners who are at high risk of dropping out and more likely to require additional attention and help. Yet, despite the major efforts in tackling MOOC dropout, the dropout rates remain particularly high [295]. More work is required to fully understand and address the problem.

Another challenge in education, one which when sufficiently addressed may shed some hope to the online dropout problem, is how to determine suitable learning content and exercises for learners who are at different stages of their learning journey. Exercises need to be of appropriate difficulty, at the proximal zone of development, to provide the desired learning effect [281]. Too easy tasks may lead to boredom and too difficult exercises may lead to frustration and demotivation, and ultimately to dropping out. It is also a long-known phenomenon that one-on-one tutoring for skill mastery leads to much more productive learning than education in a one-size-fits-all fashion [25]. But even though human experts could provide exercises at an appropriate difficulty-level for learners, this would be unfeasibly time-consuming with large groups of learners.

Personalized learning, often used to mean education that is tailored to individual learners’ prior knowledge, needs and interests [94, 145, 260], has been deemed important enough to be named as one of the 14 grand challenges [79] by The National Academy of Engineering. Not only is it important to optimize learning content for suitable challenge, but adjusting content to suit the learners’ goals and interests can greatly affect student engagement and motivation as well as overall performance [76, 227, 241, 243].

One major benefit of online education is the ability to collect data on student interactions with the learning platform. This data can be used to build models of student knowledge behaviour, which can then be used to recommend suitable learning content for learners based on their current skills and interests. As a means to tackle the tedious task of recommending suitable learning content, learning managements systems (LMS) and intelligent tutoring systems (ITS) have been extensively developed and evaluated for automatically deciding learning paths and practice exercises with the help of artificial intelligence [37].

These adaptive automated learning content providers are seen as the primary propellers of personalized learning in the recent decades [4]. Multiple studies have shown that learners who use adaptive learning systems outperform learners who receive conventional one-size-fits-all teaching [141, 284], and that learning with intelligent tutoring systems can be nearly as effective as human one-on-one tutoring [276, 178], or at least the best alternative [270].

In particular, techniques that track the learning progress of students based
on their prior task interactions, a specific area of learner modeling, collectively known as knowledge tracing (KT) has been — and still is — a key element in enabling automatic provision of content tuned to the learners’ current skill mastery.

Another problem arises from the issue of how to provide enough high quality content to be offered to learners based on their preferences and skill mastery levels. There is little use in highly proficient content recommenders with scant suitable content to recommend. In the realm of programming (which this dissertation focuses on), the time and effort it takes to write and maintain a large quantity of small exercises for drill and practice purposes has not gone unnoticed. When aiming for further personalisation of programming exercises, for instance with context based on learner interests, even more effort will be required. To enable efficient creation of scalable and intelligent tutoring systems in the ever-progressing programming world, methods to reduce effort in creating educational content will prove beneficial.

One thing that is inherently scalable in programming education, is the assessment of programming assignments. By writing code that inspects the behaviour of other code, one can reliably divine (to a certain limit) whether the code under inspection works as specified or not. Automated tests allow the provision of near immediate feedback — at least when compared to manual evaluation. Not having to wait, possibly days or even weeks, before knowing what went wrong or even whether something was wrong in the first place is a well evidenced advantage for learners that is shown to lead to better persistence and engagement in learning.

Traditional automated assessment of code, which is the typical practice to this day, has relied on executing code against instructor designed test suites or static analysis of source code. Machine learning methods are relatively new in this area, but offer much potential in providing immediate feedback. Trained machine learning models can give feedback much faster than methods that require running source code, and can provide feedback on issues that would be impractical to detect with static analysis methods. Further, through leveraging both learner and expert data, machine learning methods hold potential to improve the timeliness of feedback by learning the optimal times to intervene and provide meaningful feedback.

Motivated by the aforementioned challenges, this dissertation explores different ways how machine learning (a subfield of AI) applications can be used to (1) help learners learn programming more effectively and (2) to help educators provide quality programming education efficiently at scale. The focus is spread across the wide span of aiding education, from identifying struggling students and recommending exercises, to generating educational content and exercises and providing feedback.

*In this dissertation, “we” refers to the author of this dissertation and the co-authors the author has had the joy and pleasure to work with.*
1.1 Scope of This Dissertation

The topics covered in this dissertation lie within two overlapping research fields, learning analytics (LA) and educational data mining (EDM), that are often used interchangeably, sometimes mistakenly so. Both of these fields — which draw primarily from the broader fields of computer science (CS), education, and statistics — are concerned with the analysis and processing of data generated by learners and their learning activities. What separates the two fields from the broader area that is education, is their typical case of using data collected from students studying a subject organically rather than for a short period in a lab setting [245].

In this work, we focus on machine learning (ML) for computing education, although some of the included work concerns ML for education more broadly as well — ML is a subfield of AI that spans mathematics, statistics and CS, and is extensively explored for educational applications in EDM and LA. Thus, the topics of this dissertation also fall within the research field of computing education research (CER), also known as computer science education (CSEd) research, which is concerned with how computing is taught and learned. Here, we’ll refer to the field as CER. The topic area of this dissertation and how it is situated within the abovementioned fields is illustrated below in Figure 1.1.

![Diagram of fields and topic areas](image)

**Figure 1.1.** On the left: the fields of learning analytics (LA) and educational data mining (EDM) in the intersection of computer science, education and statistics. On the right: the topic area of this dissertation in the intersection of computing education research (CER) and LA/EDM.

The different topics covered in this dissertation in the context of CER topic areas are listed in Table 1.1. The topic areas in the table are derived from the books *Cambridge Handbook of Computing Education Research* [85] and *Computer Science Education Research* (CSER) [84].
Table 1.1. The topics of this dissertation in the context of computing education research (CER) topic areas.

<table>
<thead>
<tr>
<th>Book</th>
<th>CER area</th>
<th>Topic of dissertation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHCER</td>
<td>Systemic issues(^1)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>New Milieux</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Systems Software and Technology</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Teacher and Student Knowledge</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Case Studies</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Student understanding(^2)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Annotation, visualization and simu-</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>lation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Teaching methods</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Assessment(^3)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Education technology(^4)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Transferring professional practice to classrooms</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Incorporating new developments and new technologies</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Transferring from campus-based teaching to distance education</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Recruitment</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Retention</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Construction of the discipline</td>
<td>×</td>
</tr>
</tbody>
</table>

\(^1\)Systemic issues covers assessment and feedback.
\(^2\)One of the publications in this work covers knowledge tracing which is related to student understanding albeit not exactly what is meant by student understanding in [84].
\(^3\)Feedback is not explicitly mentioned in the list of topics in [84] but is part of formative assessment.
\(^4\)Education technology in [84] is a close match to Systems Software and Technology in [85].

1.2 Research Questions

The overarching theme of this dissertation revolves around a two-fold question of how to help learners learn programming and how to better scale programming education through machine learning methods. I formulate this question formally as *How can machine learning be leveraged in large scale programming education to (a) improve student learning and (b) reduce teacher workload?*
I narrow down the topic to the following three more specific research themes (RT) and related research questions that are addressed in the publications of this dissertation.\[^{1}\]

**RT1: Identifying learners in need of additional help or practice** The first research theme in particular is a theme that has been extensively explored in the fields of EDM and CER [115] yet remains an active area of interest [75]. The research on the theme concentrates on persisting and methodological issues within the contexts of course dropout prediction and student modeling. First, we identify an issue in prior dropout prediction research where ML models are trained on a set of course weeks to predict dropping out on future weeks. As most learners drop out early on, it is possible to achieve observably good performance by simply predicting that prior inactivity leads to future inactivity. In student modeling, the focus is especially on the new state-of-the-art deep learning for knowledge tracing (DLKT) family of ML models. Despite many new DLKT models displaying state-of-the-art performance [224, 302, 208], studies have also shown simpler models being able to achieve similar results [297, 91], and that model performance or even model ranking is difficult to replicate [91, 181, 208]. These observations motivate the following four research questions for RT1.

- **RQ1**: What performance differences do we observe in predictive models when we include versus exclude data from students who have already dropped out?
- **RQ2**: What features are most predictive of students dropping out when including versus excluding data from students who have already dropped out?
- **RQ3**: How do DLKT models compare to naive baselines and non deep-learning KT models?
- **RQ4**: What are the impacts of evaluation metrics, variations in architecture and hyperparameters on DLKT models’ perceived performance and ranking?

**RT2: Creating educational content and exercises** The second research theme concerns alleviating the process of creating learning materials for learners. Although learnersourcing [135, 266] is well-known as a means for producing learning content at scale, it involves risks of producing low-quality content [234, 225]. As a fresh alternative, this dissertation explores the application of generative AI in the form of the recently emerged highly capable large language models (LLMs). The capabilities of LLMs as learning content creators are explored through answering the following four research questions.

\[^{1}\]Due to the large amount of publications, I do not list all of the RQs addressed in the publications. The complete sets of RQs addressed in each study are presented alongside the key results in part III of the dissertation.
• RQ5: To what extent are LLM-generated programming exercises sensible, novel, and readily applicable?
• RQ6: How comprehensive and accurate are LLM natural language explanations of code solutions to introductory programming exercises?
• RQ7: What are the characteristics of LLM natural language explanations of code that students rate as most and least helpful?
• RQ8: To what extent do code explanations created by students and LLMs differ in accuracy, length, and understandability?

RT3: Towards more timely feedback
The use of automated tests that evaluate the execution of a program is a common approach in enabling near immediate feedback on student programs — often in the matter of seconds. Nonetheless, the execution of automated tests can still be a hindrance for optimal feedback timing. Depending on hardware and the tested program, text execution can easily take minutes or longer, especially when the tested program execution never halts (leading to a grading system timeout rather than feedback). In addition, feedback delays of mere seconds have been shown to have an effect on learning [255][199]. Assessment methods that do not require executing program code, such as static analysis or machine learning based methods, can provide feedback much faster, thus holding potential for improved or faster learning.

In addition, feedback is typically provided at the learners behest, even though early interventions could save significant time and reduce frustration in the learning process. However, providing proactive feedback portrays a challenge as feedback too often or too early can be distractive or even detrimental to learning [240], which underlines the possible benefits in optimising feedback interventions’ timing. Since employing ML models for a task requires appropriate data to train the models, in this dissertation, we focus the acquisition of such data and leave the actual ML part for future work.

The dissertation’s research questions that target these issues related to RT3 are as follows.

• RQ9: How much can simple ML models speed up programming assessment in an assessment framework consisting of cache, static analysis, ML and program execution checks?
• RQ10: How should we annotate datasets consisting of steps students take towards solving a programming task with information about when to give feedback and hints?
• RQ11: How does expert feedback relate to the feedback found in learning environments for programming?
1.3 Contributions

The eleven research questions of this dissertation are addressed and discussed in eight publications. In this introduction chapter, I only provide a brief outline of the publications’ contributions and how they relate to the research questions. The background literature, related work, gaps in research, as well as the publications themselves are discussed in more detail in the following chapters.

The contributions for the first research theme (RT1) focus on new methodological insight. In Publication I addressing RQs 1 and 2, we highlight a common flaw in measuring the capability of course dropout prediction models, and propose and discuss more fitting alternatives. In Publication II we address RQs 3 and 4 by empirically evaluating DLKT models’ performance against naive and other ML model baselines for a multitude of metrics and hyperparameter settings. We show e.g. how different hyperparameter variations affect different types of models differently, and how the choice of evaluation metric can easily affect the ranking of the models in the evaluation. In Publication III, we identify demographic and motivational features of learners who continue in a series of programming courses for lifelong learners, providing insight for dropout understanding and prediction in such situations.

In relation to the second research theme (RT2), this dissertation presents a novel method for automatically generating MOOC content through the use of large language models (LLMs). Our results show that the method enables generating both syntactically and semantically correct programming exercises, and helpful natural language explanations for code snippets (Publication IV), effectively answering RQs 5 and 6. Further explorations address RQs 7 and 8, and show that LLM generated code explanations can be perceived as useful by students (Publication V) and even be preferred over peer generated explanations (Publication VI).

The research questions for the final research theme (RT3) are addressed in two publications. Publication VII proposes a simple yet novel framework for speeding up automated assessment of programming exercises. The framework involves caching, static analysis and machine learning steps to provide near instant feedback as pre-steps before relying on potentially slow unit tests. To answer RQ9, we evaluate the potential benefits of the framework on an introductory programming course, showing that a simple ML model can quite reliably identify the majority of erroneous submissions that pass cache and static analysis checks, leading to a major speed up in automated assessment.

Publication VIII addresses the final two RQs (RQs 10 and 11). Within, we explore the state of current online learning environments for programming, and how expert feedback could help train or evaluate machine learning models to create better automated feedback. The work includes the formulation of how to annotate datasets with when, how and why expert educators would intervene on learners struggling with solving programming exercises.
As a summary, Table 1.2 maps the eight publications of this dissertation across the three RTs to the relevant CER topic areas portrayed in Table 1.1.

Table 1.2. The topics of this dissertation in the context of the topics of computing education research.

<table>
<thead>
<tr>
<th>Publication</th>
<th>EdTech</th>
<th>Assessment / Systemic issues</th>
<th>Retention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication I</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Publication II</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Publication III</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Publication IV</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication V</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication VI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication VII</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Publication VIII</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

1.4 Dissertation Structure

This dissertation is split into three parts. The first part outlines the work and provides background to the main topics of the dissertation. The second part summarises the publications of the dissertation, including the key results and methodology used to produce the results. The third and final part discusses the findings from the results, and concludes the dissertation with an overview of what was done and notions on the broader implications and future work.
2. Machine Learning for Education

Machine learning (ML) is a subfield within the broader field of Artificial Intelligence (AI). It is concerned with statistical models that learn from data to self-improve on their capability in making predictions or decisions on a given task. This separates ML from other forms of AI that owe their intelligent behaviour to e.g. hand-programmed conditional statements or direct computation of data.

ML comes in three flavours: supervised learning, unsupervised learning and reinforcement learning (and the different combinations of the three). In supervised learning, a machine learning model is trained with labeled data in a way that the model learns to predict the labels from other features in the data. The predicted label can be either categorical or numerical. In the former case, the task is called classification and in the latter case regression. Unsupervised ML models learn patterns from unlabeled data and can be used for instance to quickly group large amounts of such data into meaningful categories. Reinforcement learning involves an agent acting in a well defined environment where each action can be either rewarded or penalised. In the environment, the agent is directed to learn an optimal strategy that maximises the rewards through interacting with the environment.

In this dissertation, we focus on supervised learning methods and large language models (LLMs), which are generative models typically trained in an unsupervised fashion (their training may also involve supervised or reinforcement learning [304]). In particular, we focus on four machine learning tasks: performance and dropout prediction, knowledge tracing, generating educational content, and enhancing the delivery of feedback. Each of these tasks, as well as their relation to the publications of this dissertation, are outlined in the following sections of this chapter.

2.1 Dropout Prediction

Dropout prediction [231] is concerned with finding who are likely to drop out of a course and who are not, providing valuable insight for educators whose
efforts are best spent on those students who need additional assistance in their learning journey.

Owing to persisting high dropout rates, especially in online education [32, 95] and MOOCs [244], dropout and its potential remedies have been widely researched [146, 295, 231]. To underline the significance of the topic, Du et al. [75] list predicting student’s performance or the likelihood of dropout as the most popular one of the four major EDM research directions.

The prediction of course dropout is typically based on data collected from students’ previous interactions with learning materials, such as exercise submission related data, and possibly students’ self-reported characteristics obtained through surveys. Such data can then be used to train machine learning models to provide likelihood of students dropping out of a course. With the help of dropout prediction models, much effort has been poured into early warning systems to automatically highlight those who might benefit from an intervention [172] — for instance counseling or coaching [21], or automated feedback mechanisms [116, 62]. However, despite the vast research done on dropout prediction, the more recent surveys list multiple challenges that remain open in dropout research [231, 295]. These include for instance lack of standardised evaluation strategies, use of inconsistent definitions, and lack of unified theories and models.

On the more technical side, in their review of machine learning methods for dropout prediction [53], Dalipi et al. list the lack of balanced data as a prominent challenge for MOOC dropout prediction. For reliable classification performance, machine learning models need a large quantity of both positive and negative examples. As an example, Dalipi et al. note that the Harvard MOOC dataset [102] contains ~641 thousand registered students but only ~17 thousand who obtained course completion certification, resulting in a ratio of roughly 35:1 between dropouts (positive examples) and those who complete the course (negative examples).

Another challenge, besides the availability of quality data, is how the data is used to train and evaluate machine learning models. As an example, Whitehill et al. [288] take a critical look on how dropout prediction models are typically trained and evaluated on the data of one and the same course. Their study results tell that this can overestimate the practical performance of the model by several percentage points (in the situation of launching a new course): the model needs to be trained prior to its application, i.e. prior to launching the course, thus requiring data from previous courses.

Continuing in the vain of how data is used, many learners who drop out of courses do so early, often in the first weeks — or even before the first week if accounting for all registered learners who may not participate at all. Knowing this, it is possible that the data used to train and evaluate predictive models includes mostly data of learners who have already dropped out, leading to a situation where a model’s predictions are based on information that is not available in practice.
As an example, consider a course that has been running for four weeks and using a model to predict who will drop out on the next (fifth) week of the course. We have the options of predicting the next week dropouts out of everyone on the course, including inactive participants, or restricting the prediction to include only those who are still active. In such a situation, we may want to include all the students since there is no way of knowing who has truly dropped out and who has merely skipped a week or two of the course.

Now, consider training and evaluating machine learning models with full course data for the same situation. When training and evaluating machine learning models, we do have the information of whether a course participant becomes active after a period of inactivity. Including such dropouts in the training and evaluation process leads to two potential problems.

One is that the models, as they are trained to maximise accuracy or some other performance metric, overemphasise the importance of past inactivity since it is a feature that is consistently correlated with future inactivity. This can negatively affect (1) the models’ capability to correctly identify the less easily identifiable dropouts — the ones who have not yet dropped out of the course but will drop out on the coming weeks — and (2) the importance given to features not related to prior inactivity but still contributing to dropping out.

The second problem is inflated performance. As most learners drop out early, models can achieve observably good performance by just predicting that prior inactivity leads to future inactivity. This brings to question the usefulness of such models for trying to find future — not past — dropouts. This methodological misstep that has been present in prior research (e.g. [5, 172, 155]) is highlighted, analyzed, and discussed in Publication 1.

2.2 Knowledge Tracing

Knowledge tracing [47] is a learner modeling task in which learners’ actions within a learning environment are used to estimate their current knowledge or skills on learning topics. Knowledge tracing is strongly based on mastery learning [24, 23], a theory which posits that anyone can “master” most of what they are taught when given appropriate instruction at the level of the learner.

The key idea in knowledge tracing is to update an estimate of whether a learner has mastered a knowledge component — a concept or objective the learner is expected to learn — each time the learner has a chance to practice that particular knowledge component. Learner interactions within a learning environment are mapped to individual knowledge components, and the interactions may produce an output. Here, an output is observable evidence that indicates whether the learner’s action, for instance a response to an exercise, was correct or incorrect. These interaction outputs (mapped to the individual knowledge components) are then used to update knowledge tracing models’ internal estimates of the learner’s knowledge component mastery.
The use of knowledge tracing, and in particular bayesian knowledge tracing (BKT) [47] — as the scientific community has named the pioneering knowledge tracing model by Corbett and Anderson — is a widely studied approach in predicting learning and mastery for use in intelligent tutoring systems [2].

Over time, multiple approaches that leverage different machine learning techniques have been adapted for knowledge tracing. Pelanek [218] categorizes the basic approaches into five categories arranged by their complexity and whether they make assumptions about learning. The five categories are bayesian knowledge tracing, logistic models, baselines, generalisations, and black box models. I outline a slightly modified categorisation of knowledge tracing models in Table 2.1 along with a couple of model examples for each category.

In my version, BKT in Pelanek’s categorisation is replaced with bayesian methods as BKT is typically used to refer to the hidden Markov model based knowledge tracing method by Corbett and Anderson [47]. The name is changed in order for it to be more inclusive regarding other bayesian methods for knowledge tracing in the literature, for instance bayesian networks [247]. Logistic models is replaced with Factor analysis for consistency; the same (and other) logistic models are referred to as factor analysis models in both Publication II and the recent more comprehensive overview of knowledge tracing methods by Abdelrahman et al. [2] Baselines is replaced with naive baselines to emphasise that the models in this category do not learn from data but are very basic statistical models. Generalisations is replaced with bayesian + factor analysis for a more descriptive name as the category comprises models that combine bayesian methods and factor analysis models. Black box models is replaced with deep learning as deep neural networks — commonly referred to as deep learning models — form the remainder of knowledge tracing models not captured by the other categories and is a narrower term compared to black box models.

Note that the categorisation serves as a rough guideline rather than a rule. An intricate factor analysis method may very well be more complex than a combination of basic bayesian and factor analysis methods, and one could incorporate assumptions of learning into a deep learning model.

---

1I base my categorisation on Pelanek’s scheme [218] rather than that of Abdelrahman et al. [2] since Pelanek’s scheme is simpler and includes non-learning baselines that are relevant in our work in Publication II.
Table 2.1. Categorisation of basic knowledge tracing approaches by Pelanek [218] with updated category names.

<table>
<thead>
<tr>
<th>Assumptions of learning</th>
<th>No assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td></td>
</tr>
<tr>
<td>Bayesian methods(^1)</td>
<td>Factor analysis(^2)</td>
</tr>
<tr>
<td>Bayesian knowledge tracing (BKT) [47]</td>
<td>Learning factors analysis (LFA) [34]</td>
</tr>
<tr>
<td>Dynamic Bayesian Network (DBN) [247]</td>
<td>Performance factors analysis (PFA) [213]</td>
</tr>
<tr>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>Bayesian + Factor analysis(^4)</td>
<td>Deep learning(^5)</td>
</tr>
<tr>
<td>Latent factors knowledge tracing (LFKT) [131]</td>
<td>Deep knowledge tracing (DKT) [224]</td>
</tr>
<tr>
<td>Dynamic Item Response (DIR) model [285]</td>
<td>Dynamic Key-Value Memory Network (DKVMN) [302]</td>
</tr>
</tbody>
</table>

---

\(^1\) Bayesian knowledge tracing in [218] and probabilistic graphical models in Publication II.

\(^2\) Logistic models in [218].

\(^3\) Baselines in [218].

\(^4\) Generalisations in [218].

\(^5\) Black box models and includes both deep learning models and ensembles in [218].

**Bayesian methods** utilise graphs to illustrate conditional dependencies between studied variables, such as in Bayesian Knowledge Tracing (BKT) which models mastery of knowledge components using a hidden Markov model [47]. BKT updates knowledge component mastery based on student’s actions, with updates rooted in four parameters: prior learning, transition, guess, and slip. Over time, various BKT variants have been proposed. These variants have included for example introducing student-specific skill parameters [299], additional information on how students were helped during learning [163], and task difficulty on the process [212]. Other approaches include Partially Observable Markov Decision Processes [239] and (Dynamic) Bayesian Networks [247, 210, 123, 211].

**Factor analysis** models are logistic models that reduce observed variables into unobserved factors. They include for instance Learning Factors Analysis [34], Performance Factors Analysis [213], DASH [165] and DAS3H [40].

**Naive baseline** models consist of non-learning models including highly simple approaches like majority vote and moving average of past attempts, as well as more complex direct computation methods such as using the Elo rating system [80] for student modeling.

**Bayesian + Factor analysis** models include models that combine the logistic factor analysis models and bayesian methods. These include for instance the Item Response Theory (IRT) [101] based models LFKT [131], FAST [93], and DIR [285].
Deep Learning for Knowledge Tracing (DLKT) models are deep artificial neural networks models designed for the task of knowledge tracing. Deep Knowledge Tracing (DKT) [224] is the first such model and has shown great potential in a range of studies [291, 164, 182, 191, 69], leading to the proposal of a large quantity of more advanced DLKT models with varying architectures and additional input information. These variations include the use of memory networks [302, 1], attention based models [207, 209, 42], and graph neural networks [196, 269]. Additional input features have included for instance textual content of exercises [271, 169] and timestamps [233, 263]. In the context of computing education, Shi et al. [261] have combined the code2vec model [11] — which extracts meaningful vector representations of code — with DKT to achieve increased performance on programming knowledge tracing.

DLKT models are currently the newest and by a large margin most complex and popular group of knowledge tracing models with many recent studies displaying their impressive performance [2]. Despite this, comparison studies have shown that much simpler models, such as IRT models, can perform similarly as the DLKT models [297, 91]. Further, replicating the advances in performance portrayed in many novel DLKT model evaluations has appeared to be difficult. Replication studies have commonly reported the original DKT models among the best performing models [91, 171] and studies that measure the same models often lack consensus on model ranking.

For more comprehensive and elaborate overviews of knowledge tracing methods, see the background section of Publication II and especially the recent excellent survey by Abdelrahman et al. [2] that arranges knowledge tracing methods into deep learning knowledge tracing and traditional knowledge tracing methods and their subcategories.

The fact that new and increasingly complex models are being developed at a fast pace, at times with conflicting or irreproducible results, forms the premise for the empirical evaluation and methodological considerations presented in deep learning models for knowledge tracing in Publication II. While other work has also empirically evaluated DLKT models (e.g. [91, 208, 181]), our work concentrates more on methodological aspects related to hyperparameters and evaluation that are not targeted in the other studies.

2.3 Evaluation Metrics and Baselines

Evaluation metrics quantify model performance into a numerical score. They are essential in enabling the comparison of model performances, yet their proper application is non-trivial [197]. A multitude of different metrics have been proposed and employed, each of which evaluates different aspects of model performance, and using a single metric alone risks erroneous measuring of progress.
In educational data mining, some research has been done on how different metrics can influence the performance of models. For instance, Pelanek et al. [217] compared Mean Average Error (MAE), Root Mean Squared Error (RMSE), Log-likelihood (also known as binary cross-entropy), and ROC-AUC (Receiving Operator Characteristic Area Under Curve) in relation to various models including BKT, PFA, and adaptations of the Elo rating system [216]. More recently, Effenberger and colleagues [28] examined the closely related metrics, MAE and RMSE, across multiple student modeling systems, including the aforementioned Elo adaptation, an IRT model, and Random Forest. Their findings indicated that the choice of metric can influence not only model performance but also the ranking of models.

In addition to metrics, adequate baselines are essential for determining model goodness, especially when evaluating models on fresh data. Choosing only previous state-of-the-art models as baselines, while indicating the possible improvement in the task, may not tell much about the practical utility of the model. Comparison to naive baselines that are highly simple to interpret and whose outputs can be directly computed from data can provide a better perspective towards the predictive power of a machine learning model. This is because the performance of naive baselines can tell us a lot about the evaluation data and how e.g. skew in data influences metric scores. For example, accuracy, precision and recall are known to be potentially misleading [108, 160] and ROC-AUC is known to risk masking poor performance on highly skewed datasets [174, 249]. Involving naive baseline models such as majority vote helps avoid interpreting high accuracy, or other scores, as a sign of a good model when such scores could be trivial to achieve — the difference in performance to majority vote, which spells out the skew of data, is more important than the often (with good reason) critiqued raw accuracy score [64, 121].

The work in Publication II adds also to this line of work by comparing the performance of deep learning models for knowledge tracing to naive baselines and other knowledge tracing models. We employ multiple metrics in our model evaluation and dive into how evaluation metric choice can affect the DLKT model ranking and perceived relation to naive baselines.

2.4 Large Language Models for Educational Content

LLMs are huge text generating neural network models that are trained on vast amounts of data. They are able to generate relatively large pieces of text that can be difficult to distinguish from that of humans, even on difficult topics that require some level of expertise [27, 28]. LLMs trained on a broad range of textual data are also often referred to as (pre-trained) foundation models [304] as they are able to perform well in a multitude of downstream NLP tasks such as question answering and text classification [138], serving as a foundation for fine-tuned more task-specific models.
The surprisingly capable LLM models have sparked even discussions on whether they can understand the meaning of the data they are being trained upon, or whether they exhibit general intelligence [28]. One way to understand LLMs is by thinking of them as nothing but next token probability distributions, essentially stochastic parrots [20] — albeit extremely complex ones. Nonetheless, researchers have explored LLM’s “understanding” of concepts. Some evidence points to LLM’s being able to learn internal representations of concepts, such as legal rules in a board game [159]. But overall, the consensus tends to be that LLMs do not have a human-like understanding. Rather, they have some form of an internal representation of the world in their vast number of learned parameters [189] (some models exceeding the number of neurons in human-brain, averaging at around 86 billion [15] — naturally, a mere higher number of neurons doesn’t directly translate to higher level of complexity, and new LLMs can be more effective while having lower parameter counts [105]).

Regardless of how LLMs are able to produce or understand the textual content that they receive or output, their ability to produce and manipulate text has shaken the world and received a fair share of media attention as well [154]. As a consequence of their recent introduction to the public, they are greatly affecting many fields also outside of machine learning and natural language processing research, including programming [52, 296, 272] and education [73, 124] — not to mention programming education [86, 56].

Apart from the first two publications, the rest of the publications are heavily programming education oriented. Thus, the following sections are about computing education more specifically rather than scaling education in general.

### 2.4.1 Large Language Models and Programming

In the early 2020s, a short while after the emergence of early LLMs in 2018 (ELMo [221], BERT [65], GPT [235]), large language models were being trained specifically for programming tasks. For example, Mashhadi et al. [184] use the early LLM for code, CodeBert [83], to fix roughly 20% (unique) to 70% (including duplicated) bugs in two datasets.

In July 2021, OpenAI presented Codex [56], a GPT-3 [27] model (the third iteration of GPT), that was trained with 159GBs of Python code from GitHub, in addition to the data that GPT-3 itself was trained on. The Codex model, being much larger than its predecessors in terms of tunable parameters, was also orders of magnitude more performant [298], capable of generating human-like programming code from natural language instructions. Another similarly performant model was Google’s AlphaCode [161], published in 2022, which outperformed 50% of human programmers in competitive programming [161, 156]. With the advent of LLMs as capable as Google’s AlphaCode and OpenAI’s GPT-3 and Codex, educational researchers have been quick to compare the performance of these models to that of programming students [86], showing that Codex can achieve CS1 test scores in the top quartile of class performance.
LLM-powered AI tools have also been marketed as coding assistants for some time prior to Codex. For instance, GPT-2 powered TabNine has been around since 2019 [279]. But these AI programming aides really took off with Microsoft’s GitHub Copilot (powered by OpenAI Codex) that was marketed as “Your AI pair programmer”.

While these tools are mainly aimed at professional developers (and appear to be useful [219] albeit carrying risks regarding e.g. overreliance and producing bugs [214][220]), they also have great potential for supporting students in their learning. A recent study by Kazemitabaar et al. [127] has shown that for young programmers aged 10-17, having access to AI support was beneficial for obtaining good learning results. Students that had access to OpenAI Codex had increased completion rates and scores on programming tasks, and they performed slightly better on post-test evaluation where the use of additional tools such as Codex was not permitted.

2.4.2 Generating Educational Content for Programming

Programming courses often follow a mastery learning approach where learners work on numerous small exercises to build on mastery of core programming skills while avoiding the pitfalls of encountering too complex challenges too early [10][77][278]. Although the grading of the exercises can be largely automated, writing and maintaining such small programming exercises is a tedious task [173][294], leading to a need for tools to ease the process.

Further, learner’s interest has been shown to be critical to learner engagement, motivation, and also learning gains [76][243]. Indeed, in the context of programming, a recent study by Postner et al. [227] describes how humanitarian-oriented applications were more interesting and engaging for women as opposed to non-humanitarian applications. Another study, by Leinonen et al. [152], hints that contextual narrative can be helpful for learning, especially when the context of the narrative is familiar for the learner.

The required effort to provide such tailored exercises scales proportionally with the number of learners. Thus, providing such exercises with a sound amount of human resources would require crowdsourcing [7][286] or highly effective automation — robosourcing [59].

Generating Exercises

Early attempts at automatically generating programming exercises were based on programming templates and originated from the field of generative programming [50][46]. In 2010 Radošević et al. [238] proposed an automated exercise generation system based on their scripting model of application generators [236][237]. The system contained sets of templates for both exercise description and programming code where parts of the templates are filled with specification values and/or lower-level templates.

Later template based exercise works include Wakatani and Maeda’s [282, 283] system from 2015 that fills template codes with varying bugs and variable names, and Zavala and Mendoza’s [301] work from 2018 that extends traditional template based exercises by leveraging Linked Open Data [3] to create contextualised template based exercises.

The template based exercises provide a way to personalize exercises, effectively addressing issues for possible low-effort plagiarism of solutions. In addition, the approach by Zavala and Mendoza [301] makes it possible to personalize the exercises based on learner interests with simple contextualisation. However, template based systems are incapable of providing “novel” exercises or exercise descriptions per se. Rather, they only allow restricted variations of already crafted exercises. The workload of coming up with different kinds of exercises largely still remains on humans designing the exercises, and crafting good templates for different kinds of exercises requires significant effort.

One prominent avenue for offloading the work of generating exercises fully (or nearly fully) is learnersourcing [135, 266], that is, outsourcing such work to the learners themselves. Khosravi et al. [134] depict learnersourcing as “a pedagogically supported form of crowdsourcing that mobilises the learner community as experts-in-training to contribute to teaching or learning while being engaged in a meaningful learning experience themselves”.

Besides providing scalable means to reduce the work of teachers, learnersourcing offers additional benefits when learners create exercises themselves. Producing novel learning content encourages learners to deeply engage with and comprehend targeted concepts, and the act of content creation leads to enhanced information recall as opposed to passive engagement with content produced by others [66]. Moreover, content generated by peers can be more accessible to novices compared to expert-created resources due to the ‘expert blind spot’ phenomenon: experts forget what is difficult for novices and how concepts are best described for novices [97].

Nevertheless, learnersourcing is not without its challenges. Learnersourcing comes with the widely acknowledged risk of learners producing low-quality content that can carry little benefit for learning purposes [234, 225, 3]. The risk is underlined by the recent study by Pirttinen et al. [226], which displays a low inter-rater reliability between peers and experts in evaluating learnersourced content quality. Overcoming this risk would require training and supervising learners, which would hinder the scalability of learnersourcing [17].

Generating and Assessing Explanations

Understanding the behaviour of program code is a core skill for programmers. The ability to explain what code does in “plain English” [49] is found to have a strong correlation with programming ability (writing code) [175, 195], and an even stronger correlation can be found when explanation skills are combined with code tracing skills [277].

---

3. https://www.w3.org/standards/semanticweb/data
Although the correlation between explanation skills and programming ability is evident, learners have been found to have difficulties explaining code that they themselves have programmed. Using the think-aloud method \cite{274}, a study by Kennedy et al. \cite{128} showed how learners who are able to create programs with correct outputs may lack understanding of the programming language concepts and features they used. In another study, Lehtinen et al. \cite{149} explored how learners comprehend their own code submissions to exercises by asking the learners questions that target the submitted code. Their results show a large portion of learners failing to answer rudimentary questions such as “How would you describe the difference between returning a value and printing a value using console.log?”, even when having just successfully completed the coding exercise. Lehtinen et al. \cite{148,151} also propose and evaluate a system that automatically generates questions of learners’ codes, identifying that learners who repeatedly struggle answering the questions are more likely to drop out and have more difficulties when writing programs.

To help learners struggling with explaining code, one way is to provide them with examples of how it could be done. Examples play a central role in the early phases of learning \cite{275}, and thus it is easy to see how merely seeing code and a corresponding explanation might benefit learning programming and understanding code. However, like writing exercises, writing code explanations of high quality requires a non-trivial amount of work.

A study by Williams et al. \cite{290} explored learnersourcing explanations showing that explanations produced by peers are considered helpful for learning, although not as much as explanations by an instructional designer. Nonetheless, as explained above, learnersourcing carries risks related to content quality. In addition, if we are to consider having peers explaining a large portion code exercise submissions, learnersourced content quality is likely to deteriorate if learners find constantly explaining others’ (possibly messy or overly complex) code unpleasant, annoying, or too difficult.

Motivated by the need to outsource generating exercises to learners, the need to provide high quality explanations for code at scale, and the recently emerged highly capable LLM models, Publication IV explores the state of robosourcing exercises and explanations in the era of LLMs. More precisely, it explores the capability of OpenAI Codex to generate both programming exercises from natural language descriptions, as well as natural language explanations of code.

Continuing on the venue of robosourcing code explanations, in Publication V we investigate student perceptions on the quality and usefulness of LLM generated code explanations, and in Publication VI, we compare the performance of GPT-3 to that of students in generating explanations for code.
2.5 Automated Assessment and Feedback

Immediate feedback has been long known to be preferred and more enjoyable over delayed feedback [81], and beneficial for promoting learning [82]. According to the recent comprehensive overview of automated assessment in computing education by Paiva et al. [206], the most common way to grade and subsequently provide feedback for programming exercises is through testing the functionality, i.e. outcome, of student code submissions. The review authors also note that such testing typically involves either comparing program outputs against predefined test cases, or using industry-standard tools such as unit testing libraries.

This typical code assessment scenario requires executing learner code submissions. However, executing learner programs and related tests can be slow, which is a possible cause of frustration for learners waiting to know whether their solution is appropriate or not. In addition, running programs can require plenty of computing power to quickly execute potentially a huge amount of potentially highly inefficient learner programs in order to keep grading times sufficiently low. This problem is compounded by problematic issues common in novice code, such as infinite loops [117] that cause programs to run indefinitely until externally stopped and break the feedback process. Unit tests and output comparison also cannot be used to identify all problems in code, for example those related to code styling or overly complex solutions.

An alternative method for providing code feedback is static analysis of source code, a topic of interest for well over two decades [118, 185] that has nonetheless been gaining more traction lately [206, 205]. Static analysis does not require executing programs and thus can be much faster to execute than unit testing. Static analysis also has potential in capturing code issues that are difficult or impossible to identify by only inspecting code execution. Over the years, static analysis has been used to identify a multitude of issues in code, including deviation from stylistic guidelines [54], complex code [143, 114], common mistakes [12, 14], presence of infinite loops [14, 30, 113], and plagiarism [122, 143, 292]. Static analysis of code has also been effectively used to provide next step hints for student solutions based on similar model solutions [129].

A later arrival in the code grading scene, is the use of machine learning techniques to assess code correctness and provide feedback [187]. As a more progressive form of static analysis, machine learning has been recognised for its great potential for rapid delivery of effective feedback that would be impractical with direct static analysis. For example, Sharma et al. [258] train neural networks to detect code smells and Shi et al. [262] use a code embedding model (a model that transforms code into a numerical vector representation of the code) called code2vec [11] to discover misconceptions from student code.
The potential of large language models for feedback has not gone unnoticed either \[124\]. Besides investigating LLMs for explaining code like in \[Publication V\] and \[Publication VI\], LLMs have been successfully explored for instance to suggest code edits that improve performance \[180\], and to suggest ways to improve Scratch projects \[74\]. LLMs are also shown to achieve state-of-the-art in automated program repair \[296, 140\], which can be used to provide feedback to learners debugging their programs.

With the emergence of LLM powered coding assistant tools such as the GitHub Copilot, their usefulness as an always present pair programmer has also been investigated both in educational \[230\] and professional \[246\] settings. The conducted studies show promise in the use of such tools, despite the struggles novices are likely to face \[230, 52\], and the risks of overreliance and bugs in code \[214, 220\]. Although, it has been noted that LLM code completion accuracy drops drastically when source code includes bugs \[70\], and in the context of security bugs, Pearce et al. \[215\] note that LLMs are not good enough yet to deliver real value in a program repair framework \[215\]. On a more positive note, Dakhel et al. \[52\] compare junior developers and GitHub Copilot noting that while Copilot is more error prone than junior developers, its errors are less expensive to fix than what may be produced by junior developers for the same tasks. They also add that automatic repair tools can repair most of the bugs added by Copilot, further increasing its usefulness.

\[Publication VII\] brings these different approaches to code assessment together in the form of a framework where evaluating learner code happens in an efficient and effective manner to overcome the limitations of each individual method, such as the slowness of executing unit tests, the limits of static analysis, and the inevitable misclassifications by machine learning models.

### 2.6 Timeliness of Feedback

The effectiveness of feedback is highly dependent on when it is given. This is underlined through the repeated mentions in the literature about the value of timely feedback for learners \[158\]. While existing tools make it easy to provide immediate automated feedback when learners request feedback or assessment, at that point it may be too late for feedback to be timely. Indeed, when concluding a literature review on the power of feedback and presenting an effective model of feedback, Hattie and Timperley \[103\] note the importance of “having exquisite timing to provide feedback before frustration takes over”. Moreover, in Poulos et al.’s \[228\] investigation on the learner’s perspective of what is effective feedback, learners mentioned how receiving feedback — or even being forced to receive it — while they are doing work is much more helpful than getting feedback when the work is done.
A common approach to providing automated feedback and hints on programming tasks is to wait for the learner to request feedback themselves through an action such as clicking on a “provide hint” button or submitting their solutions for grading, which is evidenced in the learning environment review done in [Publication VIII]. Even though such feedback may be immediate in the sense that it is provided as soon as the learner requests it, it does not equate to providing feedback as soon as the learner needs it. For instance, it is common for learners to make a mistake and end up in a rabbit hole of thoughtless tinkering while not understanding the problem in their code [71,148], which can easily cause frustration taking over. Highlighting the issue, prior research on intelligent tutoring systems has shown that learners frequently avoid asking for help, even after making multiple mistakes [8,9].

A simple option for proactive feedback is to provide it whenever the learner makes a mistake. However, such policy may be more harmful than beneficial. As an example, when studying middle school students working on algebra, Razzaq and Heffernan [240] found that the students benefited more from on-demand hints rather than from proactive hints given whenever an error was encountered.

Instead of showing feedback for all errors, Cody et al. [45] investigated the effect of giving periodic unsolicited hints within a logic proof tutor for an undergraduate mathematics course. They observed that learning gains were similar between on-demand hints and periodic unsolicited hints, but if learners were periodically asked whether they would like to see a hint, the learning gains were lower. The results suggest that a more optimised hint policy through machine learning could remedy help-avoidance behaviour and increase learning gains.

Even with machine learning methods, identifying opportune timing to intervene automatically is not a trivial task. Therefore, in this dissertation we turn to first investigating data that could be leveraged by machine learning models for this particular task. When able to observe the process of a learner producing code, experienced educators can often identify situations where intervention is beneficial. In addition to the aforementioned learning environment review, [Publication VIII] investigates when, how and why programming education experts intervene on students as they work towards solving programming tasks. Data driven programming feedback has been shown to be on par with expert feedback [303] but the field of providing timely interventions during programming remains underexplored. [Publication VIII] forms an important preliminary for designing and evaluating automated timeliness-optimised programming feedback systems with intervention capabilities. As part of the work, datasets with fine-grained steps of student progress in coding exercises were annotated with the expert interventions. The annotated datasets were made publicly available as part of the work.

Part II

Methodology and Results
3. Identifying Learners in Need of Additional Help or Practice

The first research theme of this dissertation is: *Identifying learners in need of additional help or practice*. Three publications of the dissertation contain work related to this theme. Publication I and Publication II offer insights on methodological aspects in two widely researched areas for identifying students in need of extra help. Namely, these are course drop out prediction and knowledge tracing. The third publication, Publication III also relates to course dropout, and offers a fresh look into factors that correlate with continuing in a series of Finnish MOOC courses for lifelong learners.

3.1 Methodological considerations for predicting at-risk students

In the context of identifying students at risk of dropping out through machine learning models, Publication I inspects the effect of including already inactive (dropped out) students in the training and evaluation data of the models — a practice observed in the literature (see the publication and section 2.1). The publication critically examines the approach where data collected from weeks 1, 2, ..., $n$ is used to predict whether a student becomes inactive in the subsequent weeks $w$, $w \geq n + 1$.

This is done by measuring model performances using two approaches: one that includes all student data, and one that excludes students who have dropped out prior to the predicted week $w$. In addition to exploring model performance differences between the two approaches, we investigated the differences in feature importances given by the trained models using the two approaches. Thus, the publication addresses RQs 1 (*What performance differences do we observe in predictive models when we include versus exclude data from students who have already dropped out?*) and 2 (*What features are most predictive of students dropping out when including versus excluding data from students who have already dropped out?*) of the dissertation. The publication also addresses one additional RQ (*How does the course context influence the performance of the predictive models?*), which is left out of the key result summary of this dissertation.
The study was conducted using log data from three online university courses, where each course had different instructors, themes, and target audience. For the dropout prediction, we used two machine learning models commonly used for dropout prediction, namely: logistic regression and random forest. We also included two naive baselines models, majority vote and naive bayes, for validating the usefulness of the evaluated machine learning models.

Our key finding is that including already dropped out students holds a high risk for greatly inflated predictive power of machine learning models. Figure 3.1 shows the large difference in performance between the approach where inactive students are included and the approach where they are excluded. The difference is particularly evident at later weeks, and when looking at the machine learning models’ accuracy compared to the majority vote baseline (predict everything as the most common target). Note that the difference to majority vote is more descriptive of model performance than the raw accuracy value since accuracy is highly affected by skew in the data. The different accuracy scores for the majority vote in the two scenarios is an indication of skew changing when inactive students are excluded from the evaluation data. The results in the figure are averaged over the scores for each evaluated previous round feature set for the predicted round (e.g. round 3 was predicted using two feature sets: \{ round 1 features \} and \{ round 1 features, round 2 features \} ) and all evaluated datasets. More detailed results can be inspected in the publication.

![Figure 3.1. Performance of the models for accuracy and ROC-AUC (Receiving Operator Characteristic Area Under Curve) metrics for each round to predict. The scores are averaged over all previous round feature sets over all evaluated datasets.](image)

We also found that feature importances computed from the trained models varied greatly between the two approaches. However, we did not see clear evidence of features types (correctness vs attempt count) being inherently different between the approaches.

For Publication II, we conducted a study to replicate the promising results portrayed by emerging deep learning based models for knowledge tracing. We investigated the impact of different hyperparameter settings, architectural variations, and evaluation metrics on the ranking of multiple knowledge tracing models. In the study, we address RQs 3 (How do DLKT models compare to naive baselines and non deep-learning KT models?) and 4 (What are the impacts of evaluation metrics, variations in architecture and hyperparameters on DLKT models’ perceived performance and ranking?) of the dissertation. Like Publication I, this work also focuses on performance on different datasets, answering another RQ (How do DLKT models perform on the same and different datasets as originally evaluated with), which is not emphasized here in the summary of key results.

The study includes replicating the work done in three prominent articles that each proposed a new DLKT model (or two) and showcased results indicative of performance that surpasses prior state-of-the-art. For the replication, we re-implemented the DLKT models. We also evaluated several non-DLKT models as baselines. These include non-learning naive baseline models, bayesian knowledge tracing (BKT), and a recent model that is based on popular factor analysis models. The evaluated models are listed in Table 3.1.

The work includes a minor review of the literature to investigate discrepancies in prior reported results for the DLKT models that we evaluated. Differences in reported model ROC-AUC scores for the commonly used ASSISTments2009 dataset are shown in Table 3.2. Although not evaluated in our work, the table includes scores for LSMT-DKT+ from its respective paper to provide more values for comparison. Notably, many of the differences are large, in the magnitude of several percentage points (much of this is likely due to differences in data preprocessing), and the ranking of models varies greatly between articles.

We evaluated the studied models with seven different metrics and on seven different datasets, where one dataset is a new dataset consisting of learner data from an introductory computing course organised by Aalto university. This dataset is published as part of the work. Details on the used evaluation metrics and datasets as well as the evaluated models can be found in the paper.

When comparing model performances, we found that the neural network based DLKT models in general indeed are the new state-of-the-art approach for knowledge tracing. However, we found much room for improvement and that DLKT models are not a silver bullet for best performance. Comparatively a much simpler logistic regression based model (GLR) outperformed all of the evaluated DLKT models on one evaluated dataset on most metrics (see Table 3.3). On another dataset, a simple mean prediction (predict everything as the average of training data targets) achieved performance comparable to that
## Table 3.1. Summary of evaluated models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Short-hand</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean prediction</td>
<td>Mean</td>
<td>A simple statistic</td>
</tr>
<tr>
<td>Next as previous</td>
<td>NaP</td>
<td>A simple baseline model</td>
</tr>
<tr>
<td>Next as previous N’s mean</td>
<td>NaPNM</td>
<td>A slightly less simple baseline model</td>
</tr>
<tr>
<td>Non-DLKT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bayesian Knowledge Tracing</td>
<td>BKT</td>
<td>Commonly used as a baseline, predecessor to DLKT models</td>
</tr>
<tr>
<td>Gervet et al. Logistic Regression</td>
<td>GLR</td>
<td>Logistic regression model with best input feature combination in 301</td>
</tr>
<tr>
<td>Deep Learning Knowledge Tracing (DLKT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long Short Term Memory (Recurrent Neural Network) Deep Knowledge Tracing</td>
<td>LSTM-DKT</td>
<td>First DLKT model 224</td>
</tr>
<tr>
<td>Vanilla (Recurrent Neural Network) Deep Knowledge Tracing</td>
<td>Vanilla-DKT</td>
<td>First DLKT model along with LSTM-DKT 224</td>
</tr>
<tr>
<td>Dynamic Key-Value Memory Network (MXNet implementation)</td>
<td>DKVMN</td>
<td>First DLKT model with separate next attempt skills as input 302</td>
</tr>
<tr>
<td>Dynamic Key-Value Memory Network (as depicted in its respective paper)</td>
<td>DKVMN-Paper</td>
<td>Same as above</td>
</tr>
<tr>
<td>Self Attentive Neural Network</td>
<td>SAKT</td>
<td>A DLKT model based on Transformer neural network 207</td>
</tr>
<tr>
<td>LSTM-DKT with next skill input</td>
<td>LSTM-DKT-S+</td>
<td>LSTM-DKT variation with added skill input as in DKVMN and SAKT, presented in our work</td>
</tr>
</tbody>
</table>

of the DLKT models (see Table 3.4). Note that the tables contain results for a subset of the evaluated models. Only the results for the best models among model variations are portrayed here for the sake of brevity (e.g. LSTM-DKT is included but Vanilla-DKT is omitted). The full results are presented in the paper.

Our hyperparameter effect analysis shows how bad choices in DLKT model hyperparameter tuning can easily change the performance rankings of the evaluated models, and that the effect of seemingly unimportant architectural variations or hyperparameter choices should not be overlooked. For instance,
Table 3.2. Reported ROC-AUC score matrix for DLKT models trained on the ASSISTments2009 updated dataset.

<table>
<thead>
<tr>
<th>Article</th>
<th>LSTM-DKT</th>
<th>LSTM-DKT+</th>
<th>DKVMN</th>
<th>SAKT</th>
<th>GLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DKT</td>
<td>86</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DKT+</td>
<td>82.212</td>
<td>82.227</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DKVMN</td>
<td>80.53</td>
<td>-</td>
<td>81.57</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SAKT</td>
<td>82.0</td>
<td>82.2</td>
<td>81.6</td>
<td>84.8</td>
<td>-</td>
</tr>
<tr>
<td>GLR</td>
<td>75.7</td>
<td>82.2</td>
<td>82.0</td>
<td>84.8</td>
<td>77.2</td>
</tr>
<tr>
<td>this</td>
<td>81.4</td>
<td>-</td>
<td>80.9</td>
<td>79.8</td>
<td>72.9</td>
</tr>
<tr>
<td>avg (sd)</td>
<td>81.3 (3.33)</td>
<td>82.2 (0.02)</td>
<td>81.4 (0.40)</td>
<td>80.1 (4.61)</td>
<td>75.1 (3.04)</td>
</tr>
</tbody>
</table>

A row contains an article identifier and the best ROC-AUC scores (as a percentage) for models reported in that article. Note that the high variance is partly explained by differences in the data used in training the models in the different studies.

When splitting inconveniently long attempt sequences to reduce input length, the choice of maximum attempt count can affect different models differently. In the dataset ASSISTments 2009, LSTM-DKT benefited very slightly from no splitting as opposed to best evaluated splitting (0.1% points for both ROC-AUC and RMSE) but DKVMN-Paper and SAKT both received a major performance boost from splitting (>1% ROC-AUC and >0.5% RMSE). For scale, even 0.3% RMSE (Root Mean Squared Error) improvement in cognitive modeling has been shown to lead to more effective learning in practice [139].

We also found that even the hardware and framework version used to train the model may lead to a noticeable difference in scores. In one case, we found a 1.7% point difference in F1-score\(^1\) and 0.6% point difference in ROC-AUC between two TensorFlow\(^2\) versions, and 0.6% for both MCC and F1-score as the largest difference when training the model on a CPU instead of on a GPU. The largest observed difference in RMSE scores between the hardware and framework options was 0.2% points. However, the difference was not uncommon.

\(^1\)The harmonic mean of precision and recall
\(^2\)https://www.tensorflow.org/
Table 3.3. Results for IntroProg dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc</th>
<th>ROC-AUC</th>
<th>F1</th>
<th>MCC</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM-DKT</td>
<td>.757±.027</td>
<td>.827±.013</td>
<td>.758±.044</td>
<td>.491±.030</td>
<td>.406±.016</td>
</tr>
<tr>
<td>DKVMN</td>
<td>.756±.028</td>
<td>.827±.015</td>
<td>.756±.043</td>
<td>.490±.031</td>
<td>.406±.018</td>
</tr>
<tr>
<td>SAKT</td>
<td>.754±.029</td>
<td>.825±.015</td>
<td>.759±.045</td>
<td>.482±.034</td>
<td>.407±.020</td>
</tr>
<tr>
<td>GLR</td>
<td>.761±.007</td>
<td>.843±.008</td>
<td>.755±.004</td>
<td>.522±.014</td>
<td>.402±.005</td>
</tr>
<tr>
<td>BKT</td>
<td>.713±.012</td>
<td>.789±.012</td>
<td>.708±.041</td>
<td>.423±.012</td>
<td>.436±.004</td>
</tr>
<tr>
<td>Mean</td>
<td>.513±.013</td>
<td>.500±.000</td>
<td>.000±.000</td>
<td>.000±.000</td>
<td>.500±.000</td>
</tr>
<tr>
<td>NaP 5 Mean</td>
<td>.707±.008</td>
<td>.772±.010</td>
<td>.704±.006</td>
<td>.414±.017</td>
<td>.456±.006</td>
</tr>
</tbody>
</table>

Table 3.4. Results for ASSISTments 2015 dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc</th>
<th>ROC-AUC</th>
<th>F1</th>
<th>MCC</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM-DKT</td>
<td>.751±.027</td>
<td>.725±.020</td>
<td>.847±.021</td>
<td>.239±.013</td>
<td>.413±.019</td>
</tr>
<tr>
<td>DKVMN</td>
<td>.750±.028</td>
<td>.723±.020</td>
<td>.846±.021</td>
<td>.239±.015</td>
<td>.413±.019</td>
</tr>
<tr>
<td>SAKT</td>
<td>.748±.028</td>
<td>.714±.023</td>
<td>.846±.020</td>
<td>.230±.021</td>
<td>.415±.019</td>
</tr>
<tr>
<td>GLR</td>
<td>.750±.031</td>
<td>.702±.027</td>
<td>.849±.022</td>
<td>.204±.017</td>
<td>.416±.021</td>
</tr>
<tr>
<td>BKT</td>
<td>.747±.026</td>
<td>.694±.020</td>
<td>.847±.018</td>
<td>.209±.012</td>
<td>.421±.015</td>
</tr>
<tr>
<td>Mean</td>
<td>.738±.033</td>
<td>.500±.000</td>
<td>.849±.022</td>
<td>.000±.000</td>
<td>.440±.017</td>
</tr>
<tr>
<td>NaP 5 Mean</td>
<td>.704±.035</td>
<td>.624±.024</td>
<td>.808±.029</td>
<td>.158±.025</td>
<td>.463±.023</td>
</tr>
</tbody>
</table>

Besides the more obvious case of hyperparameters affecting performance and ranking, our comparison of different metrics shows that the choice of evaluation metric too can influence DLKT model ranking. For example, in Table 3.4, the best models according to F1-score are GLR and Mean (tied) while other metrics portray LSTM-DKT as the best model.

We further investigated how the choice of metric can affect performance in the hyperparameter tuning phase. Figure 3.2 shows how picking the best set of hyperparameters based on one metric can lead to a suboptimal hyperparameter set according to other metrics. We can also see that no single metric is clearly best for selecting optimal hyperparameters. Each choice risks lower performance for other metrics, some choices with a more balanced risk (more equally distributed losses across the other metrics), and some choices with a clearly higher risk for certain metrics than for others. The one metric that stands out in the results is F1-score, which appears to convey the overall greatest risk of selecting suboptimal hyperparameters for other metrics.
3.3 Continuing in a Series of Finnish MOOCs

In Publication III, we set out to investigate who continues in a series of Finnish introductory programming MOOCs aimed at lifelong learners. We analyzed how different learner characteristics — including self-reported motivations, previous experience, and demographics — influence student dropout. More precisely, we set out to answer the following three research questions: (1) “What characteristics (demographic, prior experience, motivations) do learners attending courses aimed at lifelong learning for programming possess?”, (2) “How are these characteristics related to performance in an introductory programming course, and continuing to a subsequent course?” and (3) “What is the relationship between gender, prior programming experience and completing the introductory and a continuation course?”.

The course series under focus offers teaching on the principles of computer science and programming for lifelong learners, and is organised by Aalto University. The series consists of four courses: (1) introduction to programming, (2) data and information, (3) internet and browser applications, and (4) mobile application development. Each course is worth 2 ECTS, which corresponds to approximately 50 to 60 study hours. The courses are graded pass/fail, and passing a course requires completing at least 90% of the available exercises.

The learner characteristics were self-reported through various surveys that were presented to the learners within the online course platform. Table 3.5 lists the surveyed characteristics and their distributions over three completion rate cohorts: 30% of introduction course completed (the final survey was introduced at this point), 90% of introduction course completed (participants
are eligible for course credits at this point), and 10% of a subsequent course completed. We excluded all learners with observed activity in the final 30 days to avoid interpreting still active learners as dropping out.

We studied the relation of the surveyed characteristics to course completion and continuation through descriptive statistics, regression analysis and statistical tests. For the regression analysis, we used linear regression and random forest models, and also mean and majority vote to serve as naive baselines for comparison.

We found that younger learners were less likely to finish the first course and much less likely to continue to another course, and that novices — while more likely to start the course series — likewise tended to dropout earlier than the more experienced programmers. We also found that out of self-reported genders, men were the most likely to both start and continue the courses. This was the case even when taking into account the other surveyed characteristics, signifying that the other explored characteristics did not account for all the differences between the genders. Out of the motivation characteristics, unsurprisingly, interest in the topic and wanting to learn about a specific technology were the most related to continuing the course series after completing the introductory course. However, wanting to complete a university course was more predictive of completing the introductory course than the surveyed internal motivation characteristics.

The plots in Figure 3.3 elucidate the relations of the surveyed prior experience characteristics and genders to completing the introductory course and continuing on another course. It can be seen that all of the experience features contributed to persisting and also that there is a clear gap between genders. We conducted a Kruskal-Wallis test to determine the relation between prior experience and gender, showing that only the number of courses taken explained continuing for women, while the effect of each of the prior experience features were statistically significant for men.
Table 3.5. Surveyed characteristics and their distributions for various learner cohorts. Lower index $i$ stands for introduction course completion and $s$ for subsequent course completion.

<table>
<thead>
<tr>
<th>group</th>
<th>characteristic</th>
<th>30%,$i$</th>
<th>90%,$i$</th>
<th>10%,$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>18-25</td>
<td>.228</td>
<td>.204</td>
<td>.156</td>
</tr>
<tr>
<td></td>
<td>26-35</td>
<td>.308</td>
<td>.291</td>
<td>.286</td>
</tr>
<tr>
<td></td>
<td>36-55</td>
<td>.333</td>
<td>.363</td>
<td>.442</td>
</tr>
<tr>
<td></td>
<td>56-</td>
<td>.031</td>
<td>.028</td>
<td>.030</td>
</tr>
<tr>
<td></td>
<td>undisclosed</td>
<td>.099</td>
<td>.114</td>
<td>.085</td>
</tr>
<tr>
<td>courses taken$^1$</td>
<td>0-1</td>
<td>.524</td>
<td>.376</td>
<td>.296</td>
</tr>
<tr>
<td></td>
<td>2-4</td>
<td>.306</td>
<td>.389</td>
<td>.421</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>.170</td>
<td>.235</td>
<td>.283</td>
</tr>
<tr>
<td>gender</td>
<td>woman</td>
<td>.361</td>
<td>.284</td>
<td>.251</td>
</tr>
<tr>
<td></td>
<td>man</td>
<td>.508</td>
<td>.574</td>
<td>.623</td>
</tr>
<tr>
<td></td>
<td>other or undisclosed</td>
<td>.131</td>
<td>.142</td>
<td>.126</td>
</tr>
<tr>
<td>education</td>
<td>other inapplicable or undisclosed</td>
<td>.120</td>
<td>.139</td>
<td>.126</td>
</tr>
<tr>
<td></td>
<td>secondary education or less</td>
<td>.245</td>
<td>.219</td>
<td>.207</td>
</tr>
<tr>
<td></td>
<td>some tertiary education</td>
<td>.635</td>
<td>.642</td>
<td>.667</td>
</tr>
<tr>
<td>lines of code$^2$</td>
<td>0</td>
<td>.373</td>
<td>.243</td>
<td>.208</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>.416</td>
<td>.482</td>
<td>.465</td>
</tr>
<tr>
<td></td>
<td>2-4</td>
<td>.211</td>
<td>.274</td>
<td>.327</td>
</tr>
<tr>
<td>motivation</td>
<td>it is free</td>
<td>.194</td>
<td>.173</td>
<td>.176</td>
</tr>
<tr>
<td></td>
<td>course was recommended to me</td>
<td>.044</td>
<td>.021</td>
<td>.005</td>
</tr>
<tr>
<td></td>
<td>for future career</td>
<td>.207</td>
<td>.159</td>
<td>.151</td>
</tr>
<tr>
<td></td>
<td>interested in the topic</td>
<td>.453</td>
<td>.422</td>
<td>.467</td>
</tr>
<tr>
<td></td>
<td>other or undisclosed</td>
<td>.066</td>
<td>.055</td>
<td>.055</td>
</tr>
<tr>
<td></td>
<td>relevant to current role</td>
<td>.064</td>
<td>.042</td>
<td>.045</td>
</tr>
<tr>
<td></td>
<td>to complete a university course</td>
<td>.130</td>
<td>.131</td>
<td>.111</td>
</tr>
<tr>
<td></td>
<td>to learn about a specific technology</td>
<td>.272</td>
<td>.242</td>
<td>.281</td>
</tr>
<tr>
<td>self estimated$^3$</td>
<td>1-2</td>
<td>.437</td>
<td>.310</td>
<td>.264</td>
</tr>
<tr>
<td></td>
<td>3-4</td>
<td>.344</td>
<td>.394</td>
<td>.396</td>
</tr>
<tr>
<td></td>
<td>5-9</td>
<td>.219</td>
<td>.296</td>
<td>.340</td>
</tr>
</tbody>
</table>

$^1$courses taken indicates the number of previously completed programming related courses

$^2$meanings of available lines of code options were: 0 = none, 1 = less than 500, 2 = 500-5000, 3 = 5001-4000, 4 = over 40000

$^3$self estimated (experience in programming) was reported on a scale of 1 (not at all experienced) to 9 (very experienced)
Identifying Learners in Need of Additional Help or Practice

Figure 3.3. Learner retention percentages for different groups. The x-axis represents total course completion rate, i.e., sum of introductory programming course (up to 1.0) and any continuation course (up to 1.0), totalling to a range from 0.3 (the minimum to respond to all surveys) to 2.0.

We also note that the performance evaluation results of the models show that the inspected factors by themselves only explain a small amount of the variance in completing courses or continuing to a subsequent course. The explored characteristics are insufficient to comprehensively understand what makes a learner to persist on the examined course set.
4. Creating Educational Content and Exercises

The second research theme of this dissertation is: Creating educational content and exercises. This chapter presents three publications that fall under this theme. First, Publication IV proposes the use of a large language model (LLM) to automatically generate programming exercises and code explanations. Then, Publication V and Publication VI explore the usefulness of LLM-generated code explanations in genuine educational settings.

4.1 Automatically Generating Programming Exercises and Explanations

In Publication IV we investigated the use of OpenAI Codex, an LLM trained on code for automatically generating programming exercises and code explanations.

Large language models are essentially text completion models that receive input (often referred to as “prompt”\(^1\)) and produce output that naturally should follow the input prompt (based on the model training data). Codex being trained on code, when given a description of desired behaviour for a function or program, Codex will most likely do its best to output source code according to the description. Likewise, when given some source code and some English to prime it for explaining, e.g. "Explanation:”, Codex will most likely output an English explanation of what the code does.

When given a full programming exercise with a description, model solution and automated tests, we found that Codex would start generating more exercises. Thus, in addition for its potential in explaining code, we saw Codex’s potential as a tool for automatically generating programming exercises — as a tool far more versatile than traditional template based solutions.

\(^1\)In the publication, we call the prompt “priming”.

57
We evaluated the capabilities of Codex for exercise creation and code explanation through answering the dissertation RQs 5 (To what extent are LLM-generated programming exercises sensible, novel, and readily applicable?) and 6 (How comprehensive and accurate are LLM natural language explanations of code solutions to introductory programming exercises?) OpenAI Codex is performant in multiple programming languages [33], yet reportedly works best in Python which it was originally trained on [36], and thus we chose to conduct our Codex experiments in Python.

4.1.1 Exercises

To assess Codex’s capability in generating programming exercises, we selected two priming exercises to include in the model prompt as examples of desired output. The first exercise was about checking for speeding (produce a function that prints out fines depending on parameter speed) and the second priming exercise was the creation of a currency converter class that uses a dictionary for exchange rates. We explored adding programming-related concepts (e.g. function, class) and contextual concepts (e.g. hiking, fishing) as keywords in the prompt to see how well Codex would accommodate them in its outputs.

In total, we generated 240 programming exercises with OpenAI Codex. We manually evaluated a random sample of 120 exercises according to their novelty (whether the generated exercise or a highly similar one was found online), their sensibility as a practice problem for students, whether the model solution matches the problem description, and whether the keyword concepts are accurately present in the output.

Our evaluation results show that the model was able to generate both novel and sensible exercises, sometimes of sufficient quality to be given to students readily without modification. As an example, the exercise in Listing 4.1 was generated by the model when primed with the contextual concepts ‘relationships’, programming concept ‘class’ and ‘list’, and the exercise about checking for speeding.

Listing 4.1. Example exercise created with keywords ‘relationships’, ‘class’ and ‘list’

| Write a class called Person that has a list of friends. |
| It has methods to add a friend and remove a friend. |
| Write a function called find_pals that takes a single parameter called person |
| and |
| that will list the friends of this person. |
| Use the Person class to create two persons and add friends to them. |
| Print out all friends of the first person. |

---

In the publication, the RQs are more explicit regarding the examined LLM, specifically mentioning OpenAI Codex.

58
Listing 4.2. Example solution to the exercise in Listing 4.1

```python
class Person():
    friends = []
    def add_friend(self, friend):
        self.friends.append(friend)
    def remove_friend(self, friend):
        self.friends.remove(friend)

def find_pals(person):
    return person.friends

pal_one = Person()
pal_two = Person()
pal_three = Person()

pal_one.add_friend(pal_two)
pal_one.add_friend(pal_three)

print("These are pal_one’s pals:")
for pal in find_pals(pal_one):
    print(pal)
```

We formulated the inputs to the model so that it would generate model solutions and automated tests in addition to the problem statements. The generated model solution for the exercise is shown in Listing 4.2. At times, however, the model would not generate tests at all. Such was the case for this particular exercise.

Another example shows the whole collection being generated. Listings 4.3, 4.4 and 4.5 show the generated problem statement, model solution and tests, respectively. The example was created with the currency converter class exercise and the keywords “class”, “dictionary” and “fishing”.

Listing 4.3. Example exercise created from currency converter with keywords ‘fishing’, ‘class’ and ‘dictionary’

Write a class called Fisherman that is initialized with a dictionary of fish, e.g., {“Salmon”: 10, “Trout”: 20, “Tuna”: 5}. The class should have a method called catch, which takes in one parameter, a fish name, and returns the number of that fish caught. The class should also have a method called throw_away, which takes in a fish name, and decreases the number of that fish in the dictionary by 1.

As an example, the code

```python
fisherman = Fisherman({'Salmon': 10, 'Trout': 20, 'Tuna': 5})
fisherman.catch('Salmon')
fisherman.throw_away('Trout')
fisherman.catch('Tuna')
print(fisherman.fish)  # should print out {'Salmon': 11, 'Trout': 19, 'Tuna': 6}
```
Creating Educational Content and Exercises

**Listing 4.4. Solution to the exercise in Listing 4.3**

```python
class Fisherman:
    def __init__(self, fish):
        self.fish = fish

    def catch(self, fish):
        if fish in self.fish:
            self.fish[fish] += 1
        else:
            self.fish[fish] = 1

    def throw_away(self, fish):
        if fish in self.fish:
            self.fish[fish] -= 1
        if self.fish[fish] == 0:
            del self.fish[fish]
```

**Listing 4.5. Test for the exercise in Listing 4.3**

```python
class TestFisherman(unittest.TestCase):
    def test_fisherman(self):
        fisherman = Fisherman({'Salmon': 10, 'Trout': 20, 'Tuna': 5})
        fisherman.catch('Salmon')
        fisherman.throw_away('Trout')
        fisherman.catch('Tuna')
        self.assertEqual(fisherman.fish, {'Salmon': 11, 'Tuna': 6})

    def test_fisherman2(self):
        fisherman = Fisherman({'Salmon': 10, 'Trout': 20, 'Tuna': 5})
        fisherman.catch('Salmon')
        fisherman.throw_away('Trout')
        fisherman.catch('Tuna')
        fisherman.catch('Tuna')
        self.assertEqual(fisherman.fish, {'Salmon': 11, 'Tuna': 7})
```

To summarise our quantitative evaluation, we found 75% of the generated exercises to be sensible, 81.8% to be novel, and that 76.7% had a matching sample solution. The priming concepts were accommodated well most of the time, ranging from 75.8% to 79.2% depending on the concept type. Programmatic analysis of all 240 programming exercises showed that 84.6% had a sample solution, 89.7% of which could be executed without errors. Automated tests were generated for 70.8% of the exercises, but only 30.9% of the sample solutions passed those tests. We note however, that the issues with tests not passing were often due to confusion between printing and returning values or incorrect test expectations. In some cases, only minor tweaks would have been needed to make the tests pass.

### 4.1.2 Code Explanations

Initially, we explored different types of primings to generate code explanations. We singled out three approaches that led to different types of explanations: 1) a high-level description (for example a single sentence or two that broadly describes the purpose of the code), 2) a problem statement-like description
Creating Educational Content and Exercises

(a description of the code that could be given as an exercise for students), and 3) a step-by-step explanation. We chose to focus only on the step-by-step explanation, as it aligns with the multistructural level of the SOLO taxonomy and is often produced by students when prompted to explain code [167].

We generated and jointly analysed five step-by-step code explanations for four different program codes, which total 20 explanations — we evaluated multiple explanations per program due to the undeterministic nature of Codex (and similar LLMs) [36]. The codes for generating the explanations can be found in Appendix A of the paper.

When evaluating the code explanations, we studied whether all parts of the code were explained, and whether each line was correctly explained. In the evaluation of 20 code explanations, 90% explained all parts of the code, but only 67.2% of the total 174 line-by-line explanations were correct. Most of the incorrect line-by-line explanations in the code explanations were related to comparison and branching conditionals. For instance, Codex produced a wrong explanation of “speed > 100” as “if speed is less than 100”. As an example of a line-by-line explanation generated by Codex, one for the Fishing exercise (presented above in Listing 4.3) is given below in listing 4.6.


We create a dictionary called fish where we’ll store our fishes and their quantity.
We create a class called Fisherman.
We create a constructor for the class Fisherman where we initialize the fish dictionary.
We create a method called catch which takes a fish as a parameter.
If the fish is already in the fish dictionary, we increment its value by 1.
If the fish is not in the fish dictionary, we add it and set its value to 1.
We create a method called throw_away which takes a fish as a parameter.
If the fish is in the fish dictionary, we decrement its value by 1.
If the fish’s value reaches 0, we delete the fish from the dictionary.

We also observed responses that were more abstract descriptions of the program codes. Four examples are shown in Table 4.1. Examples A and B show relational level responses to ‘speeding_check’ and ‘fizz_buzz’ codes. Example C includes some interesting background information on the FizzBuzz problem. Example D is non-sensical output that was generated in response to a program solving the extensively studied Rainfall problem.
Table 4.1. Examples of uncommon code explanations produced in response to the speeding_check (A) and FizzBuzz (B, C) problems. (D) is an example of a nonsensical explanation of the Rainfall problem.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>Takes the input of the speed of the car. Checks the speed and prints a fine according to the speed of the driver</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>is a simple program that checks if a number is divisible by 3, 5, or both.</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>This program is a variation of the FizzBuzz which is a popular coding challenge. It can be found here: <a href="https://blog.codinghorror.com/why-cant-programmers-program/">https://blog.codinghorror.com/why-cant-programmers-program/</a> The above program reads an array of integers, performs an if-else conditional check on the numbers and prints the result</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>Has many global and local variable lookup. Has an initial and final node. Has a number of variables, like variable names, that are used to access the values and are used to group the list of methods.</td>
</tr>
</tbody>
</table>

4.2 Evaluating LLM Generated Code Explanations With Learners

4.2.1 Experiences With GPT-3 Generated Code Explanations

In [Publication V](#) we generated code explanations with GPT-3 and embedded them into the online materials of a web software development course. We generated three types of code explanations to inject into the materials: (1) line-by-line explanations, (2) high-level summaries and (3) concept listings. Five explanations of each of the code explanation types were created for 13 different code snippets in the educational material, leading to a total of $13 \times 5 \times 3 = 195$ embedded GPT-3 explanations.

We chose GPT-3 over Codex for the study after evaluating both, and noticing overall higher quality with GPT-3 generated explanations. Three examples of the differences are shown in Figure 4.1.

The explanations were embedded into the course materials in the form of buttons — one for each explanation type. Students could click any of the buttons to view one of the five possible explanations within the chosen explanation category. It was also possible to repeatedly click the buttons to see all available explanations. Figure 4.2 shows how the buttons looked in the course platform for a code snippet that had GPT-3 explanations available.

Our main angle in this study was to investigate the readiness of GPT-3 generated explanations to be considered helpful for learners and to understand what kinds of generated explanations would be the most useful. Thus, the work addresses the dissertation RQ7 (What are the characteristics of LLM natural language explanations of code that students rate as most and least helpful?). The work also explores learner interactions with the explanations, addressing an additional two-fold RQ: How do students engage with the generated explanations,
<table>
<thead>
<tr>
<th><strong>Codex</strong></th>
<th><strong>GPT-3</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td># Summarize and explain the goal of the above code</td>
<td># Summarize and explain the goal of the above code</td>
</tr>
<tr>
<td>1. A web server is listening on port 7777. The handleRequest function is called when a request is accepted. What is the structure of the Deno module system? Deno supports ES Modules.</td>
<td>1. This code snippet creates a web server that listens on port 7777 and returns &quot;Hello world!&quot; for every request.</td>
</tr>
<tr>
<td>2. The goal of this code is to create a server that will count the number of times that the server receives a POST request. It will then return the number of times that the server has received the POST request.</td>
<td>2. This code snippet creates a server that listens for POST requests on port 7777. For each POST request received, the server increments a counter and returns the current value of the counter in the response.</td>
</tr>
<tr>
<td>3. The goal of the above code is to destructure the object obj. What is the value of first and second? first = &quot;hello&quot;, second = &quot;world!&quot;. What is the value of obj? obj = { first: &quot;hello&quot;, second: &quot;world!&quot; }; What is the value of { first, second }? { first, second } = { first: &quot;hello&quot;, second: &quot;world!&quot; };</td>
<td>3. This code snippet is using object destructuring to assign the values of the properties in the object &quot;obj&quot; to the variables &quot;first&quot; and &quot;second&quot;. It then logs the values of those variables to the console.</td>
</tr>
</tbody>
</table>

---

**Figure 4.1.** A comparison of the explanations generated by two large language models (GPT-3 and Codex) for three code snippets: (1) a 'hello world' server, (2) a server that counts POST requests, and (3) an example of object destructuring. The prompts used are shown. GPT-3 tended to produce more concise and consistently helpful explanations than Codex.

and which types of explanations do they use most frequently? For what types of code snippets do students request explanations?

Learners were presented with questions to provide subjective ratings on two questions picked from two sets: (1) either “the explanation matched the code” or “I knew what the code did before viewing the explanation.”, and (2) either “The explanation was useful for my learning.” or “The explanation was useful for me.” Figure 4.3 shows an opened explanation in the interface, and below the explanation, the question form that we used to get feedback on the explanations. In addition to learner feedback, we collected log data of how learners engaged with the AI generated explanations. Around half of the students (58 out of 116) viewed at least one explanation during three weeks. On average, students viewed 3 explanations and spent 52 seconds viewing an explanation.

The learners spent more time viewing longer explanations than shorter ones. The explanation that appeared first in the material was viewed the most — obviously because it was the first — but we also saw that explanations for more complex codes, especially ones at a later point in the course also gathered more views than the explanations for simpler ones. In addition, we observed that the student rated the explanations as less helpful when they already knew what the code was doing. These results point out that the students did actually interact with the explanation, most likely in a meaningful way.
Creating Educational Content and Exercises

Figure 4.2. The interface for viewing GPT-3 generated explanations.

Our key result, however, is that students rated the explanations as helpful to them ($\mu = 3.9$) and helpful to their learning ($\mu = 3.8$). The means are from 5-scale ratings between 1 and 5, meaning that ratings > 3 are considered at least somewhat helpful. Figure 4.4 illustrates the ratings given by students as boxplots for each explanation type and helpfulness category. From the plots, we can see that the line-by-line explanations received the lowest ratings while summary explanations were rated the highest with average ratings above 4.

Figure 4.4. Boxplot of explanation usefulness ratings with + indicating mean. Although most viewed among students, line-by-line explanations were rated least helpful.
4.2.2 Comparing GPT-3 and Learner Explanations

Publication VI presents another study we conducted on LLM explanations. Therein, we compare how learners rate 1) explanations written by their peers and 2) explanations generated by an LLM (GPT-3). In the work, we answer the dissertation RQ8 (To what extent do code explanations created by students and LLMs differ in accuracy, length, and understandability?) and also another RQ related to understanding what makes a good explanation (What aspects of code explanations do students value?).

We collected the data for the study from an introductory programming course in two iterations. First, we asked the learners to summarise and explain three different functions. Then, two weeks later, we asked the learners to evaluate a set of four explanations. The four explanations were randomly sampled for each learner from a set of explanations where half were by GPT-3, and the other half by the learners during the first iteration.

In the previous study, we asked students how helpful the GPT-3 explanation were. Here, instead, we measured the understandability, accuracy and length appropriateness of the explanations. We asked the students to rate each of the four explanation they received with respect to the following three questions.

Create a file called app.js and copy-paste the following content to the file.

```javascript
import { serve } from "https://deno.land/std@0.171.0/http/server.ts";

const handleRequest = (request) => {
    return new Response("Hello world!");
}

serve(handleRequest, { port: 7777 });
```

**Figure 4.3.** A line-by-line explanation generated by GPT-3 and the explanation feedback form.
- *This explanation is easy to understand* (5-items: Strongly disagree, Disagree, Neutral, Agree, Strongly agree)
- *This explanation is an accurate summary of the code* (5-items: Strongly disagree, Disagree, Neutral, Agree, Strongly agree)
- *This explanation is the ideal length* (5-items: Much too short, A little too short, Ideal, A little too long, Much too long)

On average, students rated the GPT-3 explanations as better than the explanations of their peers in all categories but length. The GPT-3 averages were 4.12 (easy to understand), 4.0 (accurate summary) and 2.66 (ideal length) while the student explanation averages were 3.75 (easy to understand), 3.78 (accurate summary) and 2.75 (ideal length). Figure 4.5 shows the overall distribution of the responses for understandability and summary accuracy.

![Figure 4.5. Distribution of student responses on LLM and student-generated code explanations being easy to understand and accurate summaries of code.](image)

We further investigated the statistical significance of difference in the ratings of GPT-3 and peer explanations with Mann-Whitney U tests. We found the differences in both ease in understanding and accuracy of the summary to be statistically significant, albeit with only a small effect size. The test for ideal length didn’t indicate a statistically significant difference.

Next, to illustrate the types of explanations that GPT-3 and students created, shown below are four example explanations that differ in terms of student-perceived quality — the highest- and lowest-rated LLM- and student-created explanations for the LargestValue function. The examples were ranked based on the average over accuracy and understandability.

The following LLM-generated explanation was rated the highest (understandability = 4.2, accuracy = 4.5, length = 3.5):

“This code snippet finds the largest value in an integer array. It starts by assuming that the first value in the array is the largest. It then loops through the remaining values in the array, comparing each value to the max. If a value is larger than the max, it becomes the new max. This process is repeated until all values in the array have been checked. The largest value is then returned.”
The following LLM-generated explanation was rated the lowest (understandability = 4.1, accuracy = 3.7, length = 2.0):

“This code snippet returns the largest value in an array of integers.”

The following student-created explanation received the highest evaluations (understandability = 4.2, accuracy = 4.5, length = 3.6):

“The function LargestValue, finds the largest integer value in the input array, and outputs that value as type int. It takes two inputs:

* values - an integer array
* length - the length of that array (number of elements inside the array)

It sets the first integer value in the array as the original ‘max’ value (largest integer in the array) and then scans through the input array (left to right), checking to find if there is a larger integer value, which if found it stores in it’s output integer variable - ‘max’, which is returned at the end of the function.”

The following student-generated explanation received the lowest ratings (understandability = 3.3, accuracy = 3.3, length = 1.6):

“to find the largest value within the array”

Students preferred explanations that had both information related to what the function is and how it operates being described in a line-by-line format. Many students agreed that a good explanation explains the inputs and outputs of the code. Bad explanations were characterized as missing some details of the code while either being too long or too short. For example, an explanation may state at a high level the purpose of the code, but not go into detail about what data structures were used, or what inputs are given to the function.

Interestingly, we found that all of the LLM-generated explanations started out with the statement “This code snippet” or “The purpose of this code snippet” while the student generated explanations differed more. This was partially due to the prompting of the LLM, where it was asked to explain the purpose of “the following code snippet”. However, most of the explanations by both students and the LLM generally followed a similar structure: function’s purpose, analysis of the code, and finally the return output.
5. Towards More Timely Feedback

The third research theme of this dissertation is: *Towards more timely feedback.* This chapter presents two publications related to the theme. First, [Publication VII] proposes a framework for speeding up automated code assessment. Then, as a step towards timely automated feedback interventions, [Publication VIII] explores collecting data of when, how and why experts intervene on learners struggling in the process of solving programming exercises.

5.1 Speeding Up Automated Code Assessment

In [Publication VII] we propose a framework for speeding up automated code assessment. We note that traditionally, assessment of source code is done by running the code against instructor-created unit tests or tests that evaluate program input and output. While this is an effective way to provide automated assessments, it can also be resource intensive and cause unnecessary delay in feedback, especially in large MOOC courses with frequent — possibly simultaneous — code submissions, or when submissions contain infinite loops or are otherwise inefficient to a fault. Running the code is certainly not always a necessity for evaluating code correctness; other methods like static analysis could already identify errors in the code.

Our framework is based on the following idea: Rather than running potentially resource and time intensive unit tests for each learner code submission, submitted source codes should first go through a series of checking phases (or steps) that involve both direct static analysis of code and machine learning methods. When submitted code does not pass any one of the phases, feedback is given immediately and further steps can be omitted.

We formulate the framework as follows: (1) First, when a student submits an exercise for evaluation, a cache containing feedback given previously in the course for similar exercises is checked. If a match is found, feedback can be given instantly from the cache; (2) Second, if there are no direct matches in the cache, static analysis of the student program is conducted to detect potential errors; (3) Third, if the student’s program passes the previous checks, machine
learning models that utilise features based on static analysis can be used to estimate program correctness; and (4) if all of the previous checks pass, only then we actually run the program against instructor-created unit tests. The framework and its four evaluation steps is also illustrated in Figure 5.1.

Figure 5.1. Our proposed evaluation sequence. Each step in the sequence is the final step if a match or problems are found in the step. 1. if the solution is in the cache, return cached test results; 2. if static analysis finds errors, return those; 3. if the ML-model is rather certain that there is a problem, return feedback from the model; 4. execute unit tests and return their results.

We intend the framework to be flexible and adaptable to different contexts, wherefore we describe the assessment phases at an abstract level. The first step (cache) can be implemented for instance using standard caching techniques such as LRU caching [44]. Detecting cache hits can be adjusted based on contextual needs: for example, in one context we might want to utilise string-based similarity metrics, while in another we might be happy with AST-based similarity.

The second step (static analysis) can encompass any type of static analysis, for example, identifying common syntactic errors and rule-based identification of infinite loops. Which static analysis approaches are utilised can depend on the programming language used and the needs of the specific context.

The third step (machine learning) relies on machine learning models that have been trained with static analysis based features using previously submitted student answers. What the models try to detect can again be adjusted based on the context: for example, in one context we might want to train a model to predict answer correctness, while in another we might be more keen to predict the presence of common misconceptions. A noteworthy aspect of the third step is the possibility of the machine learning model giving an incorrect prediction. Even with a highly capable model, the danger of inaccurate assessment should be mitigated by giving students an option to proceed to the fourth step if they disagree with the model’s prediction (or the feedback based on the prediction). Another important consideration here is that the third step is not about predicting the absence of errors in learners’ programs, but the presence or errors. When the models find no errors, we continue to the next assessment step.

The fourth final step (running tests) is the traditional test based automatic assessment approach and can further give “ground truth” about the correctness of the program. This is the most resource intensive step (considering only
resources used for each individual submitted program) by a large margin — as evidenced in our framework evaluation below.

To evaluate the efficiency of our framework in speeding up the assessment process, we implemented a very basic prototype of the framework, and conducted a post hoc analysis of how the prototype would perform compared to merely using unit tests. The data used in the evaluation comes from the first course in the programming course series for lifelong learners that was studied in [Publication III]. The dataset at the time of this study contained 54,904 programming exercise submissions to a total of 64 exercises from a total of 725 learners. Of the submissions, 24,649 pass the tests, meaning that approximately 55% of the submissions are at least partially incorrect. In the evaluation, we address dissertation RQ9 (How much can simple ML models speed up programming assessment in an assessment framework consisting of cache, static analysis, ML and program execution checks?), or more precisely, the following three research questions found in the publication: (1) “How often can exercise solution results be retrieved from cache?”, (2) “How often can static analysis find errors in code submissions?”, and (3) “How often can simple machine learning models reliably capture erroneous code submissions?”.

For phase one of the framework (cache), we explored exact string matching and AST matching. For phase two (static analysis), we implemented two static analysis checks for two common problems in novice codes: infinite loop[1] and a certain common problem involving an extra semicolon (the description and an example of the semicolon problem can be found in the publication). For phase three (machine learning), we trained a logistic regression model with two concatenated BoWs (Bag-of-Words) formed from source code ASTs as the model inputs. The BoWs we used are AST token BoW and AST type label BoW. When evaluating the phases two and three, we excluded any submissions caught by the previous phases to avoid biasing the results.

In our evaluation, we found that cache checking could catch roughly one third of submissions when using exact matching and 57% when using AST matching. Simple static analysis to catch the two common errors caught up to 18% of the unique submissions depending on the problem and 4% on average — since the static checks target specific problems, they are not applicable to many of the exercises in the analysed data. The logistic regression model was able to catch 10% of the remaining problems (in test data) as erroneous when setting the decision threshold of the model prediction to achieve a high precision of 99% (model predicts errors being present when there are none once every 100 submissions) and roughly 70% when aiming for a lower precision of 95% (model incorrectly predicts a submission as erroneous once every 20 submissions). The overall effectiveness of the three steps prior to unit tests is illustrated with a sankey diagram in Figure 5.2.

[1]We did not solve the halting problem. We only use regular expressions to identify very basic cases.
Towards More Timely Feedback

Figure 5.2. Sankey plot that shows the proportion of (incorrect) submissions captured at each step of the approach. Precision threshold of 95% is used for the machine learning (ML) model.

Overall, the framework caught around 80% of submissions before running the automated tests which potentially take long time to execute consuming computational resources and leaving students waiting. This amounts to saving nearly the same amount of processing time as the first three steps could be computed in a fraction of the time it takes to run unit tests (~ 10ms against ~ 2 to 30 seconds in our experiment data).

5.2 When, Why, and How to Give Feedback During the Programming Process

Publication VIII is the report by an ITiCSE working group in 2022 where we focused on two topics revolving around programming feedback: (1) how should datasets consisting of steps learners take towards solving a programming task be annotated with information about when, why, and how experts would give feedback and hints; and (2) how does expert feedback relate to the feedback found in learning environments for programming. The research questions addressed in the publication are the final two dissertation RQs: RQ10 (How should we annotate datasets consisting of steps students take towards solving a programming task with information about when to give feedback and hints?) and RQ11 (How does expert feedback relate to the feedback found in learning environments for programming?).

In the working group, we formulated guidelines for creating such an annotated dataset. These are shown in Table 5.1. We then proceeded to annotate two datasets containing fine-grained steps of learners working on programs using the methodology. For the second topic under focus, we evaluated a large body of learning environments for programming, and classified them by their feedback characteristics. Our results are shown in Table 5.2 and the legend for the classification is laid out in 5.3.

Then, we compared how the learning environments provide feedback with how experts would prefer to provide feedback. The two datasets that were selected for annotations both contained fine-grained steps of learners working
on programs. The first dataset contained edits at keystroke-level that were aggregated into token-level edits for annotation — arguably it would not be meaningful to interrupt learners with feedback while they are in the middle of writing a word. The second dataset contained edits at the token-level.

We utilised a two-step process to identify intervention points when annotating the datasets. In step one, experts examined a small set of logs, discussed potential moments to intervene and formulated a preliminary set of guidelines for situations warranting intervention. In step two, the experts employed these guidelines to annotate sequences from two datasets that were selected for the study. This step encompassed multiple rounds of refinement, validations, and discussions among the experts — the details are elaborated in the paper.

Our key finding is that experts aim to provide timely feedback and intervene on the students problem solving process, sometimes near instantly when the expert can affirm a student starts going towards a problematic direction or appears stuck. On the other hand, the evaluated learning environments are more passive and provide feedback mainly when students request it or when they submit a solution. The working group’s examination of learning platforms also shows that the provided feedback is primarily focused on symptoms and does not focus on the cause of errors.

Inspecting the annotated edit sequences reveals an interesting phenomenon. In some of the sequences, students tended to abandon their approach completely, only to retry until reaching the exact same problem and starting over yet again, instead of for instance asking for help on the first or second time they got stuck. This observation underlines the potential of interruptive feedback mechanisms that guide learners towards correct paths and to think what they are doing wrong (and what correctly).
### Table 5.1. Guidelines that identify when and how to intervene.

<table>
<thead>
<tr>
<th>Event</th>
<th>Example</th>
<th>Intervention Point (when)</th>
<th>Intervention Action (How)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compiler error</strong></td>
<td>Syntax or type error</td>
<td>If a student is not using the compiler often, or after the second compilation where the error goes unaddressed.</td>
<td>First, suggest that they compile if they have not done so. Second, offer an explanation.</td>
</tr>
<tr>
<td><strong>Semantic error</strong></td>
<td>Syntax is correct but wrong semantics, e.g. “=” instead of “==”</td>
<td>When a student moves to the next line, as these errors are hard to spot/debug.</td>
<td>Highlight the location of the error and offer an explanation.</td>
</tr>
<tr>
<td><strong>Logical error</strong></td>
<td></td>
<td>Once a student executes/tests the code or within 5 minutes if they choose not to run the code. If the error would lead to any further code being incorrect, intervene immediately.</td>
<td>Indicate the test case(s) that fail due to the error and provide a hint on how to fix it.</td>
</tr>
<tr>
<td><strong>Deviation from specification</strong></td>
<td>Changing required function signature</td>
<td>When a student leaves the line.</td>
<td>Provide clear statements from the assignment description as a reminder of the assignment specifications.</td>
</tr>
<tr>
<td><strong>Trial and Error behaviour</strong></td>
<td>Iterating through conditional operands</td>
<td>Once it becomes clear the edits are guessing – not experimentation.</td>
<td>Ask a student a question (e.g. an MCQ) about the purpose of the line. If they respond with a correct answer, provide a hint. Otherwise, suggest a (sub)goal to complete.</td>
</tr>
<tr>
<td><strong>Hint or Feedback request</strong></td>
<td>Pressing a “Hint” or “Show solution” button</td>
<td>Immediately when a student requests assistance.</td>
<td>A hint depends on the time a student requests it. If a student has a semantic/syntax error and asks for a hint, then offer a clear hint on how to fix the error. If a student has a logical error or is stuck in a specific subgoal, then offer a clear hint on how the subgoal/objective can be reached.</td>
</tr>
<tr>
<td><strong>Subgoal completion</strong></td>
<td>Correct base case(s) for recursive function</td>
<td>When a student completes all steps of a subgoal.</td>
<td>Provide positive feedback specific to the accomplishment.</td>
</tr>
</tbody>
</table>
Table 5.2. Feedback characteristics of learning environments related to knowledge about mistakes.

<table>
<thead>
<tr>
<th>Name</th>
<th>TF</th>
<th>CE</th>
<th>SE</th>
<th>SI</th>
<th>PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codewars</td>
<td>r → ●</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codecademy</td>
<td>r → ●</td>
<td>sl → ○</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coderbyte</td>
<td>r → ●</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodingBat</td>
<td>r → ●</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataCamp</td>
<td>r → ●</td>
<td>r, sb → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FITech 101</td>
<td>sb → ●</td>
<td>p, r → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FreeCodeCamp</td>
<td>r → ●</td>
<td>p → ●</td>
<td>r → ●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Funprogramming</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GATE</td>
<td>sb → *</td>
<td>sb → ●</td>
<td>sb → ●</td>
<td>sb → ○</td>
<td>sb → ○</td>
</tr>
<tr>
<td>HackInScience</td>
<td>sb → *</td>
<td>sb → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JACK</td>
<td>sb → *</td>
<td>sb → ●</td>
<td>sb → ●</td>
<td>sb → ○/ *</td>
<td></td>
</tr>
<tr>
<td>Kaggie</td>
<td>r → ●</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kaggle</td>
<td>r → ○</td>
<td>sb → ○</td>
<td>sb → ○</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Khan Academy</td>
<td>p → ○, h → *</td>
<td>p → ●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEARNJS/Java/C</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEARNPython</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python Tutor</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3Schools</td>
<td>r → ●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3. Legend for feedback coding.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
<td>Test failures</td>
</tr>
<tr>
<td>CE</td>
<td>Compiler errors</td>
</tr>
<tr>
<td>SE</td>
<td>Solution errors</td>
</tr>
<tr>
<td>SI</td>
<td>Style issues</td>
</tr>
<tr>
<td>PI</td>
<td>Performance issues</td>
</tr>
<tr>
<td>p</td>
<td>Code</td>
</tr>
<tr>
<td>r</td>
<td>Run</td>
</tr>
<tr>
<td>c</td>
<td>Compile</td>
</tr>
<tr>
<td>sb</td>
<td>Submit</td>
</tr>
<tr>
<td>h</td>
<td>Ask for hint</td>
</tr>
<tr>
<td>sl</td>
<td>Ask for solution</td>
</tr>
<tr>
<td>f</td>
<td>Function call</td>
</tr>
<tr>
<td>→</td>
<td>leads to</td>
</tr>
<tr>
<td>○</td>
<td>basic feedback</td>
</tr>
<tr>
<td>●</td>
<td>detailed feedback</td>
</tr>
<tr>
<td>*</td>
<td>enhanced/extended feedback</td>
</tr>
</tbody>
</table>
Part III

Discussion and Conclusions
6. Discussion

The discussion of the results is divided into three sections according to the three research themes of this dissertation, following a structure similar to that of the result chapters.

The first section discusses the results and implications for the three studies [Publication I, Publication II, Publication III] that relate to identifying learners who would benefit from extra help or practice through machine learning methods. The second section discusses how machine learning methods — LLMs in particular — can be used to help educators produce useful learning materials and exercises, which is explored in the publications [Publication IV, Publication V, and Publication VI] and the implications that may arise from leveraging these methods in practice. The final section focuses on feedback with particular emphasis on the timeliness feedback, discussing the results and implications of the final two publications of this dissertation, [Publication VII and Publication VIII].

6.1 Discussion on RT1

Identifying learners in need of additional help or practice

6.1.1 Dropout Prediction and Study Design

High dropout is a problem that is particularly present in MOOCs where the majority of participants drop out before completing the course [22, 198]. As a means to alleviate the problem, machine learning models that predict who is likely to drop out have been extensively studied [75, 231], and also employed in early warning systems to help educators focus their attention on at-risk students or alert students themselves of being at risk of dropping out [172]. [Publication I] shows how building a predictive model that includes data from students who have already dropped out may lead to inflated model performance through leveraging features that are not useful or present in practical scenarios. Relying on such a model may hamper efforts where data
from one course is used to predict the outcomes in another course, or lead educators astray in understanding what are the actual reasons for dropping out. Transfer learning (see e.g. [142]) may be a way to alleviate reliance on models that emphasize impractical features, but even then the situation remains suboptimal.

When considering the problem of students dropping out in general, we acknowledge that there are a multitude of issues at play. While losing interest or preferring other work, highlighted e.g. in [137, 222], might be something that cannot be addressed, other factors — also highlighted in [137, 222] — such as plagiarism or excessively relying on others could be alleviated through interventions. It is a good question, however, what kinds of features should be used to identify factors such as confusion and the actions taken in such situations, as discussed in [162].

We also note some methodological aspects that we overlooked as they were not the focus of our study. First, we defined dropout through inactivity, but students who drop out late might still get a passing grade. Second, despite our data showing that very few students become active after becoming inactive, this is still a possibility and a corner case, which the approach where already dropped out learners are included could handle — despite its flaws — better. In our defense however, in practice, the relative number of students who drop out of courses tends to decrease the further we go into the course [198]. Third, as in many evaluations of predicting dropouts, we used data from the same course, which itself could lead to bias in the data. Fourth, we did not look into demographics or more fine-grained features and thus, cannot state how the two approaches would compare in those situations. Lastly, the feature analysis we performed in the study is by no means extensive but rather limited.

6.1.2 Deep Learning Models for Knowledge Tracing

Touching the aspect of helping those in need of extra practice, in Publication II we conducted an empirical evaluation of (deep learning for knowledge tracing) DLKT models which have dominated the area of knowledge tracing in recent years, pushing the limits of the state-of-the-art further.

When considering the evolution of the knowledge tracing field, our evaluation and recent other evaluations of DLKT models [91, 181, 208] show that indeed, the introduction of DLKT to the field has clearly advanced the overall performance of knowledge tracing. The move from simpler models to deep learning models has shown robust and verified improvements in knowledge tracing performance. Although, as noted in Publication II as well as in [91], DLKT models are not a silver-bullet (at least not the tested ones) as much simpler models have been shown to achieve comparable or even better performance than the new state-of-the-art DLKT models.

In addition, in our empirical evaluation of DLKT models, we identified multiple methodological aspects that can affect the performance and the
ranking of different DLKT models, some of which are easily overlooked. For example, we saw the choices of maximum attempt count and even machine learning framework version resulting in noticeable performance differences (respectively; over 2% and 0.5% points ROC-AUC, or over 0.5% and 0.2% RMSE). We note that a meaningful performance difference need not be particularly large. As an example, Koedinger et al. [139] have demonstrated that even a 0.003 (0.3%) RMSE improvement for a knowledge model can lead to more effective learning when put to practice in learning environments.

These discoveries, along with the potentially large effect of the more commonly emphasized hyperparameters (by which deep learning models in particular are influenced), highlight the importance of both careful hyperparameter tuning and interpretation of small improvements in performance. If small differences in model performances are due to slightly more optimal hyperparameters or even non-model related attributes such as hardware used in training, it may be more fruitful to concentrate more on hyperparameter tuning rather than selecting the “best” out of a multitude of different KT models. As an example, our results showed that DLKT models in general outperform older types of models, especially BKT, but Khajah et al. [130] have shown that BKT too can achieve performance comparable with DLKT models when manually fine-tuned.

Similarly, our comparison of priorly reported results showed inconsistencies in model performances and ranking, further highlighting the problematic nature of minor improvements in performance. To be precise, we do not mean to downplay the well argued importance of minor improvements in student models [139, 170, 218], rather we want to underline the importance of extensive replication work for determining the cause and consistency of reported improvements. Another reason for highlighting the importance of replication is that of determining most lucrative avenues of improvement for the versatile DLKT models — which are not showing any signs of shortage, rather, the situation appears to be the opposite.

Multiple recent DLKT approaches show promise (e.g. [196, 169, 92, 42, 38, 209, 202, 263, 269]) with many claiming significant performance improvements compared to prior models. Many of these newer models include slightly different inputs, such as skill and previous correctness as separate inputs [42], additional time related inputs [263] or leveraging both exercise and skill labels [269], giving rise to the question of whether and how much the older architectures would also benefit from such input additions.

We see promise in exploring new inputs since DLKT models are powerful models that are likely to benefit from such additional information. Creating new models by introducing new types of inputs is hardly a new direction however, as it has been witnessed much earlier in the KT field. As an example, Performance Factors Analysis [213] is an improvement over Learning Factors Analysis [34] that includes student ability as a separate input, achieving better performance. Similarly, a variety of ways to improve BKT models by
Discussion

adding new inputs have been proposed \cite{130}, e.g., adding item difficulty as an input \cite{212}. However, to the benefit of DLKT models, as suggested by Khajah et al. \cite{130}, deep learning models can leverage regularities in the data that prior models cannot (without manually adding such capability). As the deep learning models themselves are capable of performing feature engineering \cite{144}, they provide fertile ground for evaluating the effect of adding new kinds of input features for learner knowledge modeling.

On the other hand, one of the challenges of the introduction of DLKT models is the decrease in interpretability. DLKT models, with their complex layered structures, often resemble black boxes, making them challenging to interpret \cite{190}. Techniques like feature visualization and concept detection can provide some insight to DLKT model behaviour, but they are still heuristic approaches to gauge feature importance. In contrast, BKT and logistic regression models offer more transparent interpretation. For instance, BKT (with manual fine-tuning) \cite{130} and logistic regression models \cite{91} can provide performance comparable to DLKT models while retaining clear insight into their workings. A potential danger with the adoption of DLKT models is a disconnect between the use of learning theories and knowledge tracing. One could even see a link between using deep learning for knowledge tracing and using machine learning for natural language processing, where one of the famous quotes is “Every time I fire a linguist, the performance of the speech recognizer goes up” (Frederick Jelinek in the 1980s). Unless we understand the inner workings of these models and what their predictions are based on, it will be difficult to grasp why and when certain models outperform others.

6.1.3 Mind Your Metrics and Baselines

Evaluation metrics are essential in judging model performance and ranking models based on their ‘goodness’. A multitude of metrics for model evaluation have been proposed over time, and there is no one metric to rule them all: relying on a single metric can easily lead to misguided judgement of performance \cite{197}. The usefulness of one metric over another depends on the task at hand, and as Gunawardana et al. \cite{96} state, “The decision on the proper evaluation metric is often critical, as each metric may favor a different algorithm”. For instance, in identifying at-risk students, recall can be considered preferable over precision, since it can be argued that finding struggling students is preferable over finding non-struggling students. Misidentifying a well-performing student as a struggling student is not as costly as vice-versa, as the downside is that a well-performing student might be offered additional support that they do not need. Thus, in this case, tuning precision and recall in favor of recall might be more favorable than tuning precision and recall in favor of precision or with equal weights.

In a survey of student dropout prediction in online courses, Prenkaj et al. \cite{231}, describe a number of metrics that have been used in predicting at-risk
students. The listed metrics include for instance ROC-AUC, accuracy, MSE, and MAE which are common also in knowledge tracing model evaluation (see e.g. [51]). The authors note that "given the unbalanced nature of the data, only a fraction of these methods can be used", referring to the fact that the number of persisters often differs greatly from the number of dropouts and such skew in data affects certain metrics. They point out for instance accuracy as a metric that is likely misleading when used alone, due to its nature of being influenced by unbalanced data. However, we argue that accuracy becomes much less misleading when used in conjunction with comparison to simple baselines such as majority vote to account for the effect of data skew on the metric.

When considering the metrics that we used for studying the performance of the dropout prediction models in Publication I, we observed considerable differences between the excluding (learners who have already dropped out are not considered when predicting future dropouts) and including (learners who have already dropped out are considered when predicting future dropouts) approach in most of the metrics in most of the courses. These differences can be seen in Figure 2 in the paper. We note that accuracy tended to be a poor metric by itself due to the imbalanced dataset, and also that the ROC-AUC masked performance differences for CS1-B and CS0-Web. The performance differences were clearly visible, e.g. when studying F1 score or accuracy against simple baseline models.

In defence of the ROC-AUC metric, which is popular also in domains other than education (e.g. medicine [136,111] and natural language processing [204]), some formal and quantitative studies have shown ROC-AUC to be consistent with and more discriminative than accuracy [166,110,100]. Other studies, however, are aligned with our results, showing that there is no guarantee that better ROC-AUC translates to better model [119,203,68]. Notably, area under precision-recall curve (PR-AUC) has been proposed as a reliable metric that is less prone to mask poor performance in highly imbalanced data settings [55,203,249,231]. This is despite PR-AUC also being affected by data skew similarly to accuracy [289].

All of this can make choosing the metrics to determine and report model rankings difficult. As an example of a metric choice dilemma, in our results in Publication II for the ASSISTments 2015 dataset, DLKT models hold the best ROC-AUC and MCC scores by a significant margin. On the other hand, both the GLR and BKT models achieve almost the same performance on most other metrics. Also, the simple mean baseline is not far from the DLKT models in terms of accuracy, and the mean baseline holds the best F1-score tied with GLR. The DLKT models outperform GLR and BKT only on MCC and ROC-AUC which are the metrics often presented as alternatives to the "misleading" accuracy and other metrics that can be influenced by data skew.

In a related study, Effenberger et al. [78] evaluated the metrics MAE (mean absolute error) and RMSE (root mean squared error) in detail for student modeling. Similarly to our case, they reported situations where the choice of
metric affected model ranking, and they also drew attention to the possibility of picking a “suitable” metric for a newly proposed model to make it appear better in comparison to previous models. In addition, they showed that besides metric choice, the computation methodology (RMSE over whole data vs average over RMSEs per student data) of the metric may also affect model ranking.

We note that choosing a metric is not solely a problem in reporting, but also in hyperparameter tuning. In the hyperparameter tuning phase of our DLKT study, we observed that the set of optimal hyperparameters was dependent on the metric that was used to tune the hyperparameters. Consequently, the model with the best hyperparameters according to one metric may not be optimal when considering other metrics. This problem is also noted by Sanyal et al. [250], who inspected how optimizing feature selection on different metrics influenced model performance and observed that metric selection can lead to large performance differences between datasets and selected features.

This further highlights the importance of choosing and understanding metrics for model comparison as well as the importance of using multiple evaluation metrics. Our suggestion is to always provide multiple metrics for evaluation in addition to at least one metric serving as a primary metric for model ranking. Having at least one metric unaffected by skew (e.g. ROC-AUC, MCC), one affected by skew (e.g. Accuracy, F1-score) as proposed in [119], and also at least one generic error metric (RMSE, MAE) as proposed in [168], should be included for reliable model performance interpretation. ROC-AUC is often used as the main metric for evaluation; however, PR-AUC could be a potentially better choice due to it being less prone to mask poor performance as suggested e.g. by Prenkaj et al. [231].

Lastly, simple baselines such as majority vote and mean would be best not forgotten even when much ‘better’ (performant) baselines are available. These baselines provide useful insight when interpreting model goodness with data imbalance sensitive metrics such as accuracy of F1-score. These stupidly naive baselines (or similar) have the practical property of not being useful in practice, thus serving as good indicators when a more complex model is unable to make good sense of its data. When performance of a machine learning model is in the same range as a naive baseline, its utility in practice deserves to be questioned.

6.1.4 Who continues in MOOCs?

Our analysis of how learners continue in a series of open online courses in [Publication III] lets us observe an unfortunate dropout phenomenon seen in other open online courses [244 295], despite the course series being targeted and built for lifelong learners. It is possible that the high dropout in the course series relates to too difficult exercises when considering views that posit that university-level introductory programming courses expect too much from
their students [176]. Yet at the same time, it is also possible that many of the learners are there to simply check the topic out, and that difficulty of the topic is not the reason why learners drop out.

The most prominent results in our learner characteristic examination for retention is that those with previous experience are more likely to complete and continue in our lifelong learning course set. Even though most of our starting learners have little experience, this finding alongside that most of the participants already have some tertiary education, backs the notion raised by Reich et al. [242] that “MOOCs are primarily a complementary asset for learners within existing systems”.

Regarding the traditional split of intrinsic and extrinsic motivation, our results show somewhat mixed results. While our only clearly intrinsic motivational factor (interested in topic) overall appears to be a positive factor, the various external motivations show large differences. Especially, wanting to complete a university course, which can be seen as a strong external motivation, is a much more positive factor for introductory course completion than intrinsic motivation[1] — we note however, that intrinsic motivation has not consistently been linked with programming course completion in university contexts either [259, 253].

Another point that arises from our study is that motivation, previous experience, education and demographics were not able to explain a large portion of the differences in completion rates. From the perspective of inclusivity, this is a positive finding; if gender and prior experience would be enough to explain variance in retention, it would suggest that the courses are very much tailored for people that have specific backgrounds. On the other hand, more information is required to understand who are less likely to drop out and why.

6.2 Discussion on RT2

Creating educational content and exercises

6.2.1 Pioneering Automated Programming Exercise Generation with Large Language Models

In [Publication IV] we proposed using a large language model, namely OpenAI’s Codex, for automatically generating programming exercises. We formulated a one-shot prompting approach that allowed us to generate a problem description, model solution code, and test code for specific themes and programming concepts all at once. In our evaluation of the approach, we discovered highly promising results. Most of the generated exercises were sensible (as in the

---

1It is possible that some of the participants are students at universities that allow including credits from the courses to their degree.
Discussion

Figure 6.1. An example of generating a hiking themed exercise for practicing loops with OpenAI’s ChatGPT (GPT-3.5). The image is from ChatGPT’s online user interface and taken in March 2023.

description depicting a sensible programming problem for students to solve) and novel, some even ready to be given to students as is.

We note, however, that the capabilities of the Codex model, while already practically useful, were far from ideal: out of the evaluated 120 Codex outputs, only 75% of the generated outputs were deemed having a sensible problem description, only 77% had a model solution that matched the problem description, and only 70% included tests, out of which only 31% matched the model solution (meaning that the model solution passed all the tests).

Despite most of the generated exercises being unsatisfactory at least in some aspect, our results are hardly discouraging. Upon manual inspection, many of the faults in the generated content, such as errors in model solution, were minor and easily fixable with little effort. Further, the Codex model used in our study is quite outdated due to the speed of advancement in the field, and has been superseded by for example the newer and much more capable OpenAI models GPT-3.5 and GPT-4 [272, 125]. As an example, Figure 6.1 shows an exercise generated with ChatGPT (GPT-3.5) a zero-shot learning approach of providing merely a short sentence description of a desired exercise.

As LLMs are improving fast and already highly capable [28, 251, 223], to the extent that even providing exercises generated on the fly for students could be a valid approach at some point in the near future. Regardless, human oversight will likely be relevant for the foreseeable future for optimal learning gains, especially if relying on foundation LLMs. For example, Phung et al. [223] show that the latest top model, GPT-4, remains far behind human tutor performance in generating exercises targeted for extra practice for specific concepts or bugs.

Nonetheless, even faulty exercises can greatly speed up the process of creating exercises for educators by giving worthwhile ideas and drafts to improve upon. This is particularly true for topics with which the educators may be unfamiliar with, enabling a more varied and personalised exercise palette than otherwise feasible. As anecdotal evidence, the author of this
dissertation has used LLM-based tools such as ChatGPT and GitHub Copilot successfully for speeding up learning material and exercise creation for a programming course.

Another method for content generation, learnersourcing [135], where students participate in the creation and evaluation of course materials such as questions and exercises, has become increasingly common in computing education (see e.g. [57, 61, 225, 153]). For example, the Quizius tool described by Saarinen et al. has students contribute questions to a repository, and their answers are used to produce statistical estimates of the prevalence of topic misconceptions [248].

We envision a new type of learnersourcing we coin “robosourcing”, where machine learning models are used to automatically create artefacts similar to traditional crowdsourcing, but where these “robosourced” learning materials are then evaluated by students. This would address one of the major challenges related to the use of learnersourcing, which is that students tend to be much more inclined to use and evaluate resources created by others than they are to create resources themselves [267, 226].

This method holds promise in improving both the quality of learnersourced content as well as its volume — learners can generate content faster and with less effort with the help of AI tools (this alone may improve quality as happy workers are more likely to work with rigor) and the content will be curated by both humans and AI. LLMs (or possibly completely new machine learning models) also have more potential for quality improvement compared to learners. Machine learning methods are constantly evolving and at a fast pace, learners much less so. Nonetheless, we advocate human supervision when creating learning artefacts to minimise the risks of automatically generated potentially misleading materials, not least because LLM outputs have been shown to portray the same biases as humans [254]. Building upon previous works and taking a step forward in this regard, Khosravi et al. [134] have introduced a formal framework for learnersourcing enhanced through human-AI partnership. The authors go into detail, demonstrating how the benefits of both imperfect worlds can be effectively leveraged through two case studies.

6.2.2 LLMs as Shortcuts for Learning

The possibility to create model solutions on the fly with LLMs for programming [86, 87, 252, 201] (and other [73, 41]) exercises is not entirely a positive thing. Becker et al. [19] note that the AI-generated solutions offer a low-risk/high-reward avenue for focusing on short term grades rather than deep understanding. This is especially concerning as already prior to the emergence of GPT and the like, contract cheating (outsourcing assignment work to others) has been noted to be on the rise [265] — and students are already openly sharing tips on how to fool AI detectors in social media [99].

Naturally, this problem is more pronounced when considering AI-generated
exercises. Assuming we are able to generate a problem description and a matching model solution (or several) for an exercise with public LLMs, we can of course also generate just the solution when we come across the problem description. It would be naïve to believe that learners who are tempted to take shortcuts in their learning path would not figure this out quickly.

However, on a positive note, the introduction of high-quality auto-generated exercises also carries a chance to help reduce the incentive for cheating. It has been long noted that self-efficacy is an important factor affecting inclination to cheat [188,194], and in a review of academic integrity in e-learning, Holden et al. [106] list studies that portray feeling incompetent or not appreciating one’s level of mastery as causes for cheating. On this note, if we can generate, with the help of AI, sufficient amount of exercises of sufficient interest and appropriate difficulty level for an individual student, that in itself may result in a reduction in cheating.

Since the cost of cheating, e.g. potential punishment, is a known strong deterrent for such behaviour [188,194], a possibility to make taking AI-powered shortcuts less desirable in e-learning is to ask learners questions of their solutions. This could be done at scale for instance by generating questions with LLMs [29] in conjunction with AI-detection tools [90,107,200] to highlight the most obvious cases (sophisticated attempts can be difficult to identify [132]), or through automatically generated questions from code submission ASTs [150,149,148,151]. The latter method involves extracting information from the code to generate for example simple multiple-choice questions that are individualised for the specific code submission. Detecting the use of AI tools to solve such questions may prove difficult, but the nature of the exercise itself can reduce the expected value of cheating — the effort to read the description and click an appropriate answer option is likely less than copying all relevant information to an AI-tool and getting it to produce a desired output.

Nonetheless, it is yet to be determined how much of a risk the use of AI tools such as LLMs that enable instant code creation with minimal thought pose for inducing unproductive learning behaviours, or how much they can provide shortcuts to effective learning, speeding up the overall process of acquiring the required knowledge and skills to match professional levels.

6.2.3 LLMs for Explanations and Feedback

In [Publication IV] we conducted a preliminary exploration of the capabilities of OpenAI Codex in explaining code snippets in plain English. The results showed potential for the practical utility of Codex code explanations, but also left a lot to be desired in that regard as the explanations often contained minor inaccuracies. On the other hand, GPT-3, portrayed much greater value in explaining code in both [Publication V] and [Publication VI]. Whereas Codex was prone to go off-topic and generate random code snippets, GPT-3 consistently
generated more useful explanations with a standard structure, despite (or perhaps because of) not being fine-tuned for code related tasks.

In Publication V, GPT-3 was used to explain code snippets in natural language to students in online e-book aimed to teach basic web software development, achieving mainly positive reviews from learners in terms of helpfulness. In Publication VI, GPT-3 explanations were mixed up with learner explanations of the same code snippets, and on average, learners evaluated the GPT-3 explanations as slightly more accurate and easy to understand. A similar result can be seen in [58], where students rated GPT-3 explanations regarding their helpfulness and accuracy equally as good as those of their peers.

Thus, it can be reliably said that LLMs at the level of GPT-3 are capable of producing code explanations of at least the average quality as what can be expected with learnersourcing. Furthermore, the explanations have been esteemed helpful by learners for their studies, and the current and future models that are increasingly more accurate [272, 125] are likely to exceed the already evidenced capability of the existing models.

In our thematic analysis in Publication VI, we found that students preferred line-by-line explanations. This finding was somewhat surprising as prior work on ‘explain-in-plain-English’ code explanation tasks has typically rated ‘relational’ responses (short, abstract descriptions of the purpose of the code) higher than ‘multi-structural’ (line-by-line) responses [88, 287]. This suggests that there might be a mismatch between instructor and student opinions on what constitutes a good explanation. It might even be that some prior work has “unfairly” rated student multi-structural explanations lower, since students might have been capable of producing the more abstract relational explanations, but were thinking that longer, more detailed explanations are more preferable, and thus produced those types of explanations.

In addition, the level of mastery likely affects the desired level of detail — the less proficient preferring more detailed explanations. This notion is supported for example by the results of the study by Murphy et al. [195], where those who were better at coding were also more likely to explain the relational aspects in code. Further, we found the preference for more detailed explanations in Publication VI, which comprised evaluations from novices at the first year of university. In contrast, in Publication V, which involved a course for second to third year students and where we included three types of code explanations (line-by-line, summary, concepts) by GPT-3 into the course materials, we found that the students valued the summary explanations the most.

In the thematic analysis, we also observed that the LLM-created explanations closely followed a standard format. It is possible that showing students LLM-created explanations could help them adopt a standard format for their own explanations, which would possibly help them make better explanations, which in turn could improve learning [195, 48]. This notion is similar to what has been observed in prior work: templates can help designers frame better
problems [179] and help students write better emails [112].

6.3 Discussion on RT3

Towards more timely feedback

6.3.1 Faster and More Efficient Feedback

In Publication VII, we proposed a framework to speed up the automated assessment of programming exercises that consist of four steps: 1) cache check, 2) static analysis, 3) machine learning based assessment and 4) running the program against instructor-given unit or I/O tests. The steps are executed in order and if any one step is not passed, feedback is provided immediately without requiring to run the further steps. Our framework effectively reduces the need to run the program in the last step — which potentially can take much longer than the other steps — resulting in much faster assessment feedback as well as reduced strain on computational resources, as opposed to the typical scenario of relying solely on step 4) for automated assessment of programming exercises.

Our empirical analysis of the machine learning step showed that even a highly simple machine learning model could correctly identify seven out ten incorrect submissions while keeping the prediction precision high at 95% (one correct submission out of twenty would be misidentified as incorrect). We note that the precision could be scaled down to increase recall of the incorrect submission but we’d rather advocate against this; the predictions need to uphold a high level of precision to warrant trust in them. In addition, due to the imprecise nature of the machine learning step, we recommend incorporating an option for learners to ignore the machine learning step when so desired, as well as making the step transparent by explaining its possible faults. Future work could explore how learners would appreciate and adhere to the machine learning step checks when provided the choice to ignore them.

Regarding the speeding up of the automated assessment process, the effects of the extremely immediate feedback that does not require running the program against tests are unclear in regards of learning. Although immediate versus delayed feedback timing has been extensively researched, the focus has been on longer delays, e.g. a delay of multiple days or feedback given after a multi-item test has been completed as opposed to providing feedback after completing each item in a test [273][13]. This is the case for example in the study by Mullet et al. [193] that portrayed benefits for incorporating a one week delay in giving feedback — despite learners preferring immediate feedback. Similar time ranges can be found in studies of general timeliness. Bayerlein [18] finds that higher education learners do not differentiate between timely and extremely timely feedback where even the extremely timely feedback has a delay of over...
two days.

While some studies have looked into the effects of small feedback delays in the timespan of seconds, these portray varying results for learning. The study by Schrotz and Lund [255] on artificial grammar learning showed a decrease in learning speed when introducing delays of 10, 20 or 30 second delays in feedback — the larger the delay the slower the learning. On the other hand, it was also observed that with more delay, learning was better in terms of more correct responses in a transfer task. In a similar vein, Carpenter et al. [31] find that the retention of learned face-name pairs is improved when a three-second delay is introduced to feedback. In contrast, studying artificial grammar learning as in [255], Opitz et al. [199] found immediate feedback amounting to much larger learning gains than feedback with a delay of only one second. In a more in-depth study in the context of obscure trivia questions by Mullaney et at [192], the authors found that feedback delay of several seconds helps learning, but only when the learner wants to know the answer and the delay is unpredictable.

In addition to the effects for short or mid-term learning gains, it is also important to consider the effect of feedback delays to learner satisfaction and overall enjoyment to ensure persistence in studies over longer periods times. Here, parallels can be drawn from studies in human computer interaction. For example, a recent study [89] shows that even miniscule in-process delays can significantly affect online shopping behaviour by reducing total sales and revenue per website visitor. It is possible that miniscule feedback delay has a similar — albeit probably less pronounced — effect in reduced interest to continue for example in an online MOOC, especially if the course is not mandatory for the learner to complete. Future research could look into what effect small delays in feedback can have in learner engagement, frustration as well as learning speed and gains in the context of online programming education.

Looking aside our framework’s focus on speeding up assessment, the incorporation of static analysis and machine learning based checks as part of the assessment provides opportunities for feedback that would be unfeasible or impossible through tests that assess only code execution. In particular, as a later arriver to the scene, machine learning methods and their potential for enhanced feedback should not be overlooked, as it can help overcome the limitation of the other methods. For example, Messer et al. [187] raise maintainability and readability as understudied avenues for automated assessment through machine learning methods, while mentioning that program correctness can already be well covered using existing static and dynamic techniques. Further, as Higgins et al. [104] aptly suggest in their report of meaning and impact of assessment feedback: “it is not usually sufficient simply to tell a student where they have gone wrong — misconceptions need to be explained and improvements for future work suggested”. It is particularly in these areas where machine learning offers much potential, and already success too.
The machine learning step can also encompass feedback provided by LLMs, which have been shown to provide multiple kinds of effective formative feedback. Jia et al. [120] have showed that the BART model — a model predating GPT-3 — is capable of providing near human-level feedback on programming project reports. On the other hand, Balse et al. [16] studied GPT-3 for evaluating code correctness, critiques or suggestions to improve the code, and found the model outputs too inaccurate to be useful. In a similar note, while working with LLMs in our publications, we did not formally study GPT-3 on incorrect code, but we did notice GPT-3 and Codex being less proficient in explaining buggy code compared to correct code.

Studies on the latest state-of-the-art show a significant improvement in this regard, however. Phung et al. [223] evidenced that GPT-4 is vastly better than GPT-3.5 (ChatGPT) — which in turn is known to be a significant improvement over GPT-3 [272] — and comes close to human tutors in program repair and contextual explanation. On the other hand, the study shows that GPT-4 falls behind tutors in hint generation, grading feedback, pair programming, and especially task synthesis (generating new exercises that focus on specific concepts or bugs). In another study, Li et al. [157] used static analysis in conjunction with GPT-4 to achieve a precision of 50% and a perfect (100%) recall in known UBI (use-before-initialisation) bug detection. They even report that the method was able to find previously unidentified UBI bugs in the Linux kernel.

6.3.2 Towards Automatic Feedback Interventions

On a completely different look to speeding up programming exercise feedback, in Publication VIII we found that the majority of online environments for learning programming that provide automated feedback provide it only at the learner’s behest. As an exception, a couple of environments provided feedback on compiler issues and style such as dead code — which has been the default behaviour in modern IDEs for some time.

The lack of dynamic feedback immediately upon typing code rather than waiting for explicit separate action risks slow progress towards goal, or even much time spent with no progress at all, as well as unnecessary frustration. As an example, a learner may only request feedback, e.g. by submitting faulty code, when already frustrated with an exercise — which is a scenario that should be avoided [103]. Imaginably, the frustration can be only worse if the learner has spent much time and effort on their solution only to receive feedback about not having made any progress. All of this could be countered through automated interventions in the form of corrective feedback — when the feedback covers also semantic and logical issues in code and not only syntactical and stylistic (which could be provided through existing online IDE implementations).

Indeed, while the topic is not extensively researched, some success has
been evidenced. Gusukuma et al. implemented a misconception driven feedback mechanism that, through static analysis, was capable of providing immediate feedback during code editing about common misconceptions. The provision of the feedback mechanism resulted in a 10% increase in learning gains. Similarly at the code edit level precision, Marwan et al. studied the incorporation of positive and corrective feedback related to achieving task subgoals in a summer camp for high school, where the participants worked with iSnap, a block-based programming language environment. In the study, the authors found multiple positive results for the added feedback: they found that the feedback was well received by the participants, improved intention to persist in CS, reduced idle-time, and increased likelihood to complete the task objective and learning gains.

Naturally, the feedback provided by automated interventions itself can cause frustration for an unexpecting learner. Feedback that isn’t explicitly prompted may feel distracting or bothersome rather than helpful. As a notorious example, the infamous Microsoft office assistant, Clippy, was discontinued due to being more frustrating than useful. Although, the reported negative feelings towards Clippy can be, at least partly, blamed on it providing wrong suggestions without an explanation for why the suggestions were made.

Besides aiming for high rates of feedback quality and making the reasoning for given feedback clear to learners, other possibilities to reduce potential frustration for unwarranted feedback are for example to provide it only after adequate pauses in workflow, or making the feedback easy to ignore if so desired. As an example, the feedback could be behind a noticeable but non-intrusively sized and located button, that only informs the learners of potentially helpful feedback being available. Care needs to be taken also to not make the feedback too easy to ignore. In the study by Cody et al., unsolicited hints were better for learning gains when they were shown directly instead of requiring action to see the hint. In addition, as seen in, making code-edit level automated feedback encouraging, motivational, helpful or otherwise meaningful can induce a positive reception towards the unsolicited feedback.

While the content, quality and presentation of feedback is paramount for it to be useful, the timing when the feedback is provided is also key. Especially when the feedback is unsolicited, as feedback given very often or immediately after mistakes can be worse than on-demand feedback, and feedback given late can cause unnecessary time being wasted on wrong paths. Our analysis of expert annotations on when and how to give feedback on programming showed that experts tended to give feedback when they are quite certain that learners are on a problematic path or appear stuck. Minor logic errors would be left for learners to solve themselves unless the error appeared to cause further problems. It is however unclear how well the expert opinions of when feedback should be given aligns with when it is optimal to give with respect to learning gains. Different experts can also have somewhat differing opinions.
Discussion

on when it would best to provide feedback, as evidenced in our analysis.

Optimally, the time when (and whether) to provide feedback on programming process could be determined through automated systems that provide such feedback. In theory, given data of fine-grained program edits, it would be possible to precisely determine when learners are not progressing or are showing lack of understanding similarly as human tutors. Our proposed guidelines that we use for annotation (in Table 5.1) describe multiple kinds of events, including compiler, semantic and logical errors, deviation from specification, and trial and error behaviour, as well as how these should be addressed through feedback. Some errors, for example trial and error behaviour where the learner iterates through different conditional operands, could be identified with static analysis methods. However, in many cases static analysis would be unfeasible as a solution for spotting when to intervene, because there are so many ways in which the errors or unfruitful behaviour can materialise. As Dong et al. [72] note in a related study on why experts intervene on block-based programming assignments, to provide automated feedback interventions comparable to human tutors, "we not only need to identify when a student is struggling to provide proactive intervention, but also need to understand why the student is struggling and how a human tutor would have supported the student to meet their specific needs".

Here, machine learning models could help, as they could learn to automatically identify when to intervene from fine-grained programming edit data (trained to follow the guidelines from experts). While fully supervised methods may be a difficult starting point due to the extensive manual effort required to label enough data, exploring LLMs and how they could be fine-tuned for the task could provide a viable alternative. With automatic identification in place, further studies could investigate the effectiveness of such automatic interventions and optimise them further. Nonetheless, much suitable data will be required to leverage machine learning models properly for such a task.
7. Conclusion

This concluding chapter of the dissertation provides a brief summary of key contributions in relation to the explored research themes, outlines the limitations, and discusses potential future directions.

7.1 Revisiting the Contributions

The three research themes covered in this dissertation center around the question that sets the central theme of the work “How can machine learning be leveraged in large scale programming education to (a) improve student learning and (b) reduce teacher workload?”. Here, we first summarise the contributions of this dissertation in relation to these themes.

RT1: Identifying learners in need of additional help or practice Dropout prediction and knowledge tracing (KT) are two widely researched applications for machine learning methods in education. While many battle-tested models already exist for these applications, understanding which model to employ for maximal benefit in a specific context may still prove difficult.

Publication I demonstrates how a methodological issue present in dropout prediction literature can cause distorted perception of model performance, leading to a risk of employing suboptimal models in practice. Publication II provides an empirical evaluation of deep learning for knowledge tracing models (DLKT) and how hyperparameter and metric choices matter in deciding the most powerful model.

Publication III adds to the existing literature on persisting in MOOCs a fresh look at the topic from the perspective of introductory programming for lifelong learners. The study investigates the effects of motivational aspects and background on continuing in a series of courses and repeats the common observation of high dropout rates with men who have prior experience being the most likely to persist.
RT2: Creating educational content and exercises

We propose and evaluate novel methods involving large language models for automated programming exercise and code explanation creation (Publication IV). The created exercises could be generated with problem description, a matching model solution and automated tests, all at once. The automatically generated exercises were found to be novel, to contain sensible problems for learner to practice on, and to be easily tuned to specified themes and programming concepts.

LLM-generated code explanations were rated helpful by learners when incorporated in a real educational setting within an e-book (Publication V). When compared against learnersourced code explanations (Publication VI), the LLM-generated explanations were rated as more accurate and easier to understand than the explanations given by other learners. These results pave way for new computing education research on LLMs and educational content creation.

RT3: Towards more timely feedback

We look at two ways the timeliness of programming exercise feedback can be improved. Firstly, we look at the speed of delivery: when feedback is requested, the time it takes for the feedback to be ready can be reduced. Secondly, we look at when should proactive feedback be given: unsolicited feedback intervention timing can be optimised to maximise feedback usefulness and to minimise ill-timed distractions.

To improve the time it takes to receive feedback for programming code submissions, we propose an automated assessment framework that combines cache checks, static analysis, machine learning and program execution based tests. We demonstrate that our framework can speed up automated assessment, and consequentially the provision of feedback, by nearly 80% compared to the traditional case of relying merely on program execution based tests. We also show that even a highly simplistic machine learning model can, with high precision, correctly identify nearly half of incorrect submissions as incorrect. The combination of static analysis, machine learning models and traditional automated tests also provides fertile grounds for improved targeted feedback.

As a step towards improving the timing of unsolicited feedback interventions, we propose guidelines for annotating — and annotate — fine-grained data of learners’ programming exercises solving process with information on when, why and how expert educators would intervene and provide feedback (Publication VIII). We have published two such annotated datasets as part of the work. This data can serve as a basis for developing as well as evaluating machine learning models that can automatically intervene and provide feedback to learners in a timely manner.
### 7.2 Piecing strings together

The contributions of this dissertation encompass many different ways ML can aid large scale computing education, from making it easier and faster for educators to generate education materials (essential for large scale personalised content), to automatically recommending content and providing feedback to students for more efficient learning. One common denominator of all the publications is the applicability of their results in intelligent tutoring systems and other computer-based learning environments designed for large scale education.

The existence of educational material for learners to consume is a necessary prerequisite for any learning system (bar fully automated content generation but we are not yet there) and effective knowledge tracing is paramount in suggesting appropriate exercises for practice and content for reading. Automated identification of struggling learners can be easily incorporated into learning systems, but they need to be robust in order to be useful in practice. Lastly, feedback is an essential component in learning, and the automatic feedback of learning systems should be fast and timely to minimise frustration and time wasted from being stuck solving exercises.

As a summary of the contributions of this dissertation, Figure 7.1 illustrates the contributions from the perspectives of the learner and the educator.

---

**Figure 7.1.** An illustration of the contributions of this dissertation from the perspectives of the learner and the educator.

<table>
<thead>
<tr>
<th>Learner</th>
<th>Automation for materials and exercises</th>
<th>Robust at-risk identification and learner modeling</th>
<th>Automated feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Increased engagement</td>
<td>Help received when needed, recommended content is meaningful</td>
<td>Reduced frustration, faster learning</td>
</tr>
<tr>
<td></td>
<td>Varied and personalised content</td>
<td>Accurate predictions</td>
<td>Faster Proactive</td>
</tr>
<tr>
<td>Education</td>
<td>Reduced effort in creating</td>
<td>Less time wasted on false flags</td>
<td>Can concentrate on more intricate feedback</td>
</tr>
</tbody>
</table>

---

97
7.3 Limitations of the Work

This dissertation and the work involved has multiple limitations, which are outlined here in consideration to both internal and external validity. Internal validity refers to how study results can be attributed to the treatment in question rather than to other factors. External validity, on the other hand, refers to how the results generalise to other situations.

We note a few limitations when comparing the two dropout prediction approaches examined in Publication I, one involving the exclusion of learners who have already dropped out when forecasting future dropouts, and the other involving their inclusion. Firstly, our definition of dropout is based on inactivity, yet students who drop out late may still manage to achieve passing grades. Our data indicates that very few students re-engage after becoming inactive, implying that our results are relatively unaffected by this. However, the possibility remains that the inclusive approach may handle such (albeit rare) cases better, despite its shortcomings. Thirdly, as is common in many dropout prediction evaluations, our analysis relies on data from the same course, which could introduce bias into our findings. Additionally, we did not delve into demographic or more detailed features, making it impossible to assess how the two approaches would perform under such conditions. Although our study encompassed three datasets from different courses, each with distinct focuses and instructors, they all originated from the same university. This potentially limits how our results generalise to other contexts.

In Publication II, the evaluated DLKT models are not the most recent ones (partly because of the time the evaluation took, and partly because we wanted to focus on models that already had attained some traction) and it is possible that the results may not generalise to newer models — we found that robustness to the various tested hyperparameters varied amongst the different types of evaluated DLKT models. With our main focus on DLKT models, we also reimplemented only the evaluated DLKT models, and not the other machine learning models used as baselines in the work for which we used readily available code. We also note that while reimplementations can overcome possible flaws in original code, they may also introduce new ones, potentially causing differences observed performance. Another internal validity concern is the studied hyperparameter space for evaluation. Although we did evaluate the models with a large number of different hyperparameters, a larger space could enable finding more optimal hyperparameters for the models. Finally, even though we used seven different datasets for our evaluation, the results may not generalise to other datasets. This is partly evidenced by the variance in model performance in the different datasets. In addition, recent studies (e.g. [181, 208]) have evaluated models on the EdNet dataset [43], which is much larger than any of the datasets used in our study. Such data could be better suited for deep learning models which are known to excel on large data.

A major limitation in the investigation of who continues in a series of
programming courses for lifelong learning in Publication III is that we only consider learners who have completed at least 30% of the first course in the course series. This is due to introducing the final background survey for the learners at that point — different surveys are presented at different points to minimise effects of survey fatigue. This causes sampling bias as many are likely to have already dropped out of the first course at that point. Another source of bias comes from the surveys being optional. The issue of excluding the early dropouts prior to the 30% point cutoff may be mitigated by the majority of them being merely exploring rather than interested in completing the course. However, without the data, we have no way of verifying this. Also, like in Publication I we use activity as a proxy for dropout. In this case, we excluded any learner with exercise submissions in the last 30 days of our data to avoid the majority of false drop out misclassifications, but we acknowledge that some learners do continue the courses after a month’s break (albeit very rare in our data). Further, the study is only of a small-scale and conducted on a single course series, and thus the generalisability of the results to other contexts is unclear without additional studies.

In Publication IV where we employed OpenAI Codex to generate programming exercises and code explanations, a threat to generalisability is that we evaluated a relatively small set of samples. Another one is that each sample was evaluated by only a single evaluator. We did, however, have clear guidelines for the evaluation and discussed any uncertain cases when any arose. Our sample pool also consisted only of small exercises that could be solved by writing a single simple function or class. As such, our results say little about how LLMs would perform in generating larger or more complex exercises and code snippets. In addition, our results suffer from not being replicable as the model we used was proprietary and no longer available — this is not too much of a problem however, since newer models quite certainly produce even better exercises and explanations.

In Publication V we evaluated the usefulness of LLM-generated code explanations in a real educational setting. An important limitation is the optionality of inspecting the explanations in our setting, which risks that the participants in the study may not represent the “average” participants of the course. Further the study was conducted in a single relatively small course, and as such provides only preliminary results. When comparing learner and LLM generated explanations in Publication VI, we note a similar limitation as this study also was performed only in the context of a single course. Although, the contexts of the two studies are different, with the first being conducted in Finland in a web software development course, and the second in New Zealand in a course for novices. Looking at the two studies together provides a better picture than each alone. However, both studies only evaluate the quality or usefulness through survey responses from learners, which does not directly translate to actual benefits for learning.

The evaluation of our proposed framework for speeding up programming
assessment in Publication VII involved measuring the number of cache, static analysis and machine learning check matches for erroneous submissions. We note that our evaluation context was limited to a single course and one that contained a high number of small simple exercises — cache checking is likely more efficient in such cases than when exercise pools consist of larger and more complex exercises. Further, our prototype static analysis and machine learning steps are extremely basic and provide only preliminary baseline results. It also remains unknown what are the benefits when programming assessment feedback delivery is sped up by mere few seconds, which would be commonly the case for small exercises that are fast to compile and execute. Although not a focus of the study, a limitation is also that the study discusses only little the vastly potential benefits of combining the static analysis and machine learning steps with program execution based steps for extremely improved and more extensive feedback.

In Publication VIII, we provide (1) annotated data (and propose guidelines for the annotation process) of when, why and how experts intervene and provide feedback for learners in the process of solving programming exercises, and (2) review existing openly available learning environments for programming on how they offer feedback to learners. The largest limitation for our data annotation process arises from the difficulty to find appropriately fine-grained open datasets of learners practicing programming. We initially found several datasets but in the end, only two of them were deemed suitable for the intended purpose, as the others lacked important details such as exercise descriptions. The small set of examined data is likely to affect to affect how the guidelines are suited for varying data. We also analysed only a small set of problems and submissions per problem due to the extensive effort required for manual annotation. Another issue is our observed disagreement among experts on when to intervene, although, this observation also denotes that we were able to capture varying expert opinions. A major limitation in our learning environment review is the exclusion of restricted access learning environments, which may offer more extensive feedback features than the ones we reviewed. In addition, we cannot guarantee that the environments that we found for review accurately represent all openly available environments.

7.4 Future Work

7.4.1 Methodological Research

The two first publications of this dissertation target methodological issues in machine learning for education, and future work targeting methodological issues in machine learning for education will continue to be important — perhaps now more than ever. New applied machine learning research for
education (and other areas) is published rapidly to keep up with the speed of development seen in machine learning models, particularly on research related to LLMs but also in other areas such as KT. The remarkable speed of the field going onwards can also be observed adding pressure to publish work quickly in open venues such as arXiv to not have to wait for the research to appear in peer reviewed venues — many recent works in computing education and machine learning accumulate many citations prior to peer review (see e.g. [229]). While overall the acceleration in publishing is a magnificently positive issue, it adds to the need for methodological and replication research that can help guide the quickly moving field towards the most worthwhile directions amongst the many, and the need is exacerbated by increased citing of works that haven’t gone through peer review.

7.4.2 Persistence in Online Education

Persistence, or the lack of it, is a pressing and persisting issue in online education even though it has been extensively studied. As noted in [Publication III] the features we analysed explained only little of the variance in continuing in the studied course series. One direction for future work is to analyse more features and their combined effect in different contexts and scenarios – as suggested by Shaikh and Asif [257] – to better understand the intricacies in who persists in online education.

Another direction, perhaps a more lucrative one, is to invent and evaluate new interventions that target those at risk of dropping out, especially because many prior interventions have been noted as tending to be ineffective or have only minor effect [295]. One way forward could also be to collect data on effective and ineffective interventions, then train ML models on the data to automatically choose the most suitable interventions for a given situation. The recent A/B study by Borrella et al. [26] portrays scaffolding difficult content and gradually increasing test difficulty as effective ways to reduce dropout. This implies that combining dropout prediction models together with learner modeling such as DLKT to recommend additional practice exercises of suitable difficulty could be a useful intervention for future work to explore.

Another possibility for increasing course retention rates is through providing personalised exercises that appeal to learners’ interests. An example case where future work on specialised exercises for added interest could be particularly helpful is the small percentage of women in computing education [133]. The gender gap has been noted to be related more to interest and personal beliefs of competence (self-efficacy) than to actual competence [39 264], and our examination on learners persistence in lifelong course series portrayed (self-identifying) women as much more likely to dropout than men (although this was partly explained by previous experience). Meta-analysis has also identified gender as an important factor in determining dropout [295] (although a lack of consensus on the matter has also been noted in prior research [147]).
7.4.3 Leveraging Large Language Models for Content Creation

Both of the interventions proposed above would benefit greatly from the ability to generate personalised exercises and content automatically (at least well enough to help educators create new content faster). Our preliminary study shows the potential of LLMs for this, but our work also exhibits the necessity of keeping humans in the loop as the model outputs are far from perfect on many occasions. Further, the speedup for creating new exercises has not been empirically validated. As newer LLMs are much more powerful than the one we evaluated and expected to evolve further \[28, 251\], future work can explore their usefulness as automated exercise generators.

The latest state-of-the-art model GPT-4 has already been evaluated on this to some extent with Phung et al. \[223\] examining GPT-4 for generating debugging tasks in the form of buggy solutions that target specific misconceptions. Their results show that while GPT-4 is already remarkably capable, the quality of its generated tasks falls short of human tutors by a large margin. While evaluating the raw quality of LLMs for targeted exercise and content generation is one important avenue for future work, it could be fruitful to explore how LLM output validation could be automated to ensure sufficient quality and adherence to desired traits of automatically generated content, following for example the “robosourcing” model described in \[60\].

Once reaching sufficient quality, the benefits of incorporating automated content to learning systems could also be studied to determine how useful such can be for helping learners. We did preliminary work for evaluating automatically generated code explanations', showing that they were perceived as useful by learners, and better in quality than the explanations by peers. Nonetheless, it remains unexplored how much of an effect the explanations have on learning or persistence in courses.

LLMs are also becoming smaller and more efficient \[105, 109\], paving the way for effective LLMs that can be leverage learner data and preferences without compromising privacy as the models can be run on non-corporate or even individual machines. This makes studying such models particularly useful: is important to not deny learners the benefits that modern LLMs can provide for learning, and at the same time it is important to honour the privacy of learners and their actions.

7.4.4 Timeliness of Feedback

We examined the use of cache, static analysis and machine learning methods to overcome the slowness of automated tests that require executing submitted code, which of course depend on the compiler and processing power as well as submission and test codes. In most cases, assuming the majority of exercises do not require (or risk) highly performance intensive solutions, the speedup provided by the framework won’t be drastic per submission, but will likely...
remain within a few seconds like in our preliminary exploration in Publication VII. Future work could look into how to best deliver assessment feedback for programming learners in terms of extremely short feedback delays in the matter of seconds. Is it beneficial to have a few second delay and if yes, in what situations and for what kinds of feedback (like how Mullaney et al. [192] found several seconds of delay in answer correctness feedback helpful for trivia questions, but only when the delay time was unpredictable and the learner wanted to know the answer), or should we aim for delays of less than a second whenever possible?

We also explored when and how experts provide feedback on the process of learners solving programming exercises and provide datasets with expert annotations. This lays the groundwork for training and evaluating learning models that could learn appropriate moments to provide timely feedback interventions. However, much more data and work is needed to understand when would be the best moments to intervene on learners’ work, and especially to properly train machine learning models for the task, or even to understand whether machine learning models pose the best solution for the job.


References


[32] Sarah Carr. As Distance Education Comes of Age, the Challenge is Keeping the Students. *Chronicle of Higher Education*, 46(23), 2000.


References


References


References


References


References


References


References


[279] James Vincent. This AI-powered Autocompletion Software is Gmail’s Smart Compose for Coders, July 2019.


References


