UNIVERSITY OF TECHNOLOGY SYDNEY, AUSTRALIA.

DOCTORAL THESIS

Data Analytics and the Novice Programmer

Author:
Alireza AHADI

Supervisor:
Associate Professor
Raymond LISTER
(principal) and Dr Julia
PRIOR (co-supervisor)

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Human Centred Technology Design
School of Software

January 22, 2018

Declaration of Authorship

I, Alireza AHADI, declare that this thesis titled, “Data Analytics and the Novice Programmer” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"When I was in college, my graduation thesis was called 'Female Directors.' I interviewed all of the important female directors from Mexico. There were four. That was it. "

Patricia Riggen

UNIVERSITY OF TECHNOLOGY SYDNEY, AUSTRALIA.

Abstract

Faculty of Engineering and Information Technology
School of Software

Doctor of Philosophy

Data Analytics and the Novice Programmer

by Alireza AHADI

The aptitude of students for learning how to program (henceforth *Programming learn-ability*) has always been of interest to the computer science education researcher. This issue of aptitude has been attacked by many researchers and as a result, different algorithms have been developed to quantify aptitude using different methods. Advances in online MOOC systems, automated grading systems, and programming environments with the capability of capturing data about how the novice programmer's behavior has resulted in a new stream of studying novice programmer, with a focus on data at large scale. This dissertation applies contemporary machine learning based analysis methods on such "big" data to investigate novice programmers, with a focus on novices at the early stages of their first semester. Throughout the thesis, I will demonstrate how machine learning techniques can be used to detect novices in need of assistance in the early stages of the semester. Based on the results presented in this dissertation, a new algorithm to profile novices coding aptitude is proposed and its' performance is investigated. My dissertation expands the range of exploration by considering the element of context. I argue that the differential patterns recognized among different population of novices is very sensitive to variations in data, context and language; hence validating the necessity of context-independent methods of analyzing the data.

CONFIRMATION OF ETHICS CLEARANCE

Human Negligible Low Risk Ethical clearance was granted for this PhD project by the University of Technology Sydney Human Research Ethics Committee under approval numbers ETH16-0340 (see Appendix D).

Acknowledgements

I would like to express my special appreciation and thanks to my adviser associate professor Raymond Lister, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on my research have been priceless.

A special thanks to my family. Words cannot express how grateful I am to my father, mother and my beloved sister for all of the sacrifices that you've made on my behalf.

Alireza Ahadi

KEYWORDS

Novice Programmer
Human Factors
Measurement
Computer Science Education
Data Mining
Online assessment
MOOC
Databases
SQL Queries
Learning Edge Momentum
Bimodal Grade Distribution
Machine Learning
Programming Source Code Snapshot
Pattern Recognition
Classification
Supervised Machine Learning

Contents

Declaration of Authorship **iii**

Abstract **vii**

Acknowledgements **xi**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Motivation . . . 2
1.3 Research Questions . . . 2
1.4 Research Design . . . 2
1.5 Thesis Structure . . . 3
Chapter 2: Background . . . 3
Chapter 3: Results Overview . . . 3
Chapter 4: Method . . . 3
Chapter 5 to Chapter 12 . . . 3
Chapter 13: Discussion and Conclusion . . . 3
1.6 Significance and Contribution . . . 3
1.7 Conclusion . . . 4

2 Background **5**

2.1 Introduction . . . 5
2.2 Who is a novice programmer? . . . 5
2.3 Data analysis . . . 7
2.3.1 Static success factors . . . 8
2.3.2 Success factors and contradictory reports . . . 10
2.3.3 Dynamically accumulating data . . . 11
2.4 Dynamic Programming Source Code Snapshot Data Collection Tools . . . 14
2.4.1 Blackbox . . . 14
2.4.2 CloudCoder . . . 14
2.4.3 CodeWorkout . . . 15
2.4.4 JS-Parsons . . . 15
2.4.5 PCRS [118] . . . 16
2.4.6 Problets . . . 16
2.4.7 TestMyCode . . . 16

2.4.8 URI Online Judge 17
2.4.9 UUhistle 17
2.4.10 Web-CAT 17
2.5 Publicly Available Programming Source Code Snapshot Datasets 18
2.5.1 Blackbox 18
2.5.2 Code Hunt 18
2.5.3 Code.Org 19
2.6 Approaches for Data Analysis 19
Educational data mining and learning analytics . . . 19
Machine learning and data mining approaches 20
2.7 Machine Learning in CSed 21
2.8 Assessment 23
2.9 Studies of the Novice Programming Process 24
2.10 The analysis of the novice programmer errors 24

3 Results Overview 29
3.1 Results Overview 29
3.2 The Thesis About The Thesis 29
3.2.1 Research Questions 1 30
Paper 1. Chapter 5: Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant? 30
Paper 2. Chapter 6: Exploring machine learning methods to automatically identify students in need of assistance 30
Paper 3. Chapter 7: A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries 30
Paper 4. Chapter 8: Students’ Semantic Mistakes in Writing Seven Different Types of SQL Queries 31
Paper 5. Chapter 9: Students’ Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students’ Success 31
3.2.2 Research Question 2 31
Paper 6. Chapter 10: Performance and Consistency in Learning to Program 32
Paper 7. Chapter 11: On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions 32

Paper 8. Chapter 12: A Contingency Table Derived Methodology for Analyzing Course Data . 32

4 Method 33
4.1 Introduction 33
4.2 Background 33
4.2.1 Programming source code snapshots, collected at the University of Helsinki. 36
4.2.2 Database source code snapshots, collected at the University of Technology Sydney. 37

5 Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant? 43
5.1 Introduction 43
5.1.1 Statement of Contribution of Co-Authors 43
5.2 PDF of the Published Paper 46
5.3 Discussion 53

6 Exploring machine learning methods to automatically identify students in need of assistance 55
6.1 Introduction 55
6.1.1 Statement of Contribution of Co-Authors 55
6.2 PDF of the Published Paper 58
6.3 Discussion 69

7 A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries 71
7.1 Introduction 71
7.1.1 Statement of Contribution of Co-Authors 71
7.2 PDF of the Published Paper 74
7.3 Discussion 81

8 Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries 83
8.1 Introduction 83
8.1.1 Statement of Contribution of Co-Authors 83
8.2 PDF of the Published Paper 86
8.3 Discussion 93

9 Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success 95
9.1 Introduction 95
9.1.1 Statement of Contribution of Co-Authors 95
9.2 PDF of the Published Paper 98

9.3 Discussion 105

10 Performance and Consistency in Learning to Program 107

10.1 Introduction 107

10.1.1 Statement of Contribution of Co-Authors 107

10.2 PDF of the Published Paper 110

10.3 Discussion 117

11 On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions 119

11.1 Introduction 119

11.1.1 Statement of Contribution of Co-Authors 119

11.2 PDF of the Published Paper 122

11.3 Discussion 129

12 A Contingency Table Derived Method for Analyzing Course Data 131

12.1 Introduction 131

12.1.1 Statement of Contribution of Co-Authors 131

12.2 PDF of the Submitted Paper 134

13 Discussion and Conclusion 155

13.1 Introduction 155

13.2 Overview of Research 155

13.3 Research Questions 155

13.4 Research Outcome 156

13.5 Research highlights 156

13.6 Research Significance 157

13.7 Research Findings 157

13.7.1 Data 157

13.7.2 Method 158

13.7.3 Context 158

13.8 Discussion 159

13.8.1 Data 159

13.8.2 Method 159

13.8.3 Context 160

13.9 Limitations 161

13.9.1 Data 161

13.9.2 Context 161

13.10 Recommendations 161

13.10.1 Data 162

Static data 162

Dynamic data 162

13.10.2 Method 162
13.10.3 Features of a strong algorithm to capture students learning from the source code snapshot data 162
General Attributes 162
Language Independence 163
Distribution Independence 164
Cross-Context Parameter Variation in the Construction of a Metric 164
Validity of the Operationalization 164
13.10.4 Context 165
13.11 Conclusion 165

A Definition of Authorship and Contribution to Publication 167

B Specific Contributions of Co-Authors for Thesis by Published Papers 169

C Complete list of Publications by Candidate 179

D UTS Human Ethics Approval Certificate - UTS HREC - ETH16-0340 183

E Extract from UTS Subject Outline – 31271 "Database Fundamentals" Sem. 2 2016 193

F Extract from Helsinki Subject Outline – 581325 "Introduction to Programming" Sem. 2 2016 207

G Extract from UTS Database Fundamentals – 31271, Practice Questions and Answers 215
G.1 Introduction 215
G.2 Pizza Database 215
G.2.1 Questions Used in The Practice Online SQL Test . . . 219
G.2.2 Proposed Answers for The Questions Used in The Practice Online SQL Test 219

H The Final Exam Questions Used at Helsinki University 223

I Extract from Helsinki Subject Content – 581325 "Introduction to Programming" Sem. 2 2016, Week 1. 229

Bibliography 255

List of Figures

4.1 An abstract view of the logical services where data instrumentation and collection are typically performed. Note that some data collection tools encompass multiple logical services (Adapted from Ihantola et al., 2015). 34
4.2 Data can be collected at different levels of granularity, which implies different collection frequencies and associated data set sizes (Adapted from Ihantola et al., 2015). 35
4.3 Student's home screen at AsseSQL. Students may attempt questions in any order, as many times as they wish. 38
4.4 Screen for an individual question. The question and the simple output to clarify the question are presented for each question in the test. 39
4.5 Feedback screen for an individual question with the output of model answer as well as the student's answer are presented in this page. 39
4.6 Back to the student's home screen: students may return to this at anytime. 40
4.7 A sample of the ERD used in the test. 41

List of Tables

3.1 Research Questions, the thesis about the thesis, and the corresponding chapters 29

5.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 5 44

6.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 6 56

7.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 7 72

8.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 8 84

9.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 9 96

10.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 10 108

11.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 11 120

12.1 Authors' Area of Contribution for The Paper Corresponding to Chapter 12 132

G.1 List of questions and their corresponding covered topic. . . . 220

G.2 Proposed Answers for the questions presented in G.1 221

List of Abbreviations

ACC	**Acc**uracy
CMS	**C**ource **M**anagement **S**ystem
CTP	**C**omputational **T**hinking **P**atterns
DM	**D**ata **M**ining
DT	**D**ecision **T**ree
EDM	**E**ducational **D**ata **M**ining
EQ	**E**rror **Q**uotient
FN	**F**alse **N**egative
FP	**F**alse **P**ositive
FDR	**F**alse **D**iscovery **R**atio
FNR	**F**alse **N**egative **R**atio
FPR	**F**alse **P**ositive **R**atio
IDE	**I**ntegrated **D**evelopment **E**nvironment
JAR	**J**ava **A**rchive File
LMS	**L**earning **M**anagement **S**ystem
LN	**L**ogical **N**ecessity
LS	**L**ogical **S**ufficiency
LSI	Kolb's **L**earning **S**tyle **I**nventory
ML	**M**achine **L**earning
MSLQ	**M**otivated **S**trategies **L**earning **Q**uestionnaire
NN	**N**eural **N**etwork
NPSM	**N**ormalized **P**rogramming **S**tate **M**odel
PCA	**P**rincipal **C**omponent **A**nalysis
RF	**R**andom **F**orest
RED	**R**epeated **E**rror **D**ensity
RSE	**R**osenberg **S**elf-**E**steem
SEN	**Sen**sitivity
SPC	**Spe**cificity
SQL	**S**tructured **Q**uery **L**anguage
SVM	**S**upport **V**ector **M**achine
TN	**T**rue **N**egative
TP	**T**rue **P**ositive
WATWIN	**WAT**son & God**WIN**
Web-CAT	**Web**-based **C**enter for **A**utomated **T**esting

List of Publications by Candidate

Below are the publications first authored by the Candidate which contribute to this thesis by publication.

1. Ahadi, A. and Lister, R., 2013, August. Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?. In Proceedings of the ninth annual international ACM conference on International computing education research (pp. 123-128). ACM. (See Chapter 5)

2. Ahadi, A., Prior, J., Behbood, V. and Lister, R., 2015, June. A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (pp. 201-206). ACM. (See Chapter 7)

3. Ahadi, A., Lister, R., Haapala, H. and Vihavainen, A., 2015, July. Exploring machine learning methods to automatically identify students in need of assistance. In Proceedings of the eleventh annual International Conference on International Computing Education Research (pp. 121-130). ACM. (See Chapter 6)

4. Ahadi, A., Behbood, V., Vihavainen, A., Prior, J. and Lister, R., 2016, February. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (pp. 401-406). ACM. (See Chapter 9)

5. Ahadi, A., Prior, J., Behbood, V. and Lister, R., 2016, July. Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 272-277). ACM. (See Chapter 8)

6. Ahadi, A., Lister, R. and Vihavainen, A., 2016, July. On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 218-223). ACM. (See Chapter 11)

7. Ahadi, A., Lal, S., Leinonen, J., Lister, R. and Hellas, A. Performance and Consistency in Learning to Program. Australian Computing Education Conference, 2017 (Award winning paper for best student paper). (See Chapter 10)

8. Ahadi, A., Lister, R., Hellas, A. A Contingency Table Derived Methodology for Analyzing Course Data. 17, 3, (August 2017), 19 pages. DOI: https://doi.org/10.1145/3123814 (See Chapter 12)

THESIS BY PUBLICATION

This thesis is presented in the format of scholarly papers published during the period of my candidature, according to UTS regulations set out in the website http://uts.edu.au/current-students/dab/uts-graduate-research-school. The papers included in this thesis by publication form a research narrative which is summarized in Chapter 5. Each paper then forms a separate chapter of this thesis, inserted in its published format.

To all novice programmers. . .

Chapter 1

Introduction

This chapter provides the overview and the motivation for this research project, as well as my theoretical perspectives. The chapter concludes with research questions to be addressed in this thesis.

1.1 Motivation

Can we tell if students are learning to program?

Learning to program is hard. As such, it is not surprising that decades of research has been invested into pedagogical practices that could be used to help those struggling with the task (Vihavainen, Airaksinen, and Watson, 2014a). Whilst such research has resulted in improvements in course outcome in many institutions, the factors that contribute to students' learning programming still somewhat elude us.

One stream of research that has promise to explain these factors is related to automatically extracting students performance in a programming class. This work has been largely motivated by identifying those who are more apt to learn programming than others (Evans and Simkin, 1989). However, when traditional variables such as age, gender and personality are used to predict outcomes, the performance varies considerably over different contexts (Evans and Simkin, 1989; Watson, Li, and Godwin, 2014). More recent studies that use dynamically accumulated data such as source code snapshots or assignment progress, provide an additional and a *continuous* measure of students' progress, which has led to better performance of such models (Watson, Li, and Godwin, 2014). As an example of the use of continuous source code snapshot data, Jadud created a metric that quantifies how students fix errors (Jadud, 2006).

Early advances into measuring variables that could be used to categorize students are related to the understanding of students performance in courses. Evans and Simkin describe these as approximations for *programming aptitude*, which is often investigated in the light of the student's introductory programming course performance, e.g. course grade (Evans

and Simkin, 1989). This research has witnessed a continuing trend over the years. Before 1975, much of the research was focused on demographic factors such as educational background and previous course grades, while by the end of the 1970s, research moved gradually toward evaluating static tests that measure programming aptitude. A more recent trend has been to investigate the effect of cognitive factors such as abstraction ability and the ability to follow more complex processes and algorithms (Evans and Simkin, 1989). These lines of research have been continued to this day by introducing factors related to study behavior, learning styles and cognitive factors (Watson, Li, and Godwin, 2014). However, dynamically accumulating data such as programming process data has only recently gained researchers' attention (Jadud, 2006; Porter, Zingaro, and Lister, 2014; Vihavainen, 2013; Watson, Li, and Godwin, 2013). These techniques deploy source code programming snapshot data to provide a better understanding of students learning aptitude. However, non of these method have attempted to feed data analytics methods, and in particular, machine learning tools with the collected data.
In this dissertation, I adopt the machine learning and statistical data mining on the data collected from novices to explore the question "Can we tell if students are learning to program?" Thus, my original question can be re-expressed as:

1.2 Motivation

Can machine learning techniques tell us if students are learning to program?

1.3 Research Questions

How early can we detect those who are going to struggle/are struggling?

RQ1 Is it possible to identify struggling students by analyzing the source code snapshot data?

RQ2 Can we address the problem of the sensitivity of the prediction (of the struggling novice programmers) in a more context-independent manner?

1.4 Research Design

This PhD research project takes quantitative approaches and involves quantitative data collected from programming tasks completed by whole cohorts of students under exam conditions.

1.5 Thesis Structure

This thesis by publication includes a number of peer-reviewed papers. The remainder of the thesis is organized as follows:

Chapter 2: Background

This chapter is a review of related previous research. The focus is on how the data-driven methods have attempted to capture learning aptitude of the novice programmers.

Chapter 3: Results Overview

The chapter draws together the story produced by the publications included in this thesis. A commentary is given which describes how one or more of the papers address each research question.

Chapter 4: Method

This chapter is a detailed account and justification of the quantitative data collection process adopted in my research project.

Chapter 5 to Chapter 12

Most of these chapters include one of the published papers. Each chapter is preceded by a reiteration of the storyline and research question(s) addressed.

Chapter 13: Discussion and Conclusion

This chapter summaries how each of the research questions have been addressed, together with the significance, contribution and limitations of the research. The chapter concludes with proposals for future work.

1.6 Significance and Contribution

The contribution of my PhD research lies in the use of the machine learning frameworks to analyze data collected from the novice computer programmers with the goal of identifying students in need of assistance as early as possible. Through detailed analysis of the data using different statistical and machine learning based methods, I review the state of the art, proposed new methods to analyze novices data, and develop a new metric for quantifying the programming learn-ability of novices.

1.7 Conclusion

The purpose of this research is to study the data collected from the novice programmer in a way that has not been done before: designing a machine learning based framework which can best predict who is going to struggle to learn programming and who is not. The results of this research have valuable pedagogical implications.

Chapter 2

Background

This chapter reviews the related work done in the field of computer science education research. This thesis is focused on the intersection of three main research areas: educational data mining (EDM), the novice programmer, and machine learning. As this is a thesis by publication, the literature of relevance to each chapter/paper is discussed in detail in the paper itself. The aim of this "background" chapter is merely to describe literature that sets the scene for the papers that follow.

2.1 Introduction

Learning to program is hard. As such, it is not surprising that decades of research has been invested into pedagogical practices that could be used to help those struggling with the task (Vihavainen, Airaksinen, and Watson, 2014b). Whilst such research has resulted in improvements in course outcome in many institutions, the factors that contribute to students' learning programming still somewhat elude us.

2.2 Who is a novice programmer?

To understand the characteristics of a novice programmer, I first review the attributes of the expert programmer. Most studies have characterized the programming skill through either sophisticated knowledge representation and problem solving skills (Détienne, 1990; Gilmore, 1990). It is known that programming experts have well organized knowledge schema, have organized their knowledge according to the functional characteristics of the language, and have problem solving skills on both the general and specialized levels (Mayrhauser and Vans, 1994). Compared to the list of attributes associated with the expert programmer, novice programmers show poor levels of such attributes. Winslow expressed the view that novice programmers are limited to surface knowledge, and approach a program line by line (Winslow, 1996).

Knowledge and strategy are two cognitive dimensions in the process of programming. Knowledge is manifested in the ability of a programmer to explain what a specific construct does (e.g. what a for loop does). The strategy dimension describes ways that one might use knowledge of a construct, such as a for loop, when writing a program. With regard to the knowledge and strategy dimensions, Widowski and Eyferth (1986) compared novice and expert programmers. They analyzed the number of consecutive lines tested by a (novice or expert) programmer (defined as "run") and how many lines of code a programmer looked at between runs. Experts were found to read large amounts of "usual" code between runs, and thus used fewer runs, but read fewer lines of "unusual" programs between runs. Novices were found to read both conventional code and unusual code in the same way.

In the literature, there are some studies which focus on the differences and overlaps between code generation and code comprehension. Brooks introduced a model for program comprehension (Brooks, 1977) based on different *domains* starting from problem domains and ending with the programming domain. Mapping from the problem domain into intermediate domains leading to the final programming domain is the foundation of Brooks' model. Brooks stated that the expert programmer forms hypotheses based on programming knowledge in the top-down hypothesis-driven model of program comprehension. On the other side, Rist introduced a model for program generation based on the presentation of the knowledge in the internal/external memory where knowledge is represented as nodes, and each node represents a form of action (Rist, 1995). Nodes are indexed using tuples of forms and have four ports such as use, make, obey and control. These components together form a control/data flow and hence can be used as an alphabet of modeling the plan and more specifically, the code generation strategy of the programmer. With respect to the given definition, experts are mostly engaged with retrieving plans from their memory while novices usually engage in creating plans which includes focal expansion and backward reasoning.

Du Boulay identified a number of difficulties with learning to program (Du Boulay, 1986). He pointed out the pitfalls of using mechanistic analogies (e.g. the variable as a box), because such analogies are open to over interpretation by the novice. The organising principle of Du Boulay's paper is the *notional machine*, an abstraction of the computer that one can use for thinking about what a computer can and will do. He also discussed the pragmatics of programming that a student needs so that she can use the tools available to specify, develop, test and debug a program. Du Boulay's ideas were elaborated by subsequent researchers, including Rogalski and Samurcay (Rogalski and Samurçay, 1990), who stated:

Acquiring and developing knowledge about programming is a highly complex process. It involves a variety of cognitive activities, and mental representations *related to program design, program understanding, modifying, debugging (and documenting). Even at the level of computer literacy, it requires construction of conceptual knowledge, and the structuring of basic operations (such as loops, conditional statements, etc.) into schemas and plans. It requires developing strategies flexible enough to drive benefits from programming aids (programming environment, programming methods).*

The capabilities and behaviors of the novice programmers differ from those of the experts. Winslow states that novices are limited to the surface knowledge, have limited mental models on the programming task, use general problem solving skills rather than specific skills, and approach the program on a line by line basis (Winslow, 1996). Linn and Dalbey reported that the novices tend to spend less time testing the code (Linn and Dalbey, 1989). They are poor at tracing the code (Perkins et al., 1986), have a poor understanding of the linear execution of the code (Du Boulay, 1986), and are equipped with context specific knowledge rather than general knowledge (Kurland et al., 1986).

2.3 Data analysis

Early work focused on on identifying and measuring variables that Evans and Simkin described as being approximations for *programming aptitude* (Evans and Simkin, 1989). Before 1975, much of the research was focused on demographic factors such as educational background and previous course grades, while by the end of the 1970's, research was slowly moving towards evaluating static tests that measure programming aptitude. A more recent trend has been to investigate the effect of cognitive factors such as abstraction ability and the ability to follow more complex processes and algorithms (Evans and Simkin, 1989). These lines of research have continued to this day by introducing factors related to study behavior, learning styles and cognitive factors (Watson, Li, and Godwin, 2014).

This type of research has been motivated, among others, by (1) identification of students that have an aptitude for CS-related studies (e.g. (Tukiainen and Mönkkönen, 2002)); (2) studying and identifying measures of programming aptitude as well as combining them (e.g. (Bennedsen and Caspersen, 2006; Bergin and Reilly, 2005; Rountree et al., 2004; Werth, 1986)); (3) improvement of education and the comparison of teaching methodologies (e.g. (Stein, 2002; Ventura Jr, 2005)); (4) and identifying at-risk students and predicting course outcomes (e.g. (Jadud, 2006; Watson, Li, and Godwin, 2013)).

It has always been a sustained interest of researchers to identify performance in programming subjects as early as possible (Biamonte, 1964).

Early work attempted to predict performance based on standardized aptitude testing. Most of the variables identified in this research can be categorized according to the Bigg's model of learning, a three stage model of learning (presage, process and product) (Biggs, 1978). A set of variables related to the presage level are identifiable before the novice programmer starts the program. Such factors are prior knowledge, intelligence quotient, demographic and psychological, academic, and cognitive factors, such as: previous programming experience, maths background, science background, behavioral traits, self-esteem, learning styles, learning strategies, and attributions of success. Process factors describe the learning context, which includes student perceptions. Numerous studies have tried to find correlations between programming performance and such variables. In the next few sections, I'll review those variables.

2.3.1 Static success factors

A range of demographic, cognitive and social factors have been investigated to make models which can predict success in a computing course.

Brenda Cantwell Wilson and Sharon Shrock analysed twelve success factors including maths background, attribution for success/failure (luck, effort, difficulty of task, and ability), domain specific self-efficacy, encouragement, comfort level in the course, work style preference, previous programming experience, previous non-programming computer experience, and gender (Cantwell Wilson and Shrock, 2001). Their study revealed that comfort level, maths, and attribution to luck for success/failure contributed the most to success, where the first two of those factors had a positive influence and third factor had a negative influence. They also reported that different types of previous computer experiences (including formal programming class, self-initiated programming, internet use, game playing, and productivity software use) were predictive of success.

In 2002, researchers (Rountree, Rountree, and Robins, 2002) investigated the relationship between a set of attributes including gender, age, full-time status, year, major, keenness, recent math, background, knowledge of other languages, expected difficulty, expected workload, expected success, and the intention to continue in computer science, with performance in an introductory programming course. They reported that the group of novices which indicated an intention to continue in computer science did not perform better than others. They also reported that the strongest single indicator was the expectation that the student will "get an A from the course."

Katz et al. investigated the relationship between the score difference from pre-test to post-test and preparation (SAT score, number of previous computer science courses taken, and pre-test score), time spent on the tutorial as a whole and on individual sections, amount and type of experimentation, programming accuracy and/or proficiency, approach to materials that involve mathematical formalism, and approach to learning highly unfamiliar material (string manipulation procedures) (Katz et al., 2003). Their study included 65 students (47 male, 18 female) working through a tutorial on the basics of Perl. They reported that students who were initially the least proficient programmers, at least in Perl, seemed to benefit the most from the tutorial. They also found the predictors of course grade to be pre-test score (r = 0.28), post-test score (r =0.37), aptitude (SAT total, r = 0.40), and various measures of time (negatively correlated; e.g., total time, r = -0.30).

Bennedsen and Caspersen reported that of eight potential indicators of success (grade, math, coursework, age, major, geology major, sex, mathematics major, and non-science major) only two of those indicators were significant indicators of success in the programming course: math grade from high school and course work (Bennedsen and Caspersen, 2005).

In 2005, Bergin and Reilly carried out an investigation of fifteen factors that may influence performance on a first year object-oriented programming module (Bergin and Reilly, 2005). The factors included prior academic experience, prior computer experience, self-perception of programming performance and comfort level on the module and specific cognitive skills. They reported that a student's perception of their understanding of the module had the strongest correlation with programming performance, r = 0.76; p < 0:01. They also reported that Leaving Certificate (LC) mathematics and science scores also had a strong correlation with performance.

Ventura investigated a number of potential success factors for an OO first CS1 module (Ventura Jr, 2005). Those factors included prior programming experience, mathematical ability, academic and psychological variables, gender, and measures of student effort. It was reported that cognitive and academic factors such as SAT scores and critical thinking ability offered little predictive value when compared to the other predictors of success. On the other hand, student effort and comfort level were reported to be the strongest predictors of success.

In 2005, Raadt et al. performed a multi-institutional study to investigate the association between various factors and student performance in a first programming subject. They reported the student's approach to learning (i.e. deep vs. surface) was the strongest indicator of success for the students in the eleven participating institutes (Raadt et al., 2005).

Sheard et al. reported on the difficulty of predicting whether students will complete a unit from the students' interests and expectations of the degree. They found that students with prior knowledge of programming, English as the first language, and those who entered the degree after high school received higher results in programming. They also reported a higher drop out ratio for females (Sheard et al., 2008).

Gomez et al. reported that students' results and their personal perceptions of competence during the course have a high correlation (Gomes, Santos, and Mendes, 2012).

There are fundamental problems with using static success factors. The generation of the data required to perform the analysis for the majority of these predictors requires performing lengthy tests. As an example, Pintrich et al. used up of 80 questions (Pintrich et al., 1993). Also, these factors are not able to capture changes over time hence might not be reflecting the current (for example psychological) stage of the student.

2.3.2 Success factors and contradictory reports

Traditional data include past course outcomes, questionnaire results, course examinations or tests, as well as demographic factors such as age and gender. Whilst the use of such data is (argumentatively) popular, it may also be problematic. What if a student has not taken any courses previously, what if the student declines to answer a questionnaire, or what if the student fails a test due to focusing on another examination? A certain amount of noise is always present in such data, and, as a consequence, the importance of replication and reproduction studies increases.

In 2005, Lewis et al investigated the cross university performance and individual differences (Lewis et al., 2005) on the data collected from 165 students enrolled in CS2 courses at two different universities. Through the analysis of self-reported data, they measured experience in object-oriented processing, UNIX programming, web design and computing platforms. They also collected data to evaluate cognitive abilities measures of spatial orientation, visualization, logical reasoning and flexibility. They reported significant differences on all measures of cognitive ability and most measures of prior computer science experience and the relationship to success among the two schools. Findings of Watson and Godwin also pinpointed the tendency to yield inconsistent results (Watson, Li, and Godwin, 2013).

When considering the connections in more detail, there are studies that indicate that there is no significant correlation between gender and programming course outcomes (Werth, 1986; Byrne and Lyons, 2001; Ventura Jr, 2005), while at least one other study indicated that gender may be an explanatory factor of some course outcomes (Bergin and Reilly, 2005).

Similarly, when considering past academic performance, the results are contradictory, and vary depending on the subject. One such example is mathematics, where no significant correlation is observed in some studies (Werth, 1986; Watson, Li, and Godwin, 2014), while others have found correlations (Stein, 2002). In addition to mathematics, overall grades and, for example SAT scores have been studied as well. For example, in one study, the verbal SAT score was found to have a mediocre correlation with the introductory programming course grade (Leeper and Silver, 1982), and similarly, the university-level grade average and introductory programming course grade had a mediocre correlation (Werth, 1986). Watson et al. have also studied correlations between various secondary education courses and course averages, but found no statistically significant correlations (Watson, Li, and Godwin, 2014).

It is natural to assume that past programming experience influences programming course scores, and thus, this connection has been studied in a number of contexts. While a number of studies have reported that past programming experience helps when learning to program (Hagan and Markham, 2000; Cantwell Wilson and Shrock, 2001; Wiedenbeck, Labelle, and Kain, 2004) – typically in a tertiary education context – there are, also again, contradictory results. For example, Bergin and Reilly found that students with no previous programming experience had a marginally higher mean overall score in an introductory programming course, but found no statistically significant difference (Bergin and Reilly, 2005). In another study, Watson et al. found that while students with past programming experience had significantly higher overall course points than those with no previous programming experience (Watson, Li, and Godwin, 2014), programming experience in years had a weak but statistically insignificant negative correlation with course points (Watson, Li, and Godwin, 2014).

2.3.3 Dynamically accumulating data

Another stream of research is concerned with using data that is generated as a byproduct of students using various learning management systems. The systems may be for example learning management or course management systems, where students' course outcomes appear, or the systems may be fine-grained course-specific systems that gather data as students are working on, for example, course assignments.

Researchers have analyzed course transcripts to discover sequences that are often associated with course failure. Such information can be used to help identify course patterns that students should not take, and consequently, advise students on more suitable courses to take (Chandra and Nandhini, 2010). An example of the use of data from a learning management system is provided by Minaei-Bidgoli et al., who explored a number of classifiers to identify actions that relate to course outcomes (Minaei-Bidgoli et al., 2003). Similarly, in-class clicker behavior has been recently explored by Porter et al. (Porter, Zingaro, and Lister, 2014), who studied students’ responses in a peer instruction setting, and identified that the percentage of correct clicker answers from the first three weeks of a course was strongly correlated with overall course performance. In the work done by Rodrigo et al., a series of observations based on student’s interaction with the programming environment was used to investigate the relationship between different types of factors including affective states, behaviors and automatically dis-tillable measures and performance (Rodrigo et al., 2009a). They found that students who felt confused, bored, and those students who engaged in IDE-related on-task conversation, performed poorly in the mid semester exam.

While the previous examples are related to coarser grained data such as course results and answers to questions inside a learning management system or during a lecture, a stream of research on dynamically accumulating programming process data has recently become more popular. In 2006, Jadud suggested a metric to quantify a student’s tendency to create and fix errors in subsequent source code snapshots (i.e. compiled states) and found that there was a high correlation with the course exam score (Jadud, 2006). In essence, this suggests that the less programming errors a student makes, and the better she solves them, the higher her grade will tend to be (Jadud, 2006; Rodrigo et al., 2009b). Jadud’s finding was confirmed by the study performed by Fenwick et al. (Fenwick et al., 2009) and has also been explored in other contexts with the same programming environment and language with similar results (Rodrigo et al., 2009b; Watson, Li, and Godwin, 2013), while studies with a different programming environment have had more modest results (Carter, Hundhausen, and Adesope, 2015). Carter et al. presented the Normalized Programming State Model (NPSM), characterizing students’ programming activity in terms of the dynamically-changing syntactic and semantic correctness of their programs. The NPSM accounted for 41% of the variances in students’ programming assignment grades, and 36% of the variance in students’ final course grades. Using the output of NPSM, they presented a formula capable of predicting students’ course programming performance with 36-67% percent accuracy.

The approach proposed by Jadud has been recently extended by Watson et al. (Watson, Li, and Godwin, 2013), who included the amount of time that students spend on programming assignments as a part of the approach (Watson, Li, and Godwin, 2013). This led to an improvement over Jadud’s model. They also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with programming course scores (Watson, Li, and Godwin, 2013). More recently, a similar approach that analyzes subsequent source code snapshots was also proposed by Carter et al. (Carter, Hundhausen, and

Adesope, 2015). In addition to analyzing subsequent error states, snapshots have been used to elicit finer information. For example, Piech et al. studied students' approaches to solving two programming tasks (Piech et al., 2012), and found that the solution patterns are indicative of course midterm scores. Such patterns were also studied by Hosseini et al., who analyzed students' behaviors within a programming course – some students were more inclined to build their code step by step, while others started from larger quantities of code, and reduced their code in order to reach a solution (Hosseini, Vihavainen, and Brusilovsky, 2014). A separate stream of research was recently proposed by Yudelson et al., who sought to model students' understanding of fine-grained concepts through source code snapshots (Yudelson et al., 2014). Becker (Becker, 2016) introduced a new metric to quantify repeated errors, called the repeated error density (RED). He compared the performance of his proposed metric to Jadud's Error Quotient and showed that RED has advantages over EQ including being less context dependent, and being useful for short sessions.

There have been few direct comparisons of traditional and dynamic predictors of programming success. Watson et al. looked at a total number of 50 potential success factors comprised of 12 dynamic predictors and 38 static predictors (Watson, Li, and Godwin, 2014). The dynamic predictors were extracted from the programming logged data. In that study, seven instruments were used to collect data from the students including past programming experience and academic background, attributes of success, Rosenberg's self-esteem scale (Pintrich et al., 1993), Kolb's learning style instrument, the Gregorc style delineator, the motivated strategies for learning questionnaire (MSLQ), and a logging extension added to the BlueJ IDE to collect source code snapshot data. They reported a strong correlation between programming behavior, while observing a few strong correlations between traditional factors and performance.

Interestingly, none of the mentioned studies have focused on data analytics and the ways it could be used to increase our prediction accuracy on the success/fail ratio of the novice programmer. The majority of these studies have been focused on profiling the students progress after the data collection is completed. In this dissertation, I aim to predict the course outcome by training machine learning models which are trained on the data collected early in the semester. To do so, it is important to know what the data collection tools we have available, and what the collected data is like and what are its features. Next, I attempt to review the available data collection tools, their features, and the nature of the data collected by these tools.

2.4 Dynamic Programming Source Code Snapshot Data Collection Tools

The research reviewed in the previous subsection is based on the data collected from different programming environments. The data used in the above-mentioned studies is known as **programming source code snapshot data** where the programming tool constantly captures copies from the most recent state of the code throughout the whole programming session. These snapshots, if placed next to each other, represent a sequence of different states of the code. The following subsection provides a detailed overview on the nature of the tools used in those studies. This provides a more clear description of the data collection tools and highlights how the nature of such tools can affect the quality of the collected data.

2.4.1 Blackbox

When BlueJ (Kölling et al., 2003) version 3.1.0 was released, the Blackbox project (Brown et al., 2014) went live on 11th June 2013. Upon initial installation of BlueJ, the users were prompted with a dialog regarding participation on the Blackbox project. The prompt did not reoccur unless the user enabled it via the program options. Blackbox is a tool designed for heterogeneous purposes. Blackbox data differs in from related tools in the size of the data and its availability to the public user. Blackbox is capable of collecting start and end times of programming sessions and capturing the editing behavior of the user and the use of IDE tools. Blackbox is Java specific within BlueJ which can be a constraining factor. Also, apart from a persistent unique identifier, nothing else is known about the user. There may also be an issue in tracking a user as BlueJ uses a unique identifier per clientèle meaning if two people use a single account the data available to researchers is inaccurate. Also the clientèle may use different accounts for different machines, for example, a different account for university, work and home. BlueJ requires Internet access to record data. Therefore, any drops in Internet connectivity can lead to gaps in the data. The main novelties of Blackbox is that it is an endless data collection tool with little restriction on data size and availability.

2.4.2 CloudCoder

CloudCoder (Hovemeyer and Spacco, 2013) is open source software which supports C, C++, Java, Python and Ruby. CodingBat is a web based program from which CloudCoder was created. Instructors are able to assign students' functions which allows them to carry out an analysis on whether the students have grasped the concept. CloudCoder (Papancea, Spacco, and Hovemeyer, 2013) works by asking the user to write 5 - 15 lines of code, whose fidelity is then tested by the system. While doing this, CloudCoder has the ability to compile the submission as well as record the full text of the submission. Being able to capture line edits is an important feature of CloudCoder. When a student edits a line of code the system is able to detect a stream of insertion and deletion which are captured and recorded. CloudCoder allows detailed time - based analysis as data is recorded with millisecond timestamps.

2.4.3 CodeWorkout

CodeWorkout (Buffardi and Edwards, 2014) is an online learning tool which offers active learning drill and practice. The user is able to track progress made as well as share ideas and discuss problems using the social features present in CodeWorkout. CodeWorkout ("Adaptive and social mechanisms for automated improvement of eLearning materials") provides a comprehensive choice of exercises in the form of multiple-choice questions and coding problems, which help the novices by improving their computer science cognizance. JUnit can be used as a testing framework which allows the novice to evaluate their solutions. Apart from indicating which tests are correct, feedback is also provided in the form of hints which are based on the performance of the novice's code against test cases. CodeWorkout also offers peer review feedback and once a student has finished, they can write hints to help other peers who are struggling with the same problem. Using peer based learning is the main novelty of CodeWorkout.

2.4.4 JS-Parsons

JS-Parsons (Ihantola and Karavirta, 2011) is an interactive puzzle solving tool where the novice has to do simple code construction exercises. This is an interactive method of learning which promotes an engaging learning environment where immediate feedback is given to the students while they code. Parson's puzzles (Parsons and Haden, 2006) require novices to utilize a drag and drop style. The novice drags their choice into a certain box after which they click the check button which gives them a response on the correctness of their choice. JS-Parsons provides many choices which help the novice to improve their syntax skills. JS-Parsons also helps develop the novice's ability to model code better. The novices sometimes fail to recognize that using, for example a nested if statement would be more efficient, than using several individual if statements. However, it makes the student repeat each puzzle until they have achieved the maximum mark, which in turn can only be achieved through using model code. Being web based makes it easier to have class activity for promoting active participation.

2.4.5 PCRS [118]

Python Classroom Response System (PCRS) (Zingaro et al., 2013) is a tool designed to promote active learning through the use peer based multiple choice discussion questions. Tutors are able to receive real time feedback through code evaluation, which highlights the common errors made by the novice. PCRS has both multiple choice questions and code writing questions. PCRS is open source and free to install by any public user with Internet access. However, since the tool is entirely web based, the Internet connection is always required which means that the tool would be rendered useless during network failure. On the contrary, the work of the novice can be viewed off-line for the purposes of lab participation mark and grading. PCRS aims to promote critical thinking and analysis as well as increase classroom based interactions.

2.4.6 Problets

Problets (Kumar, 2016) core function is an online programming based tutoring system. At the moment, it supports C, C++ and Java. Problets (Kumar, 2003) has a colossal number of features. It is able to generate questions for the novice to attempt by randomly instantiating problem templates encoded into them in pseudo-BNF notation by either the tool designers or the instructor using the tool. It also has an endless inventory of questions for the novice to attempt. Problets also provides feedback to the solutions the novice provides and logs the performance of the novice which is later made available for the instructor to view. Its novel features include a mechanism to decide if a novice has mastered a particular concept or whether he or she still needs to attempt more questions to attain mastery of the topic.

2.4.7 TestMyCode

TestMyCode (TMC) (Vihavainen et al., 2013) reviews code fed into its system by the novice and provides feedback. Its ability to collect data and perform code reviews makes it a useful tool for computer science researchers. TMC has the ability to record snapshots from student's code which helps computer science researchers get a better understanding of the progression of a novice programmer. The current version of TMC is able to gather data from students' programming processes, gives and receives data regarding exercises, submits exercises to the assessment server and displays built-in scaffolding messages. Having a reward based system encourages the novice to participate more actively in class. TMC rewards the user with points not just for completing the exercise but also for completing mini - goals within the exercise. TMC has been successfully tested and utilized in a few different courses.

2.4.8 URI Online Judge

URI Online Judge (Tonin and Bez, 2013) is a tool which has been developed to give a dynamic approach to programming for the novice. It has an array of programming languages to choose from, provides real time feedback, and creates an interactive learning environment through its user interactivity feature. A novel feature of this tool is its availability in English and Portuguese (Francisco and Ambrosio, 2015). After working on the code, the novice can upload the code to see if it satisfies the given problem (Tonin, Zanin, and Bez, 2012). The user is also given various difficulty levels to select from when attempting practice problems, which enables this tool to be used by novices at different levels of proficiency.

2.4.9 UUhistle

UUhistle (Sorva and Sirkiä, 2011) is a program visualization system which is mostly used to evaluate Python statements for introductory programming education. The tool allows the novice to debug with animations which enables UUhistle to serve as a very graphical and detailed debugger. Each line of code can be animated as soon as each line of code is executed and this creates an interactive learning environment which allows the novice to better understand concepts. In addition, UUhistle has in-built stop and think example programs for the novice to answer. As with most other tools, UUhistle provides detailed feedback to the novice about their code. UUhistle can develop a visual simulation for any code it can execute.

2.4.10 Web-CAT

Web-CAT (Shah, 2003) is a web based center for automated testing. It acts as a guide for the novice by assessing the quality of his or her assignment. The tool is capable of providing immediate feedback to the novice about their assignment. Furthermore, it also supports unlimited submissions and as a result enables the novice to submit as many times as they desire until they receive a satisfying score. However, the instructor needs to provide the novice with comments on style and order. Web-CAT has a wizard based interface where upon launching the novice will be presented with a page drive where they can make specific choices. Instructors are also able to troubleshoot procedures as Web-CAT maintains an extensive log of all activities performed by users as well as providing access to their old submissions on a file server. One of the novelties of Web-CAT is its reward based system where the user is given a reward for performing multiple testing for their submissions.

2.5 Publicly Available Programming Source Code Snapshot Datasets

Among the tools mentioned in the previous section, only a few of them generate publicly available datasets. It is important to have a solid understanding of the data used in the studies mentioned in the previous subsections since this will lead to a more careful consideration for the data collection process which is used in this dissertation. These dataset are reviewed in the following section.

2.5.1 Blackbox

Blackbox is the largest repository of the programming source code snapshots (Ihantola et al., 2015). The publicly available dataset for Blackbox however has a few issues associated with it. The main concern with this repository is lack of features which are related to the context where the source code snapshots originate from. That is, the only people who have knowledge about the contextual/institutional settings of the experiments are people who set up/opt in their subjects. The context where the data comes from remains unknown to other researchers. Are the source code snapshots generated by novices or advanced students? Is the data generated in a supervised environment? What's the difficulty level of the exercises/tests/assignments and, most importantly, what are the static/traditional factors of success in programming associated with each subject? The answer to such questions is crucial to the majority of research questions related to the study of the novice programmer.

2.5.2 Code Hunt

CodeHunt is an online learning game that helps novice programmers develop fundamental programming skills. The game consists of a progression of worlds and levels, which become progressively more difficult. In each level, the player needs to find a mystery code section and compile code for it (Tillmann et al., 2014). The main concern with this repository is lack of clarity, related to the context where the source code snapshots originate from. Since this game can be played in an unsupervised environment, users may use publicly available solutions to complete a level (the solutions can be found at https://genius.com/Code-hunt-solutions-with-explanations-annotated). This in turn compromises the integrity of the publicly available datasets for researchers to analyze. Furthermore, since there is no identifier available, it is unknown whether the data generated comes from an advanced level programmer or a novice. Moreover, since this game always requires an Internet connection to run, in the event of a drop in data connectivity there is the chance of gaps in the data stored in the dataset. The scoring system of Code Hunt is based on the number lines of code rather than the quality of the code.

2.5.3 Code.Org

Code.Org is a celebrity endorsed "blocky coding" tool (Kalelioğlu, 2015), which uses a drag and drop approach to teaching programming. However, a drag and drop approach encourages the novice programmer to use trial and error rather than critical thinking to solve the puzzle at hand. Does such an approach give a real indication of whether the novice has understood the concept behind the puzzles? Furthermore, is the data gathered in a supervised environment? Since the exercises can be attempted at home, the novice is able to seek help. Therefore, is the data available for researchers to analyze generated from a novice level programmer or an advanced level programmer? In addition, the research done by kaleliouglu et al. has shown that using code.org has no effect on the reflective thinking of the novice. Is this an indicator that code.org is more of a memory based learning tool?

2.6 Approaches for Data Analysis

The approaches used for analyzing student data can be broadly categorized into two categories; (1) earlier work focussed on single- and multivariate regression analysis, while (2) more recent work has focussed on machine learning and data mining approaches. The use of single- and multivariate regression as a means to analyze correlations between a number of variables is the primary analysis method used in the literature. Numerous articles discuss the connection between a specific variable and introductory programming course outcomes (Barker and Unger, 1983; Bennedsen and Caspersen, 2006; Bergin and Reilly, 2005; Leeper and Silver, 1982; Stein, 2002; Werth, 1986; Wiedenbeck, Labelle, and Kain, 2004; Byrne and Lyons, 2001; Watson, Li, and Godwin, 2014; Tukiainen and Mönkkönen, 2002) , while less attention has been invested into multivariate analysis (Bergin and Reilly, 2006; Rountree et al., 2004; Cantwell Wilson and Shrock, 2001; Ventura Jr, 2005; Rodrigo et al., 2009a).

Educational data mining and learning analytics

The concept of *learning analytics* is a relatively new. The term was defined at the first international conference on learning analytics as being "*the measurement, collection, analysis and reporting of data about learners and their contexts, for the purposes of understanding and optimizing learning and the environments* *in which it occurs*". It is important not to confuse the definition of learning analytics with academic analytics. The term *academic* analytics entered the teaching community in 2005 but had previously been adopted by the WebCT company to describe the data collection functions enabled by course management systems. There is a major difference between the target levels of learning analytics and academic analytics. Learning analytics serves at a course level with a focus on social networks, conceptual development, discourse analysis, and intelligent curriculum, or at a departmental level focusing on predictive modeling, and the recognition of success and failure patterns, etc. Academic analytics however works on the institutional level, regional level, or national/international levels. Educational data mining, intelligent curriculum and adaptive learning are also activities that fall under the banner of learning analytics.

Machine learning and data mining approaches

The implementation of data mining methods and tools for analyzing data available at educational institutions, defined as Educational Data Mining (EDM) is a relatively new stream in data mining research. Educational data mining can be used to address a wide range of problems, such as retention of students, improving institutional effectiveness, enrollment management, targeted marketing, alumni management, and course outcome prediction.

Machine learning approaches are becoming more familiar for practitioners and researchers as they provide deep insight into the data regardless of the sample size and dimensional complexity. There have been several efforts made to evaluate different aspects of students progress by using machine learning techniques.

Data mining techniques have been used to predict course outcomes using data collected from enrollment forms (Kovačić and Green, 2010) as well as using data from students' self-assessment (Kumar and Vijayalakshmi, 2011). Techniques used range from basic machine learning algorithms to more advanced ones such as decision trees (Quadri and Kalyankar, 2010). There have been also experiments using real time data to detect student failures (Attar and Kulkarni, 2015). Abdullah et al. proposed a student performance prediction system using Multi Agent Data Mining to predict the performance of the students based on their data. Kabakchieva used a dataset of 10330 student grades categorized into different groups according to their performance (Abdullah, Malibari, and Alkhozae, 2014). Using the WEKA data mining tool, a series of classification models were trained on the pre-university based and enrollment based features extracted from student's data with the goal of predicting performance. Ramaswami and Bhaskaran used the CHAID prediction model to analyze the interrelation among variables that are used to predict the outcome of the performance at higher secondary school education (Ramaswami and Bhaskaran, 2010). They identified a set of attributes: medium of instruction, marks obtained in secondary education, location of school, living area and type of secondary education. Thai-Nghe et al. applied machine learning to predict academic performances (Thai-Nghe, Busche, and Schmidt-Thieme, 2009). Arockiam et al used the FP Tree and K-means clustering technique with the goal of discovering the similarity between urban and rural students programming skills (Arockiam et al., 2010). They identified differential skills in programming tasks between rural and urban students. Cortez and Silva used decision trees to predict failure in Mathematics and Portuguese classes using the data collected from secondary school students. They used 29 features to train four data mining algorithms including the decision tree, random forest, neural networks, and support vector machine (N = 788 students). They reported high prediction ratios for these classifiers and performed a comparison between their performances. Ramesh et al. investigated the predictive features in training classification models with a goal of predicting the grade in a higher secondary exam (Ramesh, Parkavi, and Ramar, 2013). Ahmed and Elaraby used data mining techniques to predict course outcome (Ahmed and Elaraby, 2014).

2.7 Machine Learning in CSed

Machine learning and data mining techniques have also found their way into computer science education. In 2004, researchers (Rountree et al., 2004) used a decision tree classifier to identify the combination of factors that interact to predict success or failure. They reported that students who chose to answer the survey were somewhat more likely to pass the course (73% rather than 66%) and a little more likely to get a B or better (45% rather than 39%). They also found that students who reported that they were "extremely" keen to do the course had an 85% pass rate, and students who reported that they were intending to get an A-grade had a 90% pass rate.

Bergin and Reilly used logistic regression on a dataset containing over 25 attributes to train a classifier which was able to categorize novices with an 80% accuracy ratio (Bergin and Reilly, 2006). The model was trained based on the data collected from 123 students enrolled in a first-year introductory programming module.

Blikstein et al. studied how students learn computer programming, based on data collected from 154K programming source code snapshots of computer programs developed by approximately 370 students enrolled in a CS1 course (Blikstein et al., 2014). They used methods from machine learning to discover patterns in the data and predict final exam grades.

Allevato et al (Allevato et al., 2008) mined the data collected from Web-CAT (Edwards, 2003), an automated testing system, with a goal of analyzing the relationships between the number of submissions and final score, code complexity and final score, early submission versus late submission typifying student habits, and the affect of early/late testing on the final score. Edwards et al. performed hierarchical clustering analysis on the data collected from the Web-CAT system and found that students who score well in programming tasks start the coding assignment early and finish earlier than others (Edwards et al., 2009). They also reported that approximately two-thirds of the students who scored well were those individuals who started more than a day in advance of the deadline.

Lahtinen et al. used statistical cluster analysis on novice programmers to investigate possible clusters of students in different levels of Bloom's Taxonomy (Lahtinen, 2007).

Clustering methods have also been used to distinguish behavioral patterns. For example, Berland mined source code snapshots and were able to categorize novices into planners and tinkerers (those who "*try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work*") (Berland and Martin, 2011), while in a follow up study they determined how students progress from exploration, through tinkering, to refinement (Berland et al., 2013). Becker and Mooney reviewed the application of the principal component analysis in categorizing compiler errors (Becker and Mooney, 2016). They found a new way of discovering categories of related errors from data produced by the novices during the course of their programming activity.

In a recent work, Linden and colleagues utilized novel learning analytic methods combined with learning management systems and were able to identify the students at risk three weeks before the end of a seven week programming course. They used all of the course events, such as exercise submissions, lecture attendance and weekly assignments and performed the analysis on the automatically collected data. In their work they basically implemented a support vector machine (SVM) on the data collected by the LMS system mainly the metadata of the assignment submission and class attendance.

Other studies where students' progress has been visualized have been proposed by, for example, Mabinya (Mabinya, 2005) and Blikstein (Blikstein, 2011). Mabinya discussed the possibility of applying data mining techniques and algorithms to visually model the behavior of novice programmers while they learn to code, while Blikstein described an automated technique to assess, analyze and visualize students learning computer programming (Blikstein, 2011).

All the abovementioned studies have the element of machine learning included. However, non of the mentioned studies is specific enough to focus on the novice programmer, and the task of classification, or so called supervised machine learning. Also, there are a few points that I need to highlight in this section. First, the models proposed in such studies are context dependent. That is, the proposed models suffer from dependency to the local environmental variables. Such models will fail in case they are adopted to work with the data collected from other institutes. Secondly, in majority of the cases, the goal is not to predict, but is to model. The primary difference between what is done in this dissertation and the works reviewed in this section is that the result of this dissertation is context independent, and specific to the task of prediction of performance. Nevertheless, it is important to define **performance**, and how it is measured through **assessment**. Next, I will review different means of assessment to clarify what do I mean by **performance** in this context.

2.8 Assessment

Automated assessment of the programming task was initially carried out through outcome analysis (Cooper et al., 2005), where a student's source code is tested using a series of test cases. Such methods are efficient and fully automated. However, there have been debates about using outcome analysis. The main issue with such systems manifests itself in the argument that challenges the very definition of *programming*: a task which is not only about typing the programming code, but also a way of thinking and solving problems (Marion et al., 2007). The primary concern is the operationalization of learning and knowing if such constructs in fact measure whether a student learns to code or not.

Simon et al. examined the concept of thinking like a computer scientist by analyzing students' ability to apply computer programming to real-world problems (Simon et al., 2008). They found that well before the course, students already have considerable knowledge of core concepts but not in the way a computer programmer has that same knowledge. Simon et al. reported that although students showed skills for troubleshooting problems (Simon et al., 2006), they were unable to efficiently identify problems. Soloway and Ehrlich analyzed novices to identify their programming strategies (Soloway and Ehrlich, 1984). They logged the errors encountered by the novices and noted that the majority of the errors were planning errors.

2.9 Studies of the Novice Programming Process

With a focus on errors and coding process, multiple studies have attempted to categorize novices based on their coding strategy. Perkins et al. investigated novices in a laboratory environment programming in BASIC. They categorized the students as *movers*, *tinkerers* and *stoppers* (Perkins et al., 1986). Stoppers, as the name implies, simply stop, when confronted with a problem. and "*they appear to abandon all hope of solving a problem on their own*" (Perkins et al., 1986). Movers keep trying, experimenting and modifying the code. Tinkerers are those who are not able to trace their program and they make changes at random. A total of five groups of novice programmer behaviours have been subsequently identified: followers, coders, understanders, problem solvers, and participators (Booth, 1992; Bruce et al., 2006).

Turkle and Papert also categorized novices into tinkerers and planners (without using exactly those terms) (Turkle and Papert, 1992). However, they reported that student performance can be high, regardless of which category in to which they fall. Turkle and Papert showed that planning and tinkering should not be seen as ideal attributes, but just different ways of attacking the programming task without necessarily achieving good or bad results. These findings were supported by Berland et al. in a study on a larger scale (Basawapatna et al., 2011). They aimed to see whether students recognized computational thinking patterns (Ruthmann et al., 2010).

Worsley and Blikstein presented an algorithm for studying changes in programming styles among novice programmers (Worsley and Blikstein, 2013). They also concluded that looking at changes in students' programming update characteristics may provide useful insights for studying programming proficiency, when measured by assessment.

2.10 The analysis of the novice programmer errors

Nowadays, many Interactive Development Environments (IDEs) can collect data about the novice's programming process. The data collected usually has the form of 1) metadata which reports features of different events such as compilation, project open, line edit, etc, and 2) the source code snapshot data which reports the state of the code when the abovementioned events are triggered.

Error messages are one of the most important means of communication between the programming environment and the programmer. The aim of error messages is to help facilitate the program process and, as a troubleshooting technique, help the user understand the problem underlying the error. An insufficient or incomplete error message can be misleading to the programmer who could spend an enormous amount of time spotting the source of the problem. Multiple studies have attempted to analyze the errors encountered by the novice programmers. This stream of research gives insight in the challenges that novice programmers face when learning to code. The analysis of such data also generates insights into how to improve such error messages in a way that leads to better learning outcomes.

Marceau and colleagues demonstrated that error messages in general fail to convey information accurately to students (Marceau, Fisler, and Krishnamurthi, 2011). Brown and Altadmri investigated the errors generated by novices on a large scale (N = 100K) and noticed a poor overlap between the educators' beliefs of what the most common error messages were and what they actually observed in their data (Brown and Altadmri, 2014). The frequency of the most common mistakes has also been reviewed by Peterson et al. in different contexts (Petersen, Spacco, and Vihavainen, 2015).

Other researchers have used compilation error messages to quantify students' ability to handle errors. Among the early work in studying common mistakes of novices was the work by Joni et al (Joni et al., 1983). Anderson and Jeffries studied errors that students made while writing LISP functions (Anderson and Jeffries, 1985). They reported an increase in the number of mistakes when the irrelevant aspects of the problem were increased. They reported that "*slips*" are the main reason behind the errors rather than misconceptions. They analyzed their results within a working memory cognition framework and reported that errors occur when there is a loss of information from the novice's working memory. Bonar and Soloway attempted to explain the errors generated by novices though a model (Bonar and Soloway, 1985). In the work done by Johnson and Soloway, PROUST was introduced: a tool which performs online analysis of Pascal programs written by novice programmers (Johnson and Soloway, 1985). They used a knowledge base of programming plans and strategies, together with common errors associated with them, to construct the model. They reported correlations between the intention of the novice and the errors generated by the novice. Pea reviewed a set of conceptual "bugs" and how novice programmers understand their programs (Pea, 1986). These language independent errors were observed in a range of novices with varying education levels, from high school to college. They identified three main classes of novice bugs – *parallelism*, *intentionality*, and *egocentrism*. Sleeman used a set of screening tests on 68 students from grades 11 and 12 who had just finished a Pascal unit (Sleeman, 1984). He found that over 50% of students had problems learning Pascal and he reviewed the frequency of the errors generated. In the work done by Sopher et al., a descriptive theory of buggy novice programs and an error categorization scheme were introduced (Spohrer, Soloway, and Pope, 1985). The main idea behind the presented theory was the cognitively plausible knowledge that underlies programming. Spohrer and Soloway stated that novices must be familiar with specific high frequency errors, and to learn as much as possible about the origins of all bugs (Spohrer and Soloway, 1986). They reported that just a few bugs account for most mistakes made by novices, and unlike the common belief that most errors can be attributed to the students' misconceptions about the language constructs, the majority of errors happen as a result of a plan composition problem. That is, novices struggle to put different bits of the code together to achieve a certain goal.

In 2005, Jackson and Carver gathered compilation errors from 583 students and 11 academics over a single semester, which they collected by a custom-built Java IDE (Jackson, Cobb, and Carver, 2005). They found that the top ten types of bugs accounted for more than half of the mistakes. They reported a discrepancy between the programming bugs they observed and what the faculty had believed to be the most common. Ahmadzadeh et al. collected Java compilation errors with timestamps and the source code data (Ahmadzadeh, Elliman, and Higgins, 2005). They categorized the errors into three groups – *semantic*, *logical* and *syntactic*. They reported that 36% of the errors were syntactic, 63% were semantic and only one percent were lexical. They concluded that a considerable number of students with a good understanding of programming concepts were not able to debug their code effectively. They also reported that the majority of the good debuggers also perform well in the programming task, but not all who were good in programming also performed well in debugging.

Later, Others (Jadud, 2006; Rodrigo et al., 2009b; Watson, Li, and Godwin, 2013; Carter, Hundhausen, and Adesope, 2015; Becker, 2016) extensively studied the application of the generated errors with a goal of assessing students' ability to handle the errors in the code; hence quantifying learning aptitude. Spacco et al. performed a preliminary exploratory analysis of data collected by the CloudCoder (Hovemeyer and Spacco, 2013) in multiple contexts and reported that more difficult exercises require more time and effort to complete, but are not more likely to result in compilation failures (Spacco et al., 2015).

There are many open questions about profiling students errors. I have addressed this issue in detail in the Discussion Chapter of the thesis. However, to make the dissertation easier to follow, I now highlight a few points about the problems with profiling students error. Firstly, what we call an error in one programming language may instead be defined as a warning in another language. This is more evident when comparing interpreted languages with compilation based languages. This leads to a major concern that, for the same programming task in two different languages, a novice might/will demonstrate a different error profile - which error profile, in such cases, is valid? Secondly, a considerable number of errors are due to typos. None of the mentioned studies have attempted to distinguish the typo based errors and filter them out before performing the analysis. As a result, the quality of the data itself is not guaranteed, hence any model/algorithm which is trained/designed based on such data might not be accurate. Lastly, considering a hypothetical algorithm which is truly able to capture a precise profile of students errors, the very nature of such models is dynamic. That is, the same student will probably have different error profiles if measured throughout the course and hence the generated results might not be consistent. In this dissertation, I aim to design a model which is not dependent on the errors generated by students and hence is free of such variables.

Chapter 3

Results Overview

3.1 Results Overview

This chapter introduces the reader to my research story. Chapter 5 to Chapter 12 inclusive each contain a published peer-reviewed conference/journal paper that contributes to that story. Each paper chapter is preceded by a contribution statement detailing any collaborations involved. What follows in this chapter is an overview of the story that those papers tell.
Although the work that follows is predominantly mine, the use of the pronoun "we" is used instead of "I" out of respect for those who contributed to the publications. See each of the papers (Chapter 5 to Chapter 12) and Appendix B for details of coauthors' contributions.

3.2 The Thesis About The Thesis

The main question investigated in my thesis is as follows:

Is it possible to identify differences in the source code programming data collected from the novice programmers which can lead to distinguishing those who eventually learn to program and those who don't?

To answer the above statement, I investigated the answers to two major research questions. These research questions are presented in brief in Table 3.1 and reviewed in more details in subsections 3.2.1 and 3.2.2.

TABLE 3.1: Research Questions, the thesis about the thesis, and the corresponding chapters

Research Question	**Description**	**Chapters**
RQ1	Is it possible to identify struggling students by analyzing the source code snapshot data?	5, 6, 7, 8, 9
RQ2	Can we address the problem of the sensitivity of the prediction (of the struggeling novice programmers) in a more context-independent manner?	10, 11, 12

3.2.1 Research Questions 1

Research question one is mainly concerned with the quantitative metrics derived from the source code snapshot and their associations with students' performance. I have investigated this matter in two different contexts where the major difference between the two contexts is the language used to teach novices: Java, and SQL. Throughout Chapters 5, 6, 7, 8 and 9, I demonstrate how the data collected from the novices can be used to **a.** review associations between different programming tasks, and **b.** perform machine learning tasks to identify course outcome.

Paper 1. Chapter 5: Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?

What we have demonstrated in this paper is that advocates of the various hypotheses – Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum – can all find support for their respective hypotheses, in aspects of the data in this paper. This paper demonstrates, among other things, that there are substantial differences between students in their programming ability as early as the third week of semester, using test questions based only on the concept of assignment.

Paper 2. Chapter 6: Exploring machine learning methods to automatically identify students in need of assistance

This work is a starting point for using machine learning techniques on naturally accumulating programming process data. In this work, we show that the information obtained from the novices can be used in a machine learning context to identify students' performance in course outcomes. When combining source code snapshot data that is recorded from students' programming process with machine learning methods, we are able to detect high and low-performing students with high accuracy after the very first week of an introductory programming course. In this paper, two primary *types* of information are collected from the novices' source code snapshot data: **a.** the performance in completing lab exercises, and **b.** the number of steps/attempts taken during the programming task.

Paper 3. Chapter 7: A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries

In this work, we extend the context to another language, the SQL. Data collected on large scale for around 2300 students is analyzed with a focus on association analysis of students' performance writing different types of SQL statements. Programming in Java and writing SQL *SELECT* statements do not have high convergence, however they can both be viewed from the same angles: performance and number of steps/attempts. In this study we show that there is a high negative correlation between the number of attempts spent on writing simple SQL **SELECT** statements and performance in writing harder statements. Results of this paper supports what we found in the result of the previous paper: novices who require more attempts to complete a simple programming task tend to perform poorer in more complicated assessments.

Paper 4. Chapter 8: Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries

In this study we attempted to analyze the SQL *SELECT* statements generated by novices in more detail. This paper aims to explore what we could understand from the content of the data generated by the novices. We attempted to understand the path that leads a novice to construct a correct statement. The analysis of the data collected from novices revealed that both semantic and syntactic mistake are inevitable and happen through the process of the construction of the correct SQL statement.

Paper 5. Chapter 9: Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success

This study investigates the correctness and the compilation result of novice SQL statements with the aim of identifying common mistakes made by novices. We also studied how these mistakes predicted performance in other assessments. The result of this study demonstrated that it is possible to predict students' performance in assessments based on the number of steps, and correctness of the SQL *SELECT* statements.

3.2.2 Research Question 2

This research question is mainly concerned with the element of context. To put it in other words, the aim is to extend the research scope to understand caveats of data, data derived models, and the application generality to see if machine learning techniques can indeed be used to correctly predict performance of the novices. To explore the answer to this research question, I investigate the sensitivity of the prediction outcome towards changes in context. Based on the findings, I propose a machine learning based method which is capable of analyzing the data in a more context-independent fashion. I explore the application of this method in association analysis of novices' performance . Chapters 10, 11 and 12 review these findings.

Paper 6. Chapter 10: Performance and Consistency in Learning to Program

In this work, we analyze students' performance and consistency with programming assignments in an introductory programming course. We demonstrate how performance, when measured through progress in course assignments, evolves throughout the course, explore weekly fluctuations in students' work consistency, and contrast this with students' performance in the final course exam. Our results indicate that whilst fluctuations in students' weekly performance do not distinguish poor performing students from well performing students with high accuracy, more accurate results can be achieved when focusing on the performance of students on individual assignments which could be used to identify struggling students.

Paper 7. Chapter 11: On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions

In this chapter, I propose a method for analyzing programming source code snapshot data which is not dependent on the static success factor. The method is mainly concerned with the number of attempts/steps spent on a programming task, and aims to investigate the association of this element with the course outcome. This method in particular is concerned with identification of high/poor performing students.

Paper 8. Chapter 12: A Contingency Table Derived Methodology for Analyzing Course Data

This journal paper extends the scope of the method for data analysis presented in Chapter 11 by introducing new metrics which could be derived from the contingency tables.

Chapter 4

Method

4.1 Introduction

In this chapter, I review the data collection tools, different features of the generated data, and the context in which the data was generated. Section 4.2 reviews the concept of the programming source code snapshot. In sections 4.2.1 and 4.2.2. I review both the context and different aspects of data collection.

4.2 Background

Data is the fuel of educational data mining and learning analytics. From the survey of research projects involving routine data collection from CS learners, it is clear that there are wide variations in the nature and granularity of data collected by different researchers. Collecting and preprocessing data also plays a key role in verifying experimental findings. Hence, I will describe the state of the art in data collection of programming traces as well as the differences in various data collection approaches, including the granularity of the data which describes how novices have solved programming problems. From the survey of research projects involving routine data collection from CS learners, it is clear that there are wide variations in the nature and granularity of data collected by different researchers.

Figure 4.1 shows a general way of characterizing a student's interaction with a software tool or system while they work on a programming problem. Each interaction can be thought of as an *event*, consisting of some action performed by the student, along with any feedback or results that are received in response. At the same time, the student is interacting with tools in order to *construct a solution* such as a function or program. That means that, in addition to the action performed by the student, there is also the current *state* of the solution that is being constructed. Data collected by various research projects while computing students work on course activities involves capturing one or both of these entities: the events, with or without the associated feedback, and/or the state of the solution being constructed.

FIGURE 4.1: An abstract view of the logical services where data instrumentation and collection are typically performed. Note that some data collection tools encompass multiple logical services (Adapted from Ihantola et al., 2015).

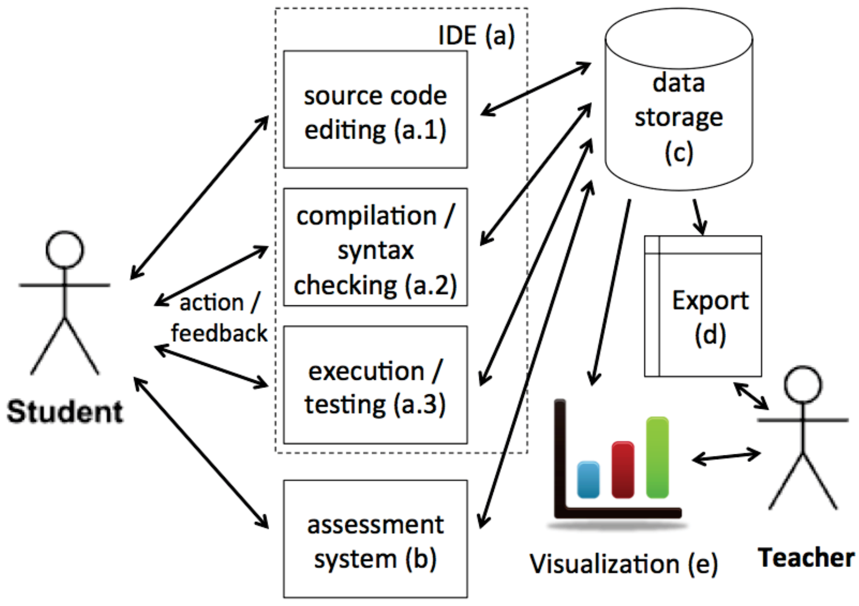
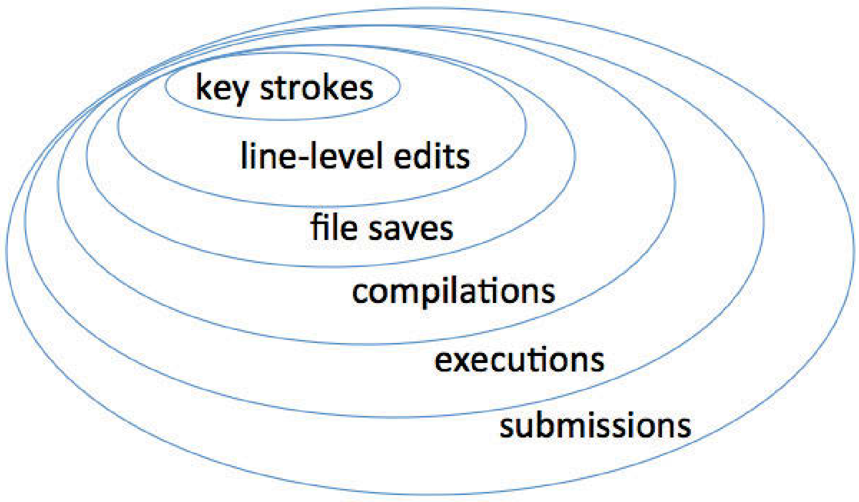


Figure 4.2 depicts the most common points on the granularity spectrum from smallest (individual key strokes) to largest (complete assignment submissions to some form of assessment or feedback system). In many cases, differences between the data collected and used by different researchers can be characterized by describing which point(s) on the granularity continuum were chosen, and whether event actions, event actions plus feedback, or the states of the solution were captured.

Researchers have commonly used different varieties of tools for systematic collection of student data:

- *Automated grading systems* – tools used to collect and process student work that is presented for assessment are commonly used in data collection. These systems typically result in data sets at the granularity of *submissions* (a complete solution state representation, usually identified by the student as ready for evaluation). Although full event information about the student's submission action and the associated feedback received may be recorded, a significant limitation is the relative sparseness of these events, and the lack of visibility into solution states or actions between submission events. Web-CAT (Edwards and Perez-Quinones, 2008) is a typical example of a tool used for this kind of data collection.

FIGURE 4.2: Data can be collected at different levels of granularity, which implies different collection frequencies and associated data set sizes (Adapted from Ihantola et al., 2015).



- *IDE instrumentation* – tools used to collect individual events within a student's IDE usually focus on "project"-level events, including file saving, compilation, and execution. Some systems, such as HackyStat (Johnson et al., 2004), focus more on recording information about events, while others, such as Marmoset (Spacco et al., 2006), focus on capturing snapshots of the solution being constructed. Some web-based programming tools also collect data at this level of granularity.

- *Version control systems* – some systems that focus on the state of the solution only, rather than event tracking, use version control systems to store histories of snapshots of the code being developed. This can be done with voluntary source code commitments when students are actively using source code control on their projects, or by using automated instrumentation to transparently commit the state of the source code to a version control system.

- *Key logging* – at the finest level of granularity, some systems track and store events at the level of individual keystrokes. Effectively, the working environment has to be augmented with a keylogger. This technique appears to be more common in web-based problem solving environments at present, as well as newer IDE instrumentations. Here, there are also multiple levels of granularity — some contexts store progress, e.g. during a pause, leading to small bulk inserts, while others store each event individually.

In the next three subsections, I review the data used to generate the results in different chapters of this PhD thesis.

4.2.1 Programming source code snapshots, collected at the University of Helsinki.

One set of data was collected from two semesters of an introductory programming course organized at the University of Helsinki. The course lasts six weeks, is taught in Java, and uses a blended online textbook that covers variables, basic I/O, methods, conditionals, loops, lists, arrays, elementary search algorithms and elementary objects.

In this programming course, the main focus is on working on practical programming assignments, accompanied by a weekly two-hour lecture that covers the basics needed to get started with the work. Support is available in open computer labs, where teaching assistants and course instructors are available some 20-30 hours each week.

Students work on a relatively large number of programming assignments each week. In the first week, an assignment may be for example "*Create a program that reads in two integers and prints their sum*".

Although no socioeconomic factors were available, the studied population is relatively homogenic, and the educational system in the context is socially inclusive, meaning that there is both a minimal under representation of students from a low educational background and a minimal over representation of students from a high educational background (Orr, Gwosć, and Netz, 2011). There are also no tuition fees, and students receive student benefits such as direct funding from the state, assuming they progress in their degree work.

The students' programming process was recorded using a system called *Test My Code* (Vihavainen et al., 2013) that is used for automatically assessing students' work in the course. For each student that consented to having their programming process recorded, every key-press and related information such as time and assignment details was stored. The students used the same programming environment both from home and at the university. Students were asked to provide information on whether they had prior programming experience. Access to information on students' age, gender, grade average, and major was given. In the studied context, major is selected before enrollment, and in both semesters, over 50% of the students had subjects other than computer science as their major.

In the first semester, a total of 86 students participated in the data collection, and in the second semester (fall), a total of 210 students participated in the data collection. Full fine-grained key-log data is available only for the first semester, while for the second semester, only higher level actions such as saves, compilation events, run events and test events are available. As the educational system in Finland forces students to choose their major at the time they apply to the university, the introductory programming course is typically the first course for the CS students, while students who major in other subjects often take the course later in their studies. However, many students with CS as a major have completed the course already before starting their university studies.

While attendance in the course activities is not mandatory, 50% of total course points comes from completing programming assignments. The rest of the course points comes from a written exam, where students answer both essay-type questions as well as programming questions. To pass the course, the students have to receive at least half of the points from the exam as well as half of the points from the programming assignments, while the highest grade in the course can be received by gathering over 90% of the course points.

A full list of the assignments used by TestMyCode is available at http://mooc.fi/courses/2013/programming-part-1/material.html/. Appendix F reviews the topics and the course structure for the subject Introduction to Programming at University of Helsinki. Details of the material and assignments used in the first week of the semester is presented in Appendix I. Due to limitations in page count of this thesis, the whole material used in other weeks of the semester were not used in the body of this thesis.

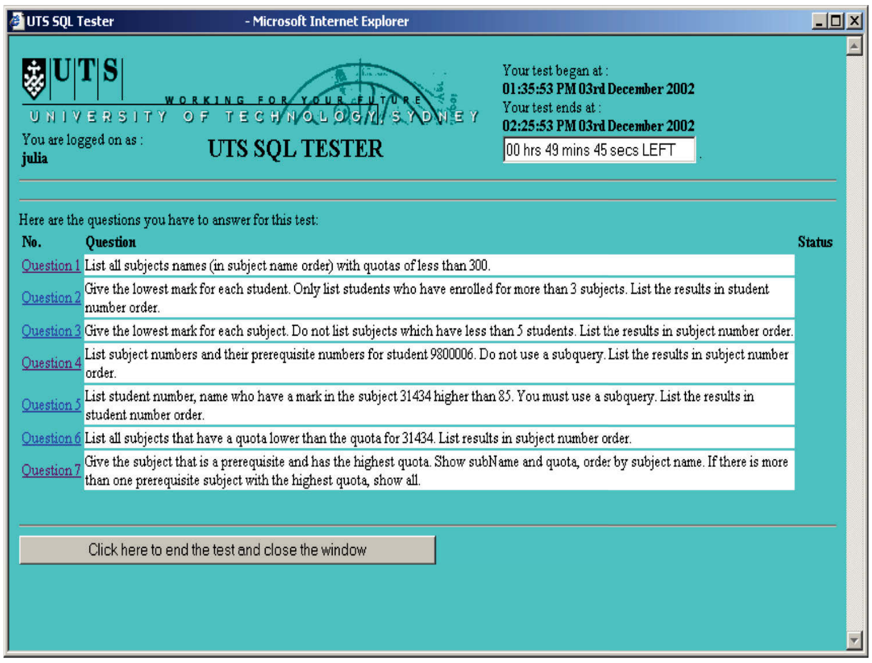
4.2.2 Database source code snapshots, collected at the University of Technology Sydney.

Data collected from a database course has fundamental differences with the data collected from a programming course. However, the data collected from the *SELECT* statements generated by novices has interesting topical convergence which makes it of interest for analysis. In regards to the scope of this thesis, each SQL *SELECT* statement can be either correct, incomplete, or syntactically incorrect. While the nature of these statements is non-procedural, the expansion of the initial template by which the novice starts the construction of the *SELECT* statement is similar to the idea of chunking in programming. Regardless of the content, the initial template, the development and the correctness status of a SQL *SELECT* statement compared to a piece of code in Java, could be seen on the same level: both datasets can be used to extract the same details which is in the data analytics process. However, the fundamental differences in context is the main reason for its inclusion in this thesis: Can I integrate the findings from data collected in a programming context with data collected in a database context?

This data was collected in a purpose-built online assessment system, AsseSQL (Prior and Lister, 2004) (See Appendix E for more information). The students in this study were all undergraduates at UTS, studying Bachelor degrees in Information Technology or Software Engineering.

In the online test, students are allowed 50 minutes to attempt seven SQL questions. On their first test page in AsseSQL, students see all seven questions 4.3, and they may attempt the questions in any order. All seven questions refer to a "scenario" database that is familiar to the students prior to their test. An example of such a scenario database is given in Figure 4.4 as an entity relationship diagram (ERD). Each question is expressed in English, and might begin, "Write an SQL query that ...". A student's answer is in the form of a complete SQL *SELECT* statement.

FIGURE 4.3: Student's home screen at AsseSQL. Students may attempt questions in any order, as many times as they wish.


UTS SQL Tester - Microsoft Internet Explorer
UTS
WORKING FOR YOUR FUTURE
UNIVERSITY OF TECHNOLOGY, SYDNEY
Click here to end the test and close the window

You are logged on as :
julia

UTS SQL TESTER

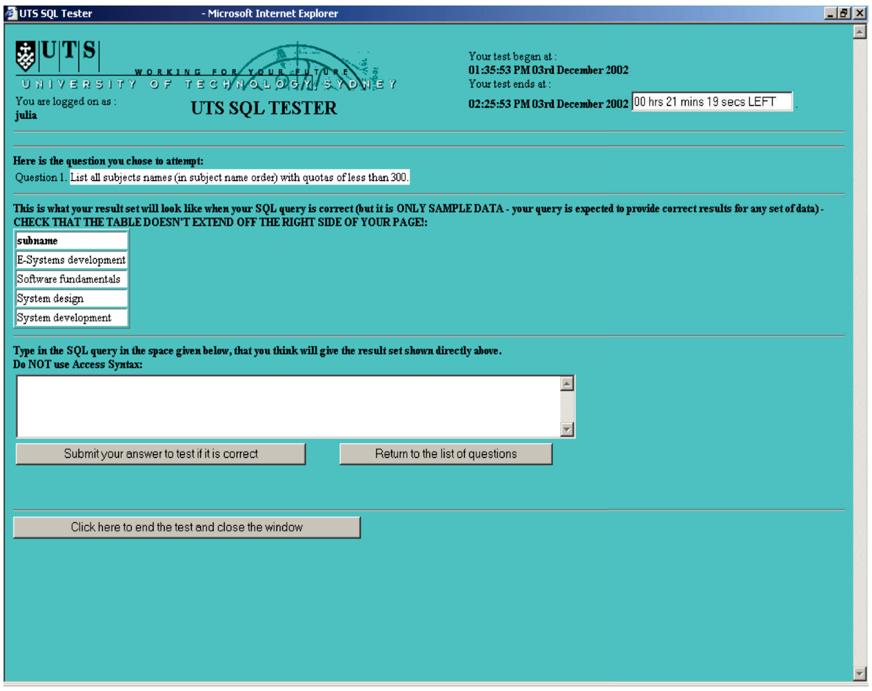
Your test began at :
01:35:53 PM 03rd December 2002
Your test ends at :
02:25:53 PM 03rd December 2002
00 hrs 49 mins 45 secs LEFT

Here are the questions you have to answer for this test:

No.	Question	Status
Question 1	List all subjects names (in subject name order) with quotas of less than 300.	
Question 2	Give the lowest mark for each student. Only list students who have enrolled for more than 3 subjects. List the results in student number order.	
Question 3	Give the lowest mark for each subject. Do not list subjects which have less than 5 students. List the results in subject number order.	
Question 4	List subject numbers and their prerequisite numbers for student 9800006. Do not use a subquery. List the results in subject number order.	
Question 5	List student number, name who have a mark in the subject 31434 higher than 85. You must use a subquery. List the results in student number order.	
Question 6	List all subjects that have a quota lower than the quota for 31434. List results in subject number order.	
Question 7	Give the subject that is a prerequisite and has the highest quota. Show subName and quota, order by subject name. If there is more than one prerequisite subject with the highest quota, show all.	

When a student submits a *SELECT* statement for any of the seven questions, the student is told immediately whether their answer is right or wrong. If the student's answer is wrong, the system presents both the desired table and the actual table produced by the student's query (see Figure 4.5). The student is then free to provide another *SELECT* statement, and may repeatedly do so until they either answer the question correctly, run out of time, or choose to move on to a different question. A database SQL *SELECT* statement source code snapshot is generated every time the novice presses the submit button. If a student moves to a new question without having successfully completed the previous question, the student may return to

FIGURE 4.4: Screen for an individual question. The question and the simple output to clarify the question are presented for each question in the test.


UTS SQL Tester - Microsoft Internet Explorer
UTS
WORKING FOR YOUR FUTURE
UNIVERSITY OF TECHNOLOGY, SYDNEY
Submit your answer to test if it is correct
Return to the list of questions
Click here to end the test and close the window

You are logged on as :
julia

UTS SQL TESTER

Your test began at :
01:35:53 PM 03rd December 2002
Your test ends at :
02:25:53 PM 03rd December 2002 00 hrs 21 mins 19 secs LEFT

Here is the question you chose to attempt:
Question 1. List all subjects names (in subject name order) with quotas of less than 300.

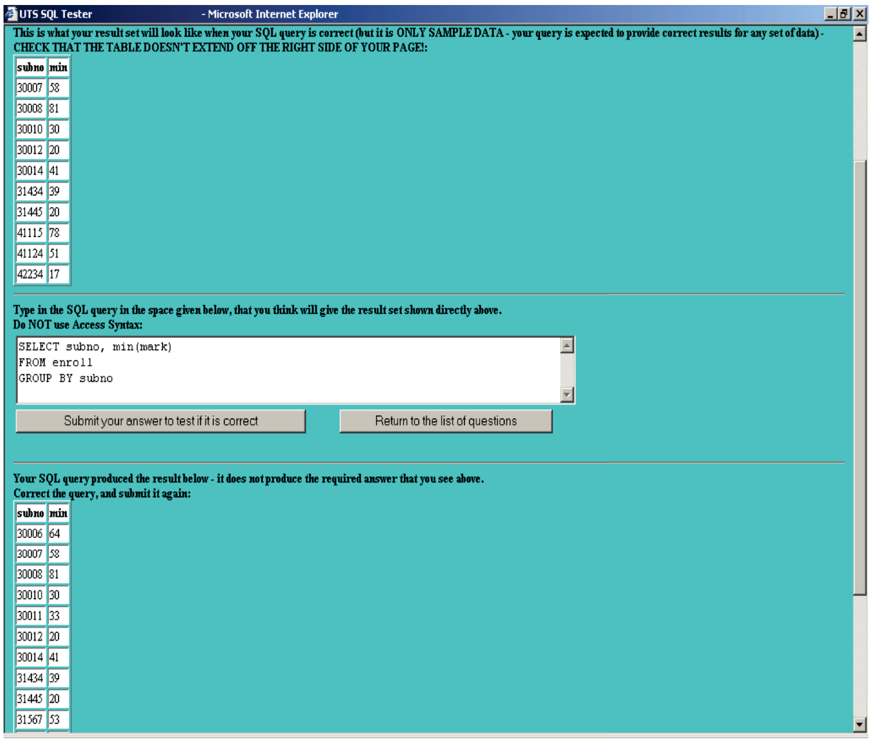
This is what your result set will look like when your SQL query is correct (but it is ONLY SAMPLE DATA - your query is expected to provide correct results for any set of data) - CHECK THAT THE TABLE DOESN'T EXTEND OFF THE RIGHT SIDE OF YOUR PAGE!:

subname
E-Systems development
Software fundamentals
System design
System development

**Type in the SQL query in the space given below, that you think will give the result set shown directly above.
Do NOT use Access Syntax:**

that question later (see Figure 4.6).

FIGURE 4.5: Feedback screen for an individual question with the output of model answer as well as the student's answer are presented in this page.


UTS SQL Tester - Microsoft Internet Explorer
Submit your answer to test if it is correct
Return to the list of questions

This is what your result set will look like when your SQL query is correct (but it is ONLY SAMPLE DATA - your query is expected to provide correct results for any set of data) - CHECK THAT THE TABLE DOESN'T EXTEND OFF THE RIGHT SIDE OF YOUR PAGE!:

subno	min
30007	58
30008	81
30010	30
30012	20
30014	41
31434	39
31445	20
41115	78
41124	51
42234	17

**Type in the SQL query in the space given below, that you think will give the result set shown directly above.
Do NOT use Access Syntax:**

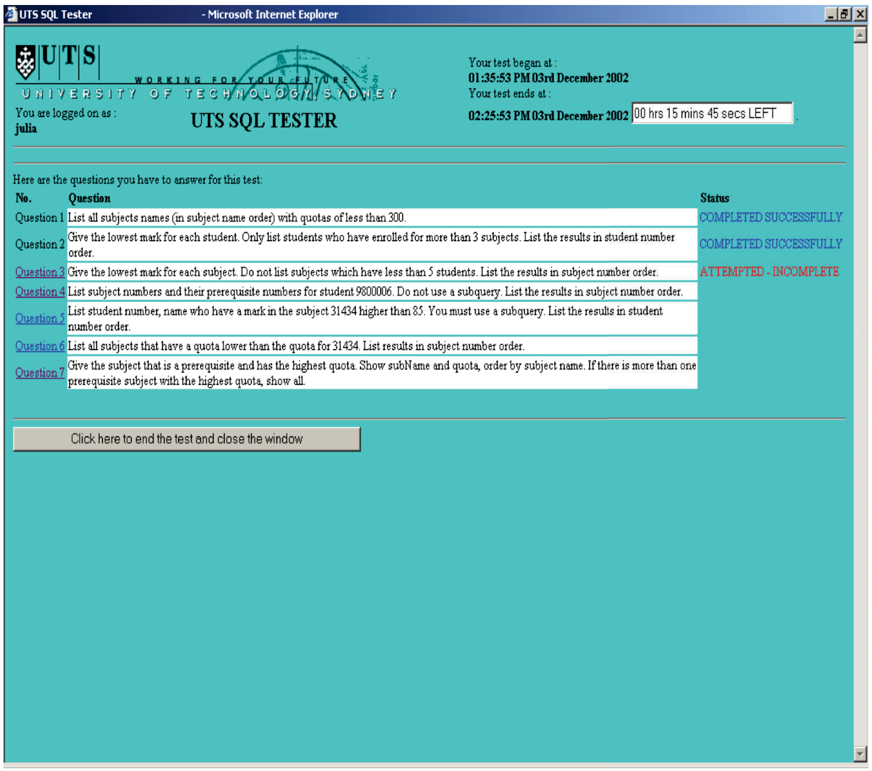
```
SELECT subno, min(mark)
FROM enroll
GROUP BY subno
```

**Your SQL query produced the result below - it does not produce the required answer that you see above.
Correct the query, and submit it again:**

subno	min
30006	64
30007	58
30008	81
30010	30
30011	33
30012	20
30014	41
31434	39
31445	20
31567	53

The grading of the answers is binary – the student's attempt is either correct or incorrect. As there may be more than one correct SQL statement for a specific question, a student's SQL statement is considered to be correct when the table produced by the student's *SELECT* statement matches the desired table. (Some simple 'sanity checks' are made to ensure that a student doesn't fudge an answer with a brute force selection of all required rows.)

FIGURE 4.6: Back to the student's home screen: students may return to this at anytime.


UTS SQL Tester - Microsoft Internet Explorer
UTS
WORKING FOR YOUR FUTURE
UNIVERSITY OF TECHNOLOGY, SYDNEY
Click here to end the test and close the window

You are logged on as :
julia

UTS SQL TESTER

Your test began at :
01:35:53 PM 03rd December 2002
Your test ends at :
02:25:53 PM 03rd December 2002 00 hrs 15 mins 45 secs LEFT

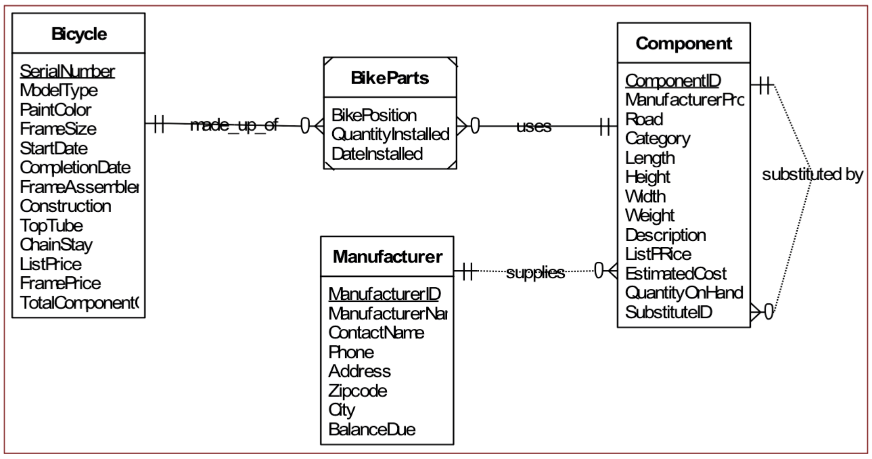
Here are the questions you have to answer for this test:

No.	Question	Status
Question 1	List all subjects names (in subject name order) with quotas of less than 300.	COMPLETED SUCCESSFULLY
Question 2	Give the lowest mark for each student. Only list students who have enrolled for more than 3 subjects. List the results in student number order.	COMPLETED SUCCESSFULLY
Question 3	Give the lowest mark for each subject. Do not list subjects which have less than 5 students. List the results in subject number order.	ATTEMPTED - INCOMPLETE
Question 4	List subject numbers and their prerequisite numbers for student 9800006. Do not use a subquery. List the results in subject number order.	
Question 5	List student number, name who have a mark in the subject 31434 higher than 85. You must use a subquery. List the results in student number order.	
Question 6	List all subjects that have a quota lower than the quota for 31434. List results in subject number order.	
Question 7	Give the subject that is a prerequisite and has the highest quota. Show subName and quota, order by subject name. If there is more than one prerequisite subject with the highest quota, show all.	

Prior to taking a test, the students are familiarized with both AsseSQL and the database scenario that will be used in the test. About a week before the test, students receive the Entity Relationship Diagram (ERD) (See Figure 4.7 for an example of such ERDs used) and the *CREATE* statements for the scenario database. The sample practice ERD and its *CREATE* statements are included in Appendix G. Note, however, that students are not provided with the data that will fill those tables, nor are they provided with sample questions for that database scenario. Several weeks before the actual test, students are provided with access to a 'practice test' in AsseSQL, which has a different scenario database from the scenario database used in the actual test.

A more detailed description of the online test software, AsseSQL, follows. All the data about each test to be taken are stored in a database, for

FIGURE 4.7: A sample of the ERD used in the test.



example, test date, duration, total number of marks, number of questions and type of SQL query to be tested in each question; in other words, the design of the test. Also stored in this database is a query pool (See Appendix G)– a selection of SQL problems and model answers (i.e. queries) that test different types of SQL statements. The structure of each test is such that although all the students in a class will do Test 1, for instance, each student will be given their own unique version of Test 1 when they actually take the test, as questions for each student are chosen at random from the pool. In the query pool, there are a number of problems that could be used for different question types. When a particular student logs on to do the test, the program chooses one of these queries for different questions of this student's test, and similarly for each of the other questions in other students' tests. A second, 'scenario' database contains the tables against which both the model solutions (queries) and the students' attempts for each test question are executed. For example, there might be an Order Entry database containing Customer, Product and Order tables for Test 1. The questions for a test would require queries to be constructed for querying data stored in this scenario database.

The data collected from AsseSQL is from tests run over the last 10 years and contains around 163000 SQL *SELECT* statement snapshots from around 2300 novices. More details about the data is presented in Chapters 7, 8, and 9.

Chapter 5

Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?

5.1 Introduction

This paper aims to find evidence for different theories for explaining why novices struggle with programming. In this paper, we demonstrate that our results can be interpreted in a variety of ways, hence different interpretations can be obtained from the data. In this paper we stress that the design of the experiment. Our findings in this paper shed light on the experiment design and the data analysis methods that motivated the studies in subsequent chapters of the thesis.

5.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;
2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;
3. there are no other authors of the publication according to these criteria;
4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and

5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?

Conference: International Computing Education Research conference 2013

URL: http://dl.acm.org/citation.cfm?id=2493416&dl=ACM&coll =DL&CFID=862572254&CFTOKEN=60379727

Status: Presented, August 2013

TABLE 5.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 5

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Raymond Lister	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi

Name Signature Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister		
Name	Signature	Date

5.2 PDF of the Published Paper

Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?

Alireza Ahadi
University of Technology, Sydney
Australia
Alireza.Ahadi@uts.edu.au

Raymond Lister
University of Technology, Sydney
Australia
Raymond.Lister@uts.edu.au

ABSTRACT

Computing academics report bimodal grade distributions in their CS1 classes. Some academics believe that such a distribution is due to their being an innate talent for programming, a “geek gene”. Robins introduced the concept of learning edge momentum, which offers an alternative explanation for the purported bimodal grade distribution. In this paper, we analyze empirical data from a real introductory programming class, looking for evidence of geek genes, learning edge momentum and other possible factors.

Categories and Subject Descriptors

K.3.2 [**Computing Milieux**]: Computers and Education - Computer and Information Science Education

General Terms

Measurement, Human Factors.

Keywords

Learning edge momentum, programming, CS1, assessment, bimodal grade distribution.

1. INTRODUCTION

Some computing academics claim that students are either born with an innate ability to program, the “geek gene”, while other students are doomed to struggle with programming. As justification, those computing academics claim to see bimodal distributions in CS1 grades.

Robins (2010) provided an alternate explanation for why CS1 grades might show a bimodal distribution. Robins introduced the concept of learning edge momentum, which works as follows. Consider a CS1 course as consisting of a sequence of topics. Suppose that two students have an equal probability of learning the nth topic but only one of them successfully learns that topic. In Robin’s model, the successful student is then a little more likely than the other student to learn topic n+1. That is, the student who learnt topic n gains a little momentum, while the student who did not learn topic n loses a little momentum. Robins constructed a computer model and showed that his model leads to bimodal grade distributions. A crucial feature of Robins’ model is that all the simulated students begin with an equal probability of learning the first topic. Thus, Robins’ simulations demonstrate that a bimodal grade distribution is not necessarily proof for a geek gene.

Apart from geek genes and learning edge momentum, there are other possible explanations for the purported bimodal grade distribution. For example, perhaps some students enter CS1 with prior programming knowledge, or superior study skills. Another possibility is what we, the authors of this paper, call “stumbling points”. These are a small number of identifiable skills and concepts (perhaps but not always threshold concepts) that can have a major impact on a student’s progress.

These hypotheses − geek genes, prior knowledge, stumbling points and learning edge momentum − are not mutually exclusive. For example, perhaps prior knowledge produces a small difference in students at the start of semester, and that difference is then amplified by learning edge momentum.

These hypotheses place differing emphasis on different points of time and different periods of time. Geek genes and prior knowledge, of course, emphasize the time before the student commenced CS1. The distinction between stumbling points and learning edge momentum is more subtle, and overlapping. The idea of stumbling points is clear cut when there appear to be a small number of identifiable difficulties that students have with learning to program, whereas learning edge momentum is clear cut when there are a large number of subtle difficulties spread uniformly across a semester.

The over-arching discussion point of this paper is this: *What types of empirical data, collected from real students, might distinguish between the competing hypotheses for the purported bimodal grade distribution*? Unfortunately, we cannot answer that question in this paper (if we could, then this would not be a discussion paper). Instead, we will use data we have collected to highlight the difficulty of choosing between the competing hypotheses. Our data is from a real introductory programming class, collected from four tests held at weeks 3, 4, 6 and 7 of semester.

2. TESTS 1 & 2 (WEEKS 3 & 4)

The introductory programming course in which these tests were conducted had a two hour lecture in each of the 13 weeks of semester. Each week, commencing from week 2, there was also a two hour lab and a one hour recitation (known as a tutorial in some countries).

Students completed each of these tests at the start of a lecture session. Every test question, or where applicable every part of a question, was graded as being either right or wrong – no fractional points were awarded.

ICER'13, August 12–14, 2013, San Diego, California, USA.
Copyright © 2013 ACM 978-1-4503-2243-0/13/08…$15.00.
http://dx.doi.org/10.1145/2493394.2493416

Completion of the tests did not contribute to a student's final grade, and was voluntary. However, as the lecture did not proceed until a test was over, most students completed the tests. The time students took to complete a test was not formally recorded, but each test took around 15 minutes. Students were under little time pressure. The lecturer asked for a show of hands on who needed more time, and the tests usually did not finish until no student raised their hand. There was rarely an unanswered question in the students' submissions.

2.1 Test 1 (Week 3)

When the students did test 1 in week 3, they had completed 4 hours of lectures, 2 hours of labs and 1 hour of recitation. The material in the week 3 test was taught in the previous week's lecture, but at the time of the test the students had not done their recitation and lab supporting that week 2 lecture.

All the questions in test 1 were about assignment statements. The most complex code presented to the students comprised three assignment statements that swapped the values between two variables. The students were required to supply a total of nine answers, which were all marked as either right or wrong. Of the nine answers, 5 required the students to trace code, 3 required the students to explain code, and 1 required the students to write code. The actual questions are provided in the appendix, along with a breakdown of how well the class answered each of the nine parts. Figure 1 shows the distribution of student scores on test 1.

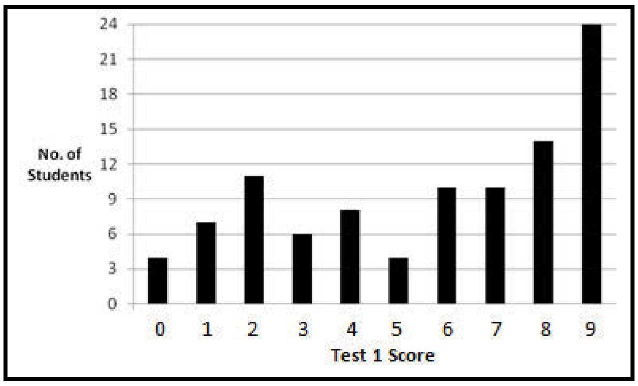


Figure 1. Distribution of student total scores on test 1 (N=98).

2.1.1 Discussion Points on Test 1

In Figure 1, the number of students peaks at two places, test 1 scores 2 and 9 − is it therefore a bimodal distribution? Either the distribution is bimodal or it is not. If it is bimodal, and if Figure 1 is indicative of CS1 classes in general, then the hypothesis of learning edge momentum is unnecessary – any bimodal distribution observed at the end of semester was simply there from the start of semester. Thus advocates for geek genes or prior knowledge might claim that Figure 1 supports their position.

Computing academic who claim that Figure 1 is NOT a bimodal distribution, but who claim to see a bimodal distribution in their end of semester grade distribution, are faced with a conundrum – how do they justify their claim that one distribution is bimodal, while the other is not? As Schilling, Watkins, and Watkins (2002) explain, it is risky to claim a bimodal distribution from merely observing two peaks. The authors of this paper believe that the distribution in Figure 1 could equally be characterized as a noisy uniform distribution with a ceiling effect – which leads the authors of this paper to wonder whether many alleged bimodal distributions of end-of-semester CS1 grades should also be classified that way.

Irrespective of whether or not the distribution in Figure 1 is bimodal, Figure 1 certainly does show a wide variation in student scores. If, as in Robins' model, all our students had an equal chance of answering each of our questions correctly, we would expect to see a distribution of total scores showing a central tendency, which is clearly not the case. Figure 1 suggests that either (1) the students differentiated very, very quickly, or (2) the students were a heterogeneous group from the beginning. Two possible explanations for the large variation in week 3 scores are:

- **Genetics:** Whether any genetic difference relates to general intelligence, or a more specific "geek gene", it seems unlikely that a college student needs to be especially bright or especially geeky to successfully reason about assignment statements.
- **Prior knowledge:** Approximately 30% of our students reported prior experience in programming, which may account for 39% of the class scoring 8 or 9 on test 1, but it does not account for the wide distribution of test scores for 7 or less.

2.2 Test 2 (Week 4)

The second test was conducted one week later. As with the first test, test 2 only used assignment statements. Test 2 was short, comprising only three questions with no subparts. The first question required students to trace some code which was (apart from changes to variable names and initial values) the same type of question as question 1(d) in the first test. The second question required students to write a swap of two variables, like question 3 in test 1. The third question was relatively novel, requiring students to write code for a swap-like process among four variables. The actual questions are provided in the appendix, along with a breakdown of how well the class answered each question. Figure 2 shows the distribution of student total scores on this second test.

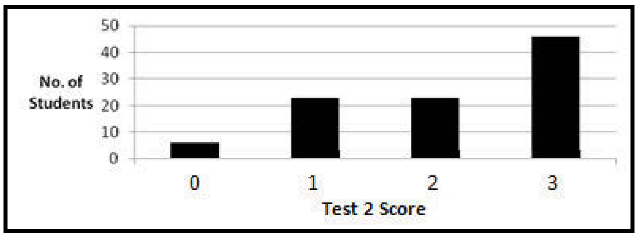


Figure 2. Distribution of student total scores on test 2 (N=98).

3. COMPARING TEST 1 AND TEST 2 Q3

In test 2, the first two questions reprised questions from test 1. Only the third question involved a problem that students had not encountered before, and so this question was considered to be the question of most interest.

Figure 3 plots the probability of students answering test 2 question 3 correctly, versus their total score on test 1. The diameters of the circles in this figure reflect the number of students who attained those respective scores on test 1. For example, the largest circle represents the 24 students who attained a perfect 9 on test 1. The smaller circle to its left represents the 14 students who scored 8 on test 1. The two smallest circles each represent 4 students.

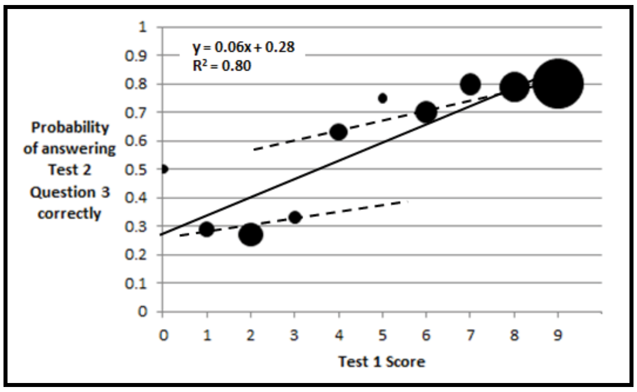


Figure 3. The relationship between student scores on test 1 and the probability of answering Test 2 Q3 correctly (N=98).

The regression line shown in Figure 3 (i.e. the solid line) is weighted according to how many students are represented by each circle. This was done by performing the regression with 24 data points for test1 score = 9 and probability = 0.80, 14 data points for test1 = 8 and probability = 0.79, and so on, down to test1 score = 0 and probability = 0.5. All regression lines in subsequent figures were calculated this way.

3.1.1 Discussion Points on Test 1 vs. Test 2 Q3

There are several ways of interpreting Figure 3:

- Advocates for **geek genes or prior knowledge** will point out that students who did very well (or very poorly) on test 1 tended to do well (or poorly) on test 2 question 3.
- Advocates for **Learning Edge Momentum** will point out that the probability of providing a correct answer to test question 3 rises as test 1 score rises. A line of best fit through the data (solid line) accounts for an astonishing 80% of the variance.
- But advocates for the **Stumbling Point** hypothesis will point to the non-linear jump in probability values, as illustrated by the two dashed lines, between test 1 scores of 2−3 and 4−6. They might then calculate data as shown in Table 1. The table shows that only 6% of students with a total score of 2−3 on Test 1 could correctly trace code in parts c, d and e of question 1. In contrast, students who scored 4−6 on test 1 were more likely to perform better on parts c, d and e (73%, 64% and 45% respectively). Parts c, d and e contain two or more assignment statements where the assignments are from one variable to another variable. Constructing the contingency table shown in Table 2 and performing a $\chi 2$ test demonstrates the statistical significance of that jump in probability values in Figure 3. Thus the jump between test 1 scores of 2−3 and 4−6 may be an identifiable difficulty that many 2−3 students have, where their difficulty is an inability to trace multiple assignments from one variable to another variable.

4. TEST 1 VERSUS TESTS 3 AND 4

The advocates of the various hypotheses will each find comfort from a similar comparison of test 1 versus test 3 Q6(b) (see Figure 4) and also test 1 versus test 4 Q5 (see Figure 5):

- Advocates for **geek genes** or **prior knowledge** will again point out that students who did very well (or very poorly) on test 1 tended to do well (or poorly) on each of the questions from tests 3 and 4.
- Advocates for **Learning Edge Momentum** will again point out that the probability of providing a correct answer to each of the questions from tests 3 and 4 rises as test 1 score rises. While the lines of best fit do not account for as much variance as before, that is to be expected with Learning Edge Momentum.
- On Figure 4, advocates for the **Stumbling Point** hypothesis might perform the same type of statistical analysis as before (see Table 3) and see hints of a second jump in probabilities between test 1 scores of 4−6 and 7−8. However, they would also need to analyze test 1 and identify the qualitative difference between students who scored 4−6 and 7−8 on test 1. In Figure 5, there is not a statistically difference jump in probabilities between test 1 scores of 2−3 and 4−6 (see Table 4a), but perhaps by week 7 many students scoring 4−6 on test 1 may be floundering as much as most students who scored 2−3.

Table 1. The percentage of students who answered correctly the tracing parts of Test 1 Q1, broken down by total score. Each cell in the rows commencing "χ2, p =" show the probability of the two percentages above and below that cell NOT being statistically significant. Thus the greyed cells indicate statistically significant differences.

Test 1 Total Score	N	Test 1 Question 1 Tracing Parts				
		a	b	c	d	e
2 − 3	17	94%	100%	6%	6%	6%
χ2, p =		0.71	0.06	< 0.001	< 0.001	<0.01
4 − 6	22	91%	82%	73%	64%	45%
χ2, p =		0.50	0.32	< 0.01	0.02	0.02
7 − 8	24	96%	92%	100%	92%	79%

Table 2. The contingency table for how students answered Test 2 Q3 vs. their total score on Test 1 (χ2 test, p < 0.01).

Test 1 Total Score	N	Test 2 Question 3	
		wrong	right
2 − 3	17	12	5
4 − 6	22	6	16

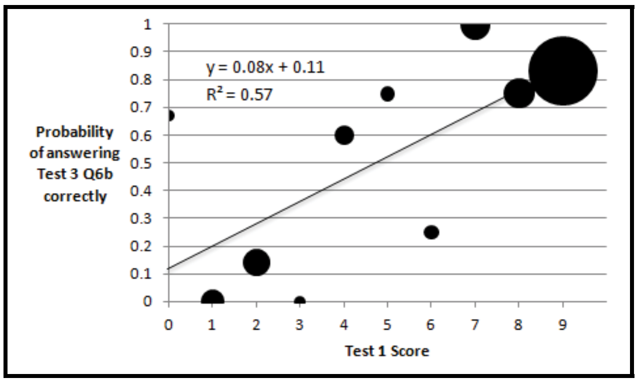


Figure 4. The relationship between student scores on test 1 and the probability of answering Test 3 Q6(b) correctly (N=66).

Table 3. Two contingency tables for how students answered Test 3 Q6(b) vs. their Test 1 score. (Both χ2 tests, p = 0.04).

Test 1 Total Score	Test 3 Q 6(b)	
	wrong	right
2 – 3	9	1
4 – 6	7	7

Test 1 Total Score	Test 3 Q 6(b)	
	wrong	right
4 – 6	7	7
7 – 8	3	15

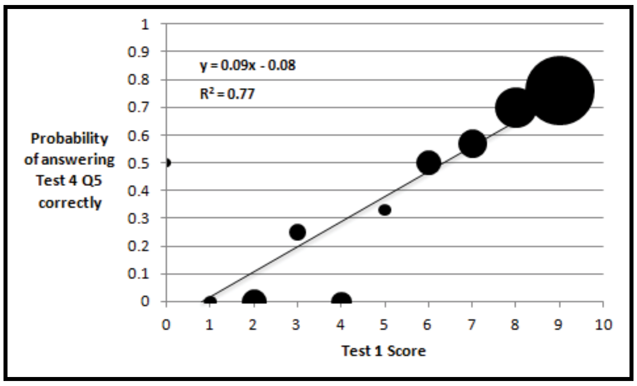


Figure 5. The relationship between student scores on test 1 and the probability of answering Test 4 Q5 correctly (N=63).

Table 4. The contingency tables for how students answered Test 4 Q5 vs. their Test 1 score.

Table 4a. χ2 test, p = 0.31

Test 1 Total Score	Test 3 Q 6(b)	
	wrong	right
2 – 3	9	1
4 – 6	11	4

Table 4b. χ2 test, p =0.03

Test 1 Total Score	Test 3 Q 6(b)	
	wrong	right
4 – 6	11	4
7 – 8	7	13

5. TEST 2 VERSUS TESTS 3 AND 4

The advocates of Learning Edge Momentum will react enthusiastically to comparisons of test 2 versus test 3 Q6(b) (see Figure 6) and also test 2 versus test 4 Q5 (see Figure 7). The advocates of the other hypotheses may point out that test 2 only has three questions, and both Figures 6 and 7 only have three data points of non-trivial sample size. Thus, so long as the probability associated with a test 2 score of 2 falls somewhere in the middle, a line of best fit is always going to be a good fit. The advocates of Learning Edge Momentum might then concede that the R^2 values are not to be taken too seriously, but never the less in both Figures the probability values rise with test 2 score − and that supports Learning Edge Momentum.

6. GENERAL DISCUSSION

A curious issue is that tests 1 and 2 were all about assignment statements, but assignments statements are not used in the questions from tests 3 and 4 (at least not explicitly). We don't know which hypotheses might be supported by that curious issue.

In an Indian folk story, blind men touch different parts of an elephant and describe the elephant in different ways. One blind

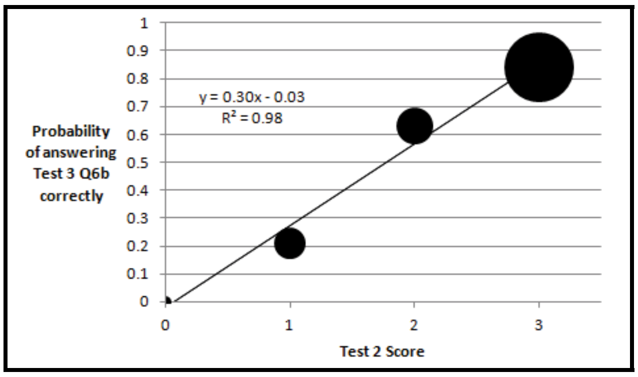


Figure 6. The relationship between student scores on test 2 and the probability of answering Test 3 Q6(b) correctly (N=66).

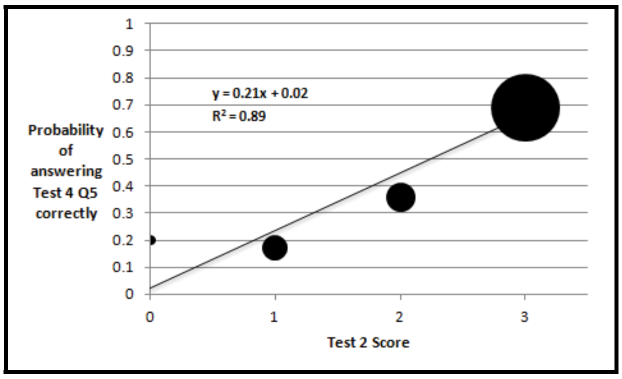


Figure 7. The relationship between student scores on test 2 and the probability of answering Test 4 Q5 correctly (N=63).

man feels a leg and claims the elephant is like a pillar; while another feels the tail and claims the elephant is like a rope, and so on. Likewise, what we have demonstrated in this paper is that advocates of the various hypotheses − Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum – can all find support for their respective hypotheses, in aspects of the data in this paper. What we need is someone with the vision required to generate a different type of data – a better way of studying the whole elephant. What is that data?

Some readers may believe it is futile to look for empirical data to distinguish between these hypotheses. Our first response to that is the old saying "*If you can't measure it, then it doesn't exist*". Our second response is more specific to this context: "*If you believe there is no empirical way of choosing between these competing hypotheses, then why believe in any one of them*?"

7. ACKNOWLEDGMENTS

Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education.

8. REFERENCES

[1] Robins, A. (2010) Learning edge momentum: A new account of outcomes in CS1. *Comp. Sci. Ed.*, 20(1), 37 – 71.

[2] Schilling, M., Watkins, A. and Watkins, W. (2002). Is Human Height Bimodal? *The American Statistician* 56 (3): 223--229. DOI:10.1198/00031300265

9. APPENDIX: THE FOUR TESTS

9.1 Test 1 (week 3)

Q1

(a) In the boxes, write the values in the variables after the following code has been executed:

```
int a = 1;
int b = 2;
a = 3;
```

The value in a is ☐ and the value in b is ☐

(b) In the boxes, write the values in the variables after the following code has been executed:

```
int r = 2;
int s = 4;
r = s;
```

The value in r is ☐ and the value in s is ☐

(c) In the boxes, write the values in the variables after the following code has been executed:

```
int p = 1;
int q = 8;
q = p;
p = q;
```

The value in p is ☐ and the value in q is ☐

(d) In the boxes, write the values in the variables after the following code has been executed:

```
int x = 7;
int y = 5;
int z = 3;

x = y;
z = x;
y = z;
```

The value in x is ☐ y is ☐ and z is ☐

(e) In the boxes, write the values in the variables after the following code has been executed:

```
int x = 7;
int y = 5;
int z = 0;

z = x;
x = y;
y = z;
```

The value in x is ☐ y is ☐ and z is ☐

(f) In part (e) above, what do you observe about the final values in x and y? Write your observation (in one sentence) in the box below.

Sample answer: *The values in x and y were swapped.*

Q2

The purpose of the following three lines of code is to swap the values in variables a and b, for any set of possible values stored in those variables. Assume that variables a, b and c have been declared and initialized.

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

(a) In one sentence that you should write in the box below, describe the purpose of the variable "c" in the above code.

Sample answer: *It is used to hold a value temporarily.*

(b) In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables. Assume that variables i, j and k have been declared and initialized.

```
j = i;
i = k;
k = j;
```

Sample answer: *The code swaps the values in i and k.*

Q3

Assume the variables first and second have been initialized. Write code to swap the values stored in first and second.

Sample answer:
```
int temp   = first;
    first  = second;
    second = temp;
```

Table A1. The percentage of students who answered correctly each question in test 1 (N=98)

Question	1(a)	1(b)	1(c)	1(d)	1(e)
Percent correct	92%	83%	66%	62%	55%
Question	1(f)	2(a)	2(b)	3	—
Percent correct	50%	63%	45%	52%	—

9.2 Test 2 (week 4)

Q1

In the boxes, write the values in the variables after the following code has been executed:

```
int a = 23;
int b = 11;
int c = 61;

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

The value in a is ☐ b is ☐ and c is ☐

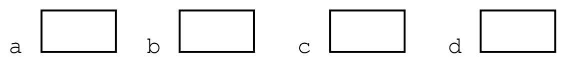
Q2

Assume the variables `black` and `white` have been initialized. Write code to swap the values stored in `black` and `white`.

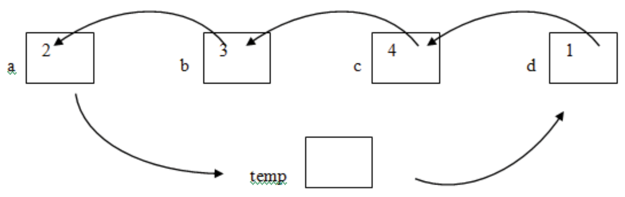
Sample answer:
```
int temp   = black;
    black  = white;
    white = temp;
```

Q3

Suppose there are four variables, a, b, c and d as depicted below:



Write code to move the values stored in those variables to the **left**, with the **leftmost** element being moved to the **rightmost** position - as depicted by this diagram:



For example, if a=1, b=2, c=3 and d=4 (as shown above), then after your code is executed those variables should contain a=2, b=3, c=4 and d=1. (But your code should work for all possible values.)

Sample answer:
```
temp = a
   a = b
   b = c
   c = d
   d = temp
```

Table A2. The percentage of students who answered correctly each question in test 2 (N=98)

Question	1	2	3
Percent correct	86%	61%	64%

9.3 Test 3 (week 6) Question 6

Consider the following block of code, where variables a, b and c each store integer values:

```
if (a > b)
     if (b > c)
        System.out.println(c);
     else
        System.out.println(b);

else if (a > c)
        System.out.println(c);
     else
        System.out.println(a);
```

(a) In relation to the above block of code, which one of the following values for the variables will cause the value in variable b to be printed? Draw a circle around the appropriate answer, (i), (ii), (iii) or (iv).

(i)	`a = 1;`	`b = 2;`	`c = 3;`
(ii)	`a = 1;`	`b = 3;`	`c = 2;`
(iii)	`a = 2;`	`b = 1;`	`c = 3;`
(iv)	`a = 3;`	`b = 2;`	`c = 1;`

(b) In one sentence that you should write in the box below, describe the purpose of the above block of if/else statements. Do NOT give a line-by-line description of what the code does. Instead, tell us the purpose of the code:

Sample answer: *It prints the smallest value.*

83% of 87 students answered test 3 question 6a correctly, while 57% of those students answered 6b correctly.

9.4 Test 4 (week 7) Question 5

In one sentence that you should write in the empty box below, describe the purpose of the following code. Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code. Assume that the variables y1, y2 and y3 are all variables with integer values. In each of the three boxes that contain sentences beginning with "`Code to swap the values` …", assume that appropriate code is provided instead of the box – do **NOT** write that code.

```
if (y1 < y2) {
     Code to swap y1 and y2 goes here.
}
if (y2 < y3) {
     Code to swap y2 and y3 goes here.
}
if (y1 < y2) {
     Code to swap y1 and y2 goes here.
}
```

Sample answer: *It sorts the numbers into ascending order.*

48% of 63 students answered test 4 question 5 correctly.

Note: Actually, two versions of this question were used. Around half the students answered another version where the numbers were sorted into descending order. The performance of students on both versions was not statistically different, so the data for the two versions was combined.

5.3 Discussion

This paper demonstrates that differences in the programming knowledge of students are evident from very early in an introductory programming course, when the only concept upon which students are tested is the very basic concept of assignment.

The data collected for this study was mainly extracted from a pen and paper environment. No source code snapshot data was analyzed at this stage. The only instrument of analysis was correlation analysis of the test outcome among multiple tests. In the next chapter, I demonstrate how the data collected from the coding environment of the novices can be used in the machine learning based correlation analysis to identify struggling students.

Chapter 6

Exploring machine learning methods to automatically identify students in need of assistance

6.1 Introduction

The paper presented in the previous chapter demonstrated that differences in the programming knowledge of students are evident from very early in an introductory programming course. The results in that paper were generated by simple correlation analysis of test questions. In that paper, no attempt was made to capture the process that led a student to a particular test answer.

The paper presented in this chapter aims to predict the performance in the course outcome through analysis of the very basic quantitative values collected from the source code snapshot data collected from novices in the early stages of the semester. The outcome of this paper shows that it is possible to predict the final exam mark of the subject with a high accuracy using dynamic data extracted from programming source code snapshots.

6.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;

2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;

3. there are no other authors of the publication according to these criteria;

4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and

5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: Exploring machine learning methods to automatically identify students in need of assistance

Conference: International Computing Education Research conference 2015

URL: http://dl.acm.org/citation .cfm?id=2787717&dl=ACM&coll=DL&CFID=862572254&CFTOKEN=60379727

Status: Presented, August 2015

TABLE 6.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 6

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Raymond Lister				✓
Heikki Haapala		✓		
Arto Vihavainen	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi

Name Signature Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister		
Name	Signature	Date

6.2 PDF of the Published Paper

Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance

Alireza Ahadi and Raymond Lister
University of Technology, Sydney
Australia
alireza.ahadi@uts.edu.au
raymond.lister@uts.edu.au

Heikki Haapala and Arto Vihavainen
Department of Computer Science
University of Helsinki
Finland
heikki.haapala@cs.helsinki.fi
arto.vihavainen@cs.helsinki.fi

ABSTRACT

Methods for automatically identifying students in need of assistance have been studied for decades. Initially, the work was based on somewhat static factors such as students' educational background and results from various questionnaires, while more recently, constantly accumulating data such as progress with course assignments and behavior in lectures has gained attention. We contribute to this work with results on early detection of students in need of assistance, and provide a starting point for using machine learning techniques on naturally accumulating programming process data.

When combining source code snapshot data that is recorded from students' programming process with machine learning methods, we are able to detect high- and low-performing students with high accuracy already after the very first week of an introductory programming course. Comparison of our results to the prominent methods for predicting students' performance using source code snapshot data is also provided.

This early information on students' performance is beneficial from multiple viewpoints. Instructors can target their guidance to struggling students early on, and provide more challenging assignments for high-performing students. Moreover, students that perform poorly in the introductory programming course, but who nevertheless pass, can be monitored more closely in their future studies.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; H.2.8 [**Database Applications**]: Data mining

Keywords

introductory programming; source code snapshot analysis; programming behavior; educational data mining; learning analytics; novice programmers; detecting students in need of assistance

1. INTRODUCTION

Every year, tens of thousands of students fail introductory programming courses world-wide, and numerous students pass their courses with substandard knowledge. As a consequence, studies are retaken and postponed, careers are reconsidered, and substantial capital is invested into student counseling and support. World-wide, on average one third of students fail their introductory programming course [4, 40]. Even when looking at statistics describing pass rates after teaching interventions, as many as one quarter of the students still fail the courses [38].

One of the challenges in organizing teaching interventions is that any change is likely to also affect students for whom the prevalent situation is more suitable. For example, if a student is already at a stage where she could work on more challenging projects on her own, mandatory excessively structured learning activities that everyone needs to follow may even be counterproductive for her [16, 31]. To provide another example, while collaborative learning practices such as pair programming [45] have been highlighted as efficient teaching approaches for introductory programming [23, 38], there are contexts in which students mostly work from a distance and rarely attend an institution.

This diversity of institutions, students, and teaching approaches is the setting upon which our work builds. We believe that the appropriate next step in teaching interventions is the transition towards interventions that address only those students that are in need of guidance, and work towards that goal by analyzing methods for detecting such students as early as possible. More specifically, in this work, we explore methods for detecting high- and low-performing students in an introductory programming course already based on the performance during the very first week of the course. Variants of the topic have been investigated previously, for example, by Jadud, who proposed an approach to quantify students' ability to solve errors using source code snapshots [15], Ahadi et al., who measured students' knowledge using tests [1, 2], and Porter et al., who used in-class clicker data as a lens into students' performance [24, 25].

This work is organized as follows. First, in Section 2, we provide an overview of the evolution of the field of understanding factors that contribute to students' performance in introductory programming. Then, in Sections 3 and 4 we outline our research questions and data in more detail, as well as explain the methodology and outline the results. The results are discussed in Section 5, and finally, Section 6 concludes the work and outlines future research questions.

2. BACKGROUND

In the article "*What best predicts computer proficiency*" [9], Evans and Simkin describe early advances into understanding attributes that contribute to the ability of learning to program. This ability, *programming aptitude*, is often defined as the student's ability to succeed in an introductory programming course, and is measured through e.g. the course grade or a finer-grained measure such as within-course point accumulation. Before 1975, the research focused mainly on demographic factors such as educational background and scores from previous courses, while by the end of the 1970s, the focus moved slowly to evaluating static tests that measure programming aptitude. This was followed by research that started to investigate the effect of cognitive factors such as abstraction ability and the ability to follow more complex processes and algorithms [9]. Such research has continued to this day by introducing factors related to study behavior, learning styles and cognitive factors [42]. However, recently, dynamically accumulating data from students' learning process has gained researchers' attention [15, 25, 37, 41].

Overall, this stream of research has been motivated by multiple viewpoints, which include identification of students that have an aptitude for CS-related studies (e.g. [35]); studying and identifying measures of programming aptitude as well as combining them (e.g. [3, 5, 30, 43]); improvement of education and the comparison of teaching methodologies (e.g. [34, 36]); and identifying at-risk students and predicting course outcomes (e.g. [15, 41]).

Next, we outline some of this work in more depth. We begin by focusing on factors that do not change at all or change very slowly, and continue towards dynamic factors that change more rapidly and where new information may be constantly accumulated.

2.1 Gender

Studies in past often investigated gender as one of the factors that may explain programming aptitude – one of the reasons may be that the field of computing is at times seen as being dominated by males, and thus exhibits a male-oriented culture. However, the results show no clear trend. For example, in an analysis of introductory programming course grade and gender, Werth found no significant correlation ($r = 0.080$) [43]. In a similar study, Byrne and Lyons found that female participants in introductory programming course had a marginally higher point average than their male counterparts, but the difference was not statistically significant [6]. The role of gender was also investigated by Ventura, who studied the effect of gender by comparing students' programming assignment, exam, and overall course points, and found no effect that could be explained by gender [36].

Studies exist that suggest a referential connection between programming aptitude and gender. For example, in a small study ($n = 11$), Bergin and Reilly observed that female students had statistically significant and strong correlations ($r = 0.72 - 0.93$) between an Irish high-school leaving certificate test and programming course scores [5] – an effect that was not visible among male counterparts.

2.2 Academic Performance

The connection between students' academic performance and programming aptitude has been investigated in several studies. For example, Werth analyzed the connection between the amount of tertiary education mathematics courses and programming aptitude, but found no significant correlation ($r = -0.019; p > 0.1$). She suggested that a large amount of mathematics courses in tertiary education may actually be an indicator of improving a weak mathematics background [43]. Other studies have found connections between mathematics and introductory programming. For example, Stein studied the connection between Calculus and Discrete Mathematics and the grade from an introductory programming course. The correlations, overall, were weak (Calculus: $r = 0.244$; Discrete Math: $r = 0.162$) [34]. Watson, Li and Goldwin did a similar study, and, similarly, found no significant correlation between the Discrete Math and the introductory programming grade ($r = 0.06; p > 0.05$). However, there was a mediocre albeit not statistically significant effect between the Calculus course grade and programming course points ($r = 0.37; p = 0.06$) [42].

In addition to mathematics, factors such as language performance and overall grade averages have also been studied. For example, Leeper and Silver studied students' English language scores and the score of the verbal part of the SAT test. In their study, only the verbal SAT score had a mediocre correlation with the introductory programming course grade ($r = 0.3777$) [19]. Werth found no significant correlation between secondary education grade average and the grade achieved in an introductory programming course ($r = 0.074; p > 0.1$), but she did find a weak correlation between university-level grade average and the introductory programming course grade ($r = 0.252; p < 0.01$) [43]. Similarly, Watson et al. studied correlations between various secondary education courses and course averages, but found no statistically significant correlations [42].

2.3 Past Programming Experience

It is natural to assume that past programming experience influences programming course scores, and thus, the connection has been studied in a number of contexts, albeit with contradictory results. Hagan and Markham found that students with previous programming experience received considerably higher course marks than the students with no programming experience [10]. Wilson and Shrock utilized five variables related to programming and computer use, such as formal programming education, the use of internet, and the amount of time spent on gaming. The combination of these variables had a significant correlation with the midterm score in an introductory programming course ($r = 0.387; p < 0.01$) [7]. Similarly, in 2004, Wiedenbeck et al. reported on a study in which the number of ICT courses taken by students, the number of programming courses taken, the number of programming languages students had used, the number of programs students had written, and the length of those programs were combined into a single factor. The combination had a weak but significant correlation with the introductory programming score ($r = 0.25; p < 0.05$) [44].

While multiple studies indicate a positive correlation between past programming experience and introductory programming course outcomes, somewhat contradictory results also exist. For example, Bergin and Reilly found that students with no previous programming experience had a marginally higher mean overall score in an introductory programming course, and found no statistically significant difference between students with and without previous programming experience [5]. In another study, Watson et al. found that while students with past programming experience had significantly higher overall course points than those with no previous programming experience [42], programming experience in years had a weak but statistically insignificant negative correlation with the course points ($r = -0.20$) [42].

2.4 Behavior in Lectures and Labs

Rodrigo et al. studied students' observed behavior in programming labs [26]. They studied students' gestures, outbursts, and other factors including collaboration with other students, and sought to identify factors that are potentially related to students' success. In addition, they collected source code snapshots from students' programming process. Six statistically significant factors ($p < 0.05$) that had a mediocre correlation with an introductory programming course midterm score were identified. Four of them were related to students' behaviors; confusion ($r = -0.432$), boredom ($r = -0.389$), focus ($r = 0.346$), and discussion about the programming environment (-0.316), while two were related to snapshots. The number of consecutive snapshots with errors ($r = -0.326$) and compilation events in which the student had worked on the same area in the source code ($r = -0.336$) were both negatively correlated with the midterm score [26].

Another angle at studying students behavior was recently proposed by Porter et al. [25], who studied students' responses to clicker questions in a peer instruction setting. In their study, they identified that the percentage of correct clicker answers from the first three weeks of a course was strongly correlated with overall course performance ($r = 0.61; p < 0.05$).

2.5 Source Code Snapshots

In "*Methods and Tools for Exploring Novice Compiling Behaviour*" [15], Jadud presents a method to quantify a student's tendency to create and fix errors, which he called the *error quotient*. In his study, the correlation between the error quotient and the average score from programming assignments was mediocre and statistically significant ($r = 0.36; p = 0.012$), while the correlation between the error quotient and the grade from a course exam was high ($r = 0.52; p = 0.0002$) [15]. Rodrigo et al. used an alternative version of Jadud's error quotient, and found that in their context the correlation between the error quotient and the midterm score of an introductory programming course was strong and statistically significant ($r = -0.54; p < 0.001$) [27]. In essence, this suggests that the less programming errors a student makes, and the better she solves them, the higher her midterm grade will tend to be [27].

Watson et al. also conducted a study using Jadud's error quotient, and found a significant correlation between the error quotient and their programming course scores ($r = 0.44$) [41]. They proposed that the amount of time that students spend on programming assignments should be taken into account, and that one should consider the files that a student is editing as a part of the error quotient calculation [41]. They proposed an improvement to the error quotient called *Watwin*, and found that with this improvement the correlation increased from ($r = 0.44$) to ($r = 0.51$) [41]. They also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with programming course scores ($r = -0.53; p < 0.01$).

Source code snapshots have been used to elicit information in finer detail as well. For example, Piech et al. [22] studied students' approaches to solving two programming tasks, and found that students' solution patterns are indicative of course midterm scores. Programming patterns were also studied by Hosseini et al., who identified students' behaviors within a programming course – some students were more inclined to build their code step by step, while others started from larger quantities of code, and reduced their code in order to reach a solution [14]. Another approach recently proposed by Yudelson et al. was to use fine-grained concepts extracted from source code snapshots, and to model students' understanding of these concepts as they proceed [46].

Next, we explore some of these methods for source code snapshot analysis, as well as provide researchers with an outline for performing such studies.

3. RESEARCH DESIGN

This study is driven by the question of identifying high- and low-performing students as early as possible in a programming course to provide better support for them. By high- and low-performing students, we mean students in the upper- and lower-half of course scores, and by early, we mean after the very first week of the programming course. This means that instructors could plan and provide additional guidance to specifically selected students already during the second week of the course.

For the task, we explore previously proposed methods for predicting students' performance from source code snapshots, and evaluate a number of machine learning techniques that have previously received little attention for the task at hand.

3.1 Research Questions

Our research questions for this study are as follows.

RQ1 Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?

RQ2 Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?

To answer the first question, we have implemented the algorithms described in [15, 41], and evaluate their performance on our data. For the second question, we first identify relevant features from a single semester, then evaluate different machine learning techniques to build a predictive model using the extracted features to determine a top-performing approach. Finally, the top-performing predictive model is evaluated on a dataset from a separate semester to determine cross-semester performance of the selected model.

3.2 Data

The data for the study comes from two semesters of an introductory programming course organized at the University of Helsinki. The course lasts six weeks, is taught in Java, and uses a blended online textbook that covers variables, basic I/O, methods, conditionals, loops, lists, arrays, elementary search algorithms and elementary objects. In the programming course, the main focus is on working on practical programming assignments, accompanied by a weekly two-hour lecture that covers the basics needed to get started with the work. Support is available in open computer labs, where teaching assistants and course instructors are available some 20-30 hours each week (see [18] for details).

Although no socio-economic factors were available for this study, the studied population is relatively homogenic, and the educational system in the context is socially inclusive, meaning that there is both a minimal underrepresentation of students from low education background and a minimal overrepresentation of students from high education background [21]. There are also no tuition fees, and students receive student benefits such as direct funding from the state, assuming that they progress in their degree work.

For the purposes of this study, students' programming process was recorded using Test My Code [39] that is used for automatically assessing students' work in the course. For each student that consented to having their programming process recorded, every key-press and related information such as time and assignment details was stored. The students used the same programming environment both from home and at the university. Students were asked to provide information on whether they had prior programming experience, and access to information on students' age, gender, grade average, and major was given for the researchers for the purposes of this study. In the studied context, major is selected before enrollment, and in both semesters, over 50% of the students had other subjects than computer science as their major – for students with CS as a major, the studied course is the first course that they take.

In the first semester (spring), a total of 86 students participated in the study, and in the second semester (fall), a total of 210 students participated in the study. Full fine-grained key-log data is available only for the first semester, while for the second semester, only higher level actions such as saves, compilation events, run events and test events are available.

While attendance in the course activities is not mandatory, 50% of total course points comes from completing programming assignments. The rest of the course points comes from a written exam, where students answer both essay-type questions as well as programming questions. To pass the course, the students have to receive at least half of the points from the exam as well as half of the points from the programming assignments, while the highest grade in the course can be received by gathering over 90% of the course points.

4. METHODOLOGY AND RESULTS

The students were divided into groups based on their performance in (1) an algorithmic programming question given in the exam, (2) the overall course, and (3) a combination of the two. The first division into groups is motivated by students' struggling with writing programs even at a later phase of their studies [20], and has also been the focus in related studies, such as the work by Porter et al. [25]. The algorithmic programming question is a variant of the Rainfall Problem [32], where students have to create a program that reads numbers, possibly filters them, and prints attributes such as the average of the accepted numbers. The second division into groups outlines the students' overall performance, and the third division combines the previous. Table 1 shows student counts in these groups for the dataset that is used to evaluate the algorithms in RQ1, and to train the predictive model for RQ2.

4.1 Research Question 1

To answer the first research question, "*Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?*", we implemented these algorithm's as they were described [15, 41]. Both algorithms use a set of successive compilation event pairings to quantify the students' ability to fix syntactic errors in the programs that they are writing. The main difference between Jadud's error quotient and the Watwin-algorithm is that the Watwin-algorithm also considers the possibility that students may be working on multiple files, where one file has errors, and the other does not. Thus, changing from one file to the other is not seen as if the user fixed the errors. Moreover, the Watwin algorithm also takes into account the amount of time that students spend on fixing errors.

Table 1: Student counts for the studied population, binned based on the predicted variable.

Target class	Median or Above	Below Median
Exam Question	47	39
Final Grade	48	38
Combined	43	43

Unlike the data used by Jadud and Watson et al., the data recorded from standard programming environments do not have explicit compilation events as the environments continuously compile the code and highlight errors to developers. To approximate these explicit compilation events for Jadud EQ and Watwin algorithm, two options were evaluated: (1) use only snapshots where students perform an action that does not involve changing the code, i.e. run their code, test their code, or submit the code to the assessment server (i.e. *action* in Table 2), and (2) use only snapshot pairs between which the students have taken at least a ten second pause from programming (*pause* in Table 2). The value for the pause was determined by evaluating the algorithms with 60, 30, 10 and 5 second pauses, after which the value which resulted in the best average performance was selected. Our rationale for the use of actions is that in such cases, the students want explicit feedback from the system, while the rationale for pauses is that the students have stopped to, for example, debug their program. Option (2) is only available for the first semester, as fine-grained key-log data is not available for the second studied semester.

Pearson correlation coefficients between the predicted variables (Table 1) and Jadud's error quotient and Watwin-score are given in Table 2. The correlations are given as absolute values, and are all low ($r < 0.3$).

Table 2: Pearson Correlation coefficients for between the Jadud's error quotient, the Watwin-score, and the predicted variables.

Variable	Semester	Jadud		Watwin	
		action	pause	action	pause
Exam Quest.	First	.15	.21	.25	.18
	Second	.20	-	.09	-
Final Grade	First	.03	.08	.01	.13
	Second	.10	-	.005	-
Combined	First	.02	.08	.01	.13
	Second	.12	-	.01	-

4.2 Research Question 2

To answer the second research question, "*Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?*", the problem was approached as a supervised learning task, where existing data is used to infer a function that can be used to categorize incoming data into groups [13].

First, features were extracted from the dataset. Then, to avoid the use of irrelevant or redundant features, feature selection was used to identify relevant features. Once a relevant subset of features had been selected, we evaluated a number of classifiers. Finally, when a classifier had been selected from the evaluated classifiers, we tested the model against a data set from a separate semester. Feature selection and classifier evaluation was performed using the WEKA Data Mining toolkit [11].

Feature Extraction

For the study, we extracted two types of attributes: (a) Attributes based on previously studied success factors, such as previous academic performance (tertiary education) and past programming experience; (b) programming assignment specific Source-code snapshot attributes that potentially reflect students' persistence and success with the course assignments. For each assignment, the number of steps that a student took, measured in key-presses and other actions, as well as the maximum achieved correctness when measured by automated tests was extracted. The Source-code snapshot attributes were programmatically extracted from the programming process data, which is recorded by Test My Code as students are working on the assignments. An overview of the used attributes is given in the Table 3. The datasets were also normalized.

Table 3: Features extracted for the study

Features	Type
Gender	Categorical
Major	Categorical
Grade Average	Numerical
Age	Numerical
Programming experience	Binary
Maximum obtained correctness for each programming assignment	Numerical $[0-1]$
Amount of steps taken in each of the programming assignment	Numerical $[0-\infty]$

Feature Selection

After the feature extraction phase, there was a total of 53 features. To reduce the amount of overlapping features, possible over fitting, and to potentially improve predictive accuracy of the feature set, feature selection was performed. We used correlation-based feature subset selection [12], where individual predictive ability of each feature along with the degree of redundancy between them was evaluated using three methods; (1) genetic search, (2) best first method and (3) greedy stepwise method. Results of the feature selection phase are given in Table 4.

After this phase, the information gain of each feature was measured to reveal features that had little or no predictive value. Information gain, or Kullback-Leibler divergence [17], is used to measure the amount of information that the feature brings about a predicted value, assuming that they are the only two existing variables, and is measured by the difference of two probability distributions (in our case, e.g., the difference of the probability distributions of the exam question results and grade average). After measuring information gain for each of the features and predicted value, the low-contributing features were removed. The features above the line in Table 5 were retained in the training set.

Table 5: Information gain of the features. Features below the line were excluded from further use.

Feature	Exam question	Grade	Both
Grade Average	0.34	0.36	0.44
Correctness of a20	0.40	0.40	0.38
Steps for a23	0.44	0.32	0.29
Steps for a21	0.23	0.20	0.20
Steps for a22	0.22	0.16	0.19
Major	0.17	0.11	0.13
Steps for a17	0.27	0.15	0.12
Steps for a20	0.26	0.15	0.12
Steps for a18	0.14	0.15	0.12
Steps for a19	0.23	0.13	0.11
Age	0.11	-	0.11
Prog. Exp	-	0.05	0.07
Gender	0.01	0.008	0.003

Classifier Evaluation

As is typical for studies that explore machine learning methodologies, a number of classifiers were evaluated. In our case, we evaluated three families of classifiers; Bayesian classifiers, Rule-learners, and Decision tree -based classifiers, and chose a total of nine classifiers from these three families. All of these approaches are commonly used for classifying students [28, 29]. The evaluation was performed using two separate validation options: k-fold cross validation (with k=10), and percentage split (2/3 of the dataset used for training and 1/3 for testing). This means that during the classifier training and evaluation phase, parts of the data was hidden during the training, and was then used for the evaluation. Table 6 presents the results for the classification algorithms that were investigated in this study.

As can be seen in Table 6, the overall accuracy of decision trees is higher than that of the other two classifier families. Among decision trees, Random Forest has on average the highest accuracy for all predictive variables with 86%, 90% and 90% accuracy for predicting Exam question, Final Grade and the combination of both. To show the predictive accuracy in more detail, Table 7 shows the confusion matrix of the Random Forest classifier when predicting the combination of the Exam Question and Final grade, when using 10-fold cross-validation on the training data set.

Table 7: Confusion matrix of Random Forest on predicting whether students are equal-to-or-above or below the median score on the combination of exam question and final grade

	Predicted above	Predicted below
Actual above	38	5
Actual below	3	40

Thus, we selected the Random Forest as the classifier that is used to evaluate students' performance. More detailed evaluation of the performance of the Random Forest classifier is given in Table 8. The F1-Measure, which represents the balanced precision-recall, shows that Random Forest provides a strong result in this prediction task. Moreover, the Receiver operating characteristic value (*ROC*) suggests that the classifier still performs well when the classification threshold is changed from the median, i.e. if we would rather

Table 4: Features selected during feature selection. The left-hand side describes the feature selection method, and the columns describe the features selected for the different predictive variables. ***Steps*** **denotes the number of recorded events for a student on a specific programming assignment.** ***Correctness*** **denotes the percentage of tests passed by a student on a specific programming assignment.**

Method	Exam question	Final Grade	Both
Best First	Age; Grade Average; Steps for a17, a21, and a23	Grade Average; Steps for a21 and a23; Correctness for a23	Grade Average; Steps for a21 and a23; Correctness for a20
Genetic Search	Age; Grade Average; Steps for e17, e19, e20, e21, and e23; Correctness for e2, e6, e11, and e12	Grade Average; Steps for e20, e21, and e23; Correctness for e23	Grade Average; Steps for e21, and e23; Correctness for e20
Greedy Stepwise	Age; Grade Average; Steps for e17, e21, and e23	Grade Average; Steps for e21 and e23; Correctness for e23	Grade Average; Steps for e21 and e23; Correctness for e20

Table 6: Classifier accuracy when performing evaluation of the classifiers on the training set from a single semester. The highest accuracies are marked with bold. Exam question is shown as Q in the Table.

Classifier	Family	10-fold cross-validation accuracy			percentage split accuracy		
		Q	Final grade	Q + Final grade	Q	Final grade	Q + Final grade
Naive Bayes	Bayesian	80%	80%	77%	86%	86%	86%
Bayesian Network	Bayesian	81%	77%	76%	82%	76%	72%
Decision Table	Rule Learner	78%	73%	84%	86%	76%	90%
Conjuctive Rule	Rule Learner	73%	80%	83%	72%	86%	90%
PART	Rule Learner	85%	79%	**93%**	90%	76%	82%
ADTree	Decision Tree	80%	85%	86%	90%	83%	83%
J48	Decision Tree	83%	82%	**93%**	**93%**	89%	83%
Random Forest	Decision Tree	**86%**	**90%**	90%	90%	**90%**	**93%**
Decision Stump	Decision Tree	73%	76%	84%	83%	**90%**	90%

seek to identify the lowest performing quartile of students, and the Matthews correlation coefficient (*MCC*) shows a high correlation ($r = 0.71 - 0.81$) between the classifier and the predicted values.

Evaluation on a Separate Semester

As the data that is produced within educational settings varies between semesters, due to variations in student cohorts and course changes, the generalizability of the model needs to be evaluated using data from a separate semester. Accordingly, we evaluated the Random Forest -classifier (i.e. our best performing classifier from above) on data from a separate semester with $n = 210$ students. We found that the Random Forest -classifier was able to categorize students on the Exam Question, the Final Grade, and the combination of both with the accuracy of 80%, 73%, and 71% respectively, when the training of the model was performed on the data from the first semester with $n = 86$ students.

5. DISCUSSION

5.1 Research Question 1

To answer research question one, "Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?", the performance of the approaches differs from the studies in which the algorithms have traditionally been evaluated. Next, we discuss factors which may explain this result.

First, we use data from a considerably shorter period than Jadud and Watson et al. use in their studies. The first results in the article by Watson et al. [41] are given after three weeks into the course, and at that time, the correlation coefficients are near 0.3 for both Watwin-score and Jadud's error quotient – marginally better than our results. Moreover, in Watson et al.'s work, the analysis is performed against overall coursework mark, that is, the overall score from programming assignments [41], and not against the performance in a written exam.

Second, the programming environment used in the studies by Jadud and Watson et al. expects the student to take an extra step for her to receive information on whether her code compiles or not, while such a step is not necessary in current programming environments. It is possible that such a feature stimulates specific working behavior, which in turn may have contributed to previously observed outcomes.

A third factor is related to the quantity and type of the programming assignments. In the context of our study, the students work on a relatively large number of programming assignments during the very first week. Many of the assignments are relatively straightforward, and have been designed to help students gain confidence. This means that it is possible that the predictive approaches that are based on students' programming errors may also be dependent on the programming assignments being non-trivial for the students, which is not always the case in the studied context. These details from the contexts of Jadud and Watwin are not at our disposal.

Finally, the fourth factor is the guidance that students receive during the course. For example, in the context of Watson et al. [41], the students have specific and limited lab hours during which they can receive support on the programming assignments, while in the context that we studied, the labs are open most of the time, and anyone can attend. It is also possible that the type of guidance provided in labs differs.

Table 8: Statistical measures for the Random Forest -classifier when predicting the considered target variables. TPR stands for True Positive Rate, FPR stands for False Positive Rate, ROC stands for Receiver Operating Characteristic, and MCC stands for Matthews Correlation Coefficient.

Class	TPR	FPR	Precision	Recall	F1-Measure	ROC	MCC
Exam question	0.86	0.14	0.86	0.86	0.86	0.92	0.71
Final grade	0.89	0.10	0.89	0.89	0.89	0.92	0.78
Exam question & Final grade	0.90	0.09	0.90	0.90	0.90	0.95	0.81

5.2 Research Question 2

To answer research question two, ”Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?”, we both described the workflow of creating and evaluating machine learning algorithms as well as outlined the results. The process starts with feature extraction, continues with feature selection that is followed by classifier evaluation, and finally concludes with evaluation with a separate data set – in our study, from a separate semester. While the performance of the classifier was high when evaluating the approach within a single semester, ranging from 86% to 90% accuracy with 10-fold cross-validation, the performance was lower (ranging from 71% to 80%) when the predictive model was evaluated on data from a separate semester.

When extracting and selecting the most important features, it was observed that most *a priori* features such as past programming experience, age, and gender made relatively little contribution to the predicted values. This is in line with previous research, which was discussed in Section 2. The information provided by *a priori* features was lower than that of the performance in the actual programming assignments. The most important features were students' grade average, the maximum percentage of automated tests that a student's solution to a specific programming assignment reached, number of steps that students took in a number of programming assignments, and the students' major. Note that for the students who have CS as their major, no grade average was available as the programming course was the very first course that they took – tree-based models handle this well.

5.3 Analysis of Programming Assignments

The feature selection process selected a number of programming assignments as important for the predictive process. All of the programming assignments were from the later part of the week – assignments 17 to 23 were selected, out of a total of 24 assignments in the first week. In all of these programming assignments from the first week, students were given a class that had an empty main-method. In the assignments leading to assignment 17, students had practiced producing different kinds of outputs, the use of variables such as `int` and `String`, reading input from the keyboard, simple comparisons with `if` and `if-else` structures, and combinations of these. Instructions for assignments 17 to 23 are given in Table 9. In addition to what is shown in the table, students had one or two examples of the program output. Also, assignment 23 had an API description of the visualization library.

As with assignments 1-16, assignments 17 and onwards introduce new concepts step-by-step. For example, in assignment 17, the students practice the use of an `else if` structure for the first time, and in assignment 18, the students are expected to use multiple `else if` statements. In assignment 19, the students are practicing the same concepts as in assignment 18, but with a different task and a smaller number of cases that need to be taken into account. Assignment 20 is the first assignment in which the students compare String variables. Assignments 21 is more algorithmic in nature than earlier assignments. Finally, assignments 22 and 23 are programs that require the student to use a loop for the first time. These are concepts that are known not to be easy in other contexts as well (see e.g. [8]).

It is somewhat surprising that assignment 20 was the only assignment for which the student's maximum achieved correctness, i.e. the percentage of tests passed, was highlighted as an important feature. Upon further analysis, as the students were accustomed to comparing numbers, many had initially challenges with comparing strings and the use of the `equals` method, which was needed in the assignment. Most of the students eventually did tackle this, and some of those that did not seemed to be confused with comparing multiple strings at the same time; even if not completing the assignment, students eventually moved forward. At the same time, a persistent student could work through the assignments with the support from the programming environment and course staff, given that she would not start too late, which likely also explains parts of the correctness not being important. From

Table 9: Programming assignments that were highlighted during the feature selection process. Examples of input/output were also given to students.

#	assignment instructions
17	Write a program that reads in two numbers from the user, and prints the larger of them. If the numbers are equal, the program should output ”they are equal.”
18	Write a program that reads in a number between 0 and 60, and transforms it to a grade using the following rules: 0-35 should be F, 36-40 D, 41-45 C, 46-50 B, and 51-60 A.
19	Write a program that reads in a number and checks that it is a valid age [0-120]. If the number is within the range, the program should output ”OK!”, otherwise the program should output ”Impossible!”.
20	Write a program that reads an username and a password, and compares them to user credentials that are given with the assignment. The program should print ”correct”, if the credentials are correct, otherwise, ”false”.
21	Write a program that reads in a number, and determines whether it is a leap year or not.
22	Write a program that continuously asks for a password until the user types in the right password.
23	Write a program that continuously reads in numbers, if the numbers are between [-30, 40], they are to be added to a plot (a ready library given). The program execution should never end.

the viewpoint of a material designer, the first finding could imply that it might be beneficial to consider an assignment with simpler string comparisons at first, e.g. by comparing just a single string, instead of the first assignment being one where two strings are compared at the same time. However, this was no longer an issue in assignment 22, where the students combined the same behavior with a loop construct.

Overall, when considering the number of steps that the students took to reach a solution, the students in the high-performing group took more steps on average than the students in the low-performing group. While initially one would assume that this would be explained simply by the low-performing students not attempting the assignments, this was not the case. It simply seems that the students in the high-performing group, when generalized, tried out more than a single approach and were not always content with simply reaching a working solution. Such behavior was also encouraged by the course staff.

5.4 Misclassified Students

We also performed an analysis of the students who were misclassified, i.e. students who were classified into another category than that to which they belonged. When considering the students who were classified as high-performing but belonged to the low-performing group, a number of them had adopted a work behavior where they diligently worked through the assignments by battling their way through the automatic tests. This is likely due to a result that has been previously pointed out by Spacco, i.e., if students are given full test results, they may adopt the habit of *"programming by 'Brownian motion', where students make a series of small, seemingly random changes to the code in the hopes of making their program pass the next test case"* [33] – currently, the programming environment used does not provide ways to battle this behavior.

Similarly, when considering the students who were classified as low-performing, but were high-performing, some of them used copy-paste in a quantity that had the classifier consider them as students who did not explore the solutions at length. Note that this does not mean that these students were plagiarizing their solutions from others, but seemed to extensively utilize their solutions from previous assignments.

5.5 Practical Implications

Our work implies that one can differentiate between the high- and low-performing students in a programming class already based on the performance of a single week with a relatively high accuracy. This means that instructors may, potentially, provide targeted interventions already during the second week. Practices such as additional rehearsals could be introduced for low-performing students, while high-performing students may benefit from additional challenges.

The results also indicate that students' programming behavior during the class is more important than background variables such as age, gender, or past programming experience, which is in line with previous studies. Moreover, in the studied context, the correctness of the students' solutions was not as important as the effort. That is, students who simply pushed towards a solution did not benefit from the programming tasks as much as the students who did additional experiments. It is plausible that such information on students' behavior can also be used to guide students towards more productive learning strategies.

5.6 Limitations of work

Predictive models are generalizations over a dataset gathered during a single or a number of semesters, and should always be validated using an additional dataset. As is evident in our case, the new dataset, when gathered within the same context but during a different semester, had different results than those from the initial evaluation. At the same time, the comparison was strict as we compared the performance of a model built on data from a spring semester against data from a fall semester. This effectively demonstrates that if the teaching approach, materials, or other related variables change, the performance of the predictive model may also change. That is, the predictive model is tuned to a specific context and dataset, and thus, it should be adjusted if the context changes.

Naturally, while the machine learning approach described in this article generalizes to other contexts, one should not assume that the same features would be the best features in other contexts as well. That is, the process should be started from the first step, i.e. extracting features, and followed as described in this article. That is, the predictive model that works on our data set would likely be different from a predictive model from other data sets – how different is a question that is left for future work. This is likely similar for all related studies.

6. CONCLUSIONS AND FUTURE WORK

In this work, we explored methods for early identification of students to guide from naturally accumulating programming process data. Such information can be useful for instructors and course designers, and can be used to create targeted interventions and to adjust materials accordingly. For example, the students who are performing well in the course may benefit from additional, more challenging tasks, while the students who are performing poorly are likely to benefit from rehearsal tasks as well as other activities that are typically used to help at-risk students.

The three main contributions of this article are as follows: (1) Analysis of the performance of existing source code snapshot-based methods for identifying high- and low-performing students in a new context; (2) Exploration of machine learning techniques for identifying high- and low-performing students; and (3) Analysis of cross-semester performance of the predictive models.

When analyzing the performance of the methods proposed by Jadud and Watson et al., we observed that the approaches had relatively poor performance on the data at our disposal. When exploring the performance of the machine learning techniques, the within-dataset performance was higher than that of the cross-semester performance, which was measured based on the predictive performance during a separate semester. This is explainable by the natural variance between semesters and student populations. Even so, with the cross-semester accuracy that ranges between 70% and 80%, reaching many of the right students is possible.

As a part of our future work, we are tuning the predictive models using additional data, seeking to further understand the students' behavior by delving deeper into their programming process, and conducting interviews that hopefully will shed further light on students' working practices as well as to those students who were misclassified. We are also performing targeted interventions within the studied context.

7. REFERENCES

[1] A. Ahadi and R. Lister. Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of the one elephant? In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 123–128, New York, NY, USA, 2013. ACM.

[2] A. Ahadi, R. Lister, and D. Teague. Falling behind early and staying behind when learning to program. In *Proceedings of the 25th Psychology of Programming Conference*, PPIG '14, 2014.

[3] J. Bennedsen and M. E. Caspersen. Abstraction ability as an indicator of success for learning object-oriented programming? *ACM SIGCSE Bulletin*, 38(2):39–43, 2006.

[4] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36, 2007.

[5] S. Bergin and R. Reilly. Programming: factors that influence success. *ACM SIGCSE Bulletin*, 37(1):411–415, 2005.

[6] P. Byrne and G. Lyons. The effect of student attributes on success in programming. In *ACM SIGCSE Bulletin*, volume 33, pages 49–52. ACM, 2001.

[7] B. Cantwell Wilson and S. Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. In *ACM SIGCSE Bulletin*, volume 33, pages 184–188. ACM, 2001.

[8] Y. Cherenkova, D. Zingaro, and A. Petersen. Identifying challenging CS1 concepts in a large problem dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 695–700, New York, NY, USA, 2014. ACM.

[9] G. E. Evans and M. G. Simkin. What best predicts computer proficiency? *Communications of the ACM*, 32(11):1322–1327, 1989.

[10] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? *ACM SIGCSE Bulletin*, 32(3):25–28, 2000.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[12] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[13] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[14] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a Java programming course. In *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group*, 2014.

[15] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[16] H. Jang, J. Reeve, and E. L. Deci. Engaging students in learning activities: It is not autonomy support or structure but autonomy support and structure. *Journal of Educational Psychology*, 102(3):588, 2010.

[17] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[18] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, pages 3–8, New York, NY, USA, 2011. ACM.

[19] R. Leeper and J. Silver. Predicting success in a first programming course. In *ACM SIGCSE Bulletin*, volume 14, pages 147–150. ACM, 1982.

[20] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, 33(4):125–180, Dec. 2001.

[21] D. Orr, C. Gwość, and N. Netz. *Social and economic conditions of student life in Europe: synopsis of indicators; final report; Eurostudent IV 2008-2011.* W. Bertelsmann Verlag, 2011.

[22] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[23] L. Porter, M. Guzdial, C. McDowell, and B. Simon. Success in introductory programming: What works? *Communications of the ACM*, 56(8):34–36, 2013.

[24] L. Porter and D. Zingaro. Importance of early performance in CS1: Two conflicting assessment stories. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 295–300, New York, NY, USA, 2014. ACM.

[25] L. Porter, D. Zingaro, and R. Lister. Predicting student success using fine grain clicker data. In *Proceedings of the tenth annual conference on International computing education research*, pages 51–58. ACM, 2014.

[26] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and behavioral predictors of novice programmer achievement. *ACM SIGCSE Bulletin*, 41(3):156–160, 2009.

[27] M. M. T. Rodrigo, E. Tabanao, M. B. E. Lahoz, and M. C. Jadud. Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science*, 138(2):177–190, 2009.

[28] C. Romero and S. Ventura. Educational data mining: a review of the state of the art. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(6):601–618, 2010.

[29] C. Romero, S. Ventura, P. G. Espejo, and C. Hervás. Data mining algorithms to classify students. *Educational Data Mining 2008.*

[30] N. Rountree, J. Rountree, A. Robins, and R. Hannah. Interacting factors that predict success and failure in a CS1 course. In *ACM SIGCSE Bulletin*, volume 36, pages 101–104. ACM, 2004.

[31] E. Sierens, M. Vansteenkiste, L. Goossens, B. Soenens, and F. Dochy. The synergistic relationship of perceived

autonomy support and structure in the prediction of self-regulated learning. *British Journal of Educational Psychology*, 79(1):57–68, 2009.

[32] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, Sept. 1986.

[33] J. Spacco. *Marmoset: a programming project assignment framework to improve the feedback cycle for students, faculty and researchers.* PhD thesis, 2006.

[34] M. V. Stein. Mathematical preparation as a basis for success in CS-II. *Journal of Computing Sciences in Colleges*, 17(4):28–38, 2002.

[35] M. Tukiainen and E. Mönkkönen. Programming aptitude testing as a prediction of learning to program. In *Proc. 14th Workshop of the Psychology of Programming Interest Group*, pages 45–57, 2002.

[36] P. R. Ventura Jr. Identifying predictors of success for an objects-first CS1. 2005.

[37] A. Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*. IEEE, 2013.

[38] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[39] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 117–122. ACM, 2013.

[40] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44. ACM, 2014.

[41] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 319–323. IEEE, 2013.

[42] C. Watson, F. W. Li, and J. L. Godwin. No tests required: comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 469–474. ACM, 2014.

[43] L. H. Werth. *Predicting student performance in a beginning computer science class*, volume 18. ACM, 1986.

[44] S. Wiedenbeck, D. Labelle, and V. N. Kain. Factors affecting course outcomes in introductory programming. In *16th Annual Workshop of the Psychology of Programming Interest Group*, pages 97–109, 2004.

[45] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner. Building pair programming knowledge through a family of experiments. In *Proc. Empirical Software Engineering*, pages 143–152. IEEE.

[46] M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky. Investigating automated student modeling in a Java MOOC. In *Proceedings of The Seventh International Conference on Educational Data Mining 2014*, 2014.

6.3 Discussion

The information extracted from the data collected from the novices was originally representing two main features: the so called "number of attempts" and the "performance". One of the issues with throwing data at a data mining algorithm is how the data is treated within the algorithm and consequently how it is interpreted at the final result level. The *mark* of each programming task in the data preprocessing step was either zero or one. I assigned the value one to the exercise mark if the mark was greater than the median of the mark of all novices doing that particular assignment, or a zero if it was not. This means that the mark allocated to a student's assignment is calculated based on the comparison of his mark to the whole class. On the other side, *what* exactly does the "number of attempts" represent? Is it a ratio of level of engagement to course? Is it reporting on how much the novice struggled with the code, or how much attention was given by the student to the code? Regardless of what can be understood from the data, I understood something from the very nature of the machine learning tools: it is very important to understand the relationship between what the data is representing and what is going to be predicted. There might be a strong positive correlation between the distance from the student's home to the university, and the performance in the course. However, this is not explained by the machine learning algorithm and might simply distract the prediction algorithm from the core concepts that might give clues towards a better understanding of the novice. There is fundamental intelligence missing behind most state of the art machine learning techniques: they can't make *sense* of the data.

These questions all evolve from the data and the data analysis. The findings of this chapter shed lights on the experiment design and indicates that there is more in depth analysis of the data required so that the researcher can draw general comments on the novices. I then aimed to explore the application of the machine learning tools on the information extracted from another context. This not only opens other windows on the caveats of data analysis and data collection in other contexts, but also gives clues on issues that may be raised in other context, and also a sense of consistency on the findings reported in this chapter.

Chapter 7

A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries

7.1 Introduction

As reviewed in the previous chapter, the basic information extracted from the source code snapshot data generated by the novices can be used in a machine learning context to predict the final exam performance of the novices in an introduction to programming course. I discussed the importance of a solid understanding of the data, and how it is generated, processed and interpreted. In this chapter, I aim to perform the analysis done in Chapter 5 using the data collected from the novices in another context: the SQL *SELECT* statements.

The information which is extracted from novices used to perform data analysis in this chapter and the next two chapters is basically extracted from the SQL source code snapshots generated by novices during their attempt to answer seven different types of SQL questions in a built-in online assessment environment called AsseSQL (Prior, 2014). Features extracted from these snapshots includes "number of attempts" and "performance" collected through analysis of around 163000 SQL *SELECT* statement snapshots generated by around 2300 database novices. In this chapter, I perform a correlation analysis between the performance of different questions of the online SQL tests.

7.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;
2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;
3. there are no other authors of the publication according to these criteria;
4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and
5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries

Conference: ACM Conference on Innovation and Technology in Computer Science Education

URL: http://dl.acm.org/citation.cfm?id=2742620&CFID=862572254&CFTOKEN=60379727

Status: Presented, June 2015

TABLE 7.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 7

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Julia Prior	TBA	TBA	TBA	TBA
Vahid Behbood	TBA	TBA	TBA	TBA
Raymond Lister	TBA	TBA	TBA	TBA

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi		
Name	Signature	Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister		
Name	Signature	Date

7.2 PDF of the Published Paper

A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries

Alireza Ahadi, Julia Prior, Vahid Behbood and Raymond Lister

University of Technology, Sydney, Australia

{Alireza.Ahadi, Julia.Prior, Vahid.Behbood, Raymond.Lister}@uts.edu.au

ABSTRACT

This paper presents a quantitative analysis of data collected by an online testing system for SQL "select" queries. The data was collected from almost one thousand students, over eight years. We examine which types of queries our students found harder to write. The seven types of SQL queries studied are: simple queries on one table; grouping, both with and without "having"; natural joins; simple and correlated sub-queries; and self-joins. The order of queries in the preceding sentence reflects the order of student difficulty we see in our data.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages – query languages.

General Terms

Management, Measurement, Human Factors.

Keywords

Online assessment; databases; SQL queries.

1. INTRODUCTION

The chief executive officer of edX, Anant Agarwal, has been quoted as saying that Massive Open Online Courses (MOOCs) will be the "*particle accelerator for learning*" (Stokes, 2013). A similar sentiment was expressed 30 years earlier. In her study of the usability of database query languages, Reisner (1981) asked rhetorically, "*Why experiment at all*?" before answering:

> *... should not just using one's own judgment, or the judgment of some expert, suffice? ... Unfortunately, the computer scientist is not necessarily a good judge of the abilities of* [students]*; there are no experts whose opinions would be generally accepted, and if there were, they might not agree. ... A more cogent reason to forego judgment by peers or by an expert is that such judgment is not quantitative.*

With the same spirit as Agarwal and Reisner, in this paper we use data collected over eight years, from 986 students, to study quantitatively the relative difficulty our students had with completing seven different types of SQL queries.

ITiCSE'15, July 4–8, 2015, Vilnius, Lithuania.
Copyright © 2015 ACM 978-1-4503-3440-2/15/07…$15.00.
http://dx.doi.org/10.1145/2729094.2742620

2. BACKGROUND

Many reports have been published about online SQL tutoring/assessment tools. However, most of these reports focus on the functionality of the tool itself, or on how the system supports a certain pedagogical model (e.g. Brusilovsky *et al.*, 2008 & 2010; Mitrovic, 1998 & 2003; Prior *et al.*, 2004 & 2014). Those reports do not analyze the data collected by those systems to determine the difficulties students have with SQL queries.

From the literature, we identified only Reisner (1981) as a published systematic study of student difficulties with SQL queries. She found that the subjects in her study had difficulty recognizing when they should use "group by", but her subjects could successfully use "group by" when explicitly told to do so. Reisner's focus was on comparing SQL with three other query tools that were then seen as competitors to SQL. Thus, the "group by" case was her only discussion of an aspect of SQL that was troublesome for novices. Over 30 years later, we believe a new study is warranted that focuses on SQL and looks at multiple aspects of SQL.

We also found some papers in which the authors mentioned their intuitions about student difficulties, based upon their teaching experiences. These intuitions were mentioned briefly in the introductory/motivation sections of the authors' papers, before those authors went on to describe the architecture of their tutoring/assessment system. Kearns *et al.* (1997) and also Mitrovic (1998) mentioned "group by" as a problem, especially aggregate functions and the use of "having". They also mentioned as a problem the complete specification of all necessary "where" conditions when joining multiple tables. Sadiq *et al.* (2004) nominated the declarative nature of SQL as a problem for students as it "*requires learners to think sets rather than steps*" (p. 224).

3. ASSESQL

We collected our data in a purpose-built online assessment system, AsseSQL. It is therefore necessary to describe some aspects of how AsseSQL works, so that readers can assess for themselves the validity of our data. In this paper, we provide the briefest possible description of the system. Further details of AsseSQL, and how we used it to test our students, can be found in prior publications (Prior *et al.*, 2004 & 2014). The students in this study were all undergraduates at our university, studying Bachelor degrees in Information Technology or Software Engineering.

In the online test, students were allowed 50 minutes to attempt seven SQL questions. On their first test page in AsseSQL, students see all seven questions, and they may attempt the questions in any order. All seven questions refer to a database that is familiar to the students prior to their test.

Each question is expressed in English, and might begin, "*Write an SQL query that …*" A student's answer is in the form of a complete SQL "select" statement.

When a student submits a "select" statement for any of the seven questions, the student is told immediately whether their answer is right or wrong. If the student's answer is wrong, the system presents both the desired table and the actual table produced by the student's query. The student is then free to provide another "select" statement, and may repeatedly do so until they either answer the question correctly, run out of time, or choose to move on to a different question. If a student moves to a new question, the student may return to this question later.

The grading of the answers is binary – the student's attempt is either correct or incorrect. As there may be more than one correct SQL statement for a specific question, a student's SQL statement is considered to be correct when the table produced by the student's "select" statement matches the desired table. (Some simple 'sanity checks' are made to ensure that a student doesn't fudge an answer with a brute force selection of all required rows.)

Prior to taking a test, the students are familiarized with both AsseSQL and the database scenario that will be used in the test. About a week before the test, students receive the Entity Relationship Diagram (ERD) and the "create" statements for the scenario database. Note, however, that students are not provided with the data that will fill those tables, nor are they provided with sample questions for that database scenario. Several weeks before the actual test, students are provided with access to a 'practice test' in AsseSQL, which has a different scenario database from the scenario database used in the actual test.

4. A SCENARIO DATABASE: BICYCLE

One of the scenario databases that we use in the online SQL test is called "Bicycle", based on a database from Post (2001). This is the database from which most of the results described in this paper are generated. The Bicycle database is composed of four tables, as shown in the Entity Relationship Diagram (ERD) of Figure 1. A bicycle is made up of many components; each component can be installed in many bicycles. Each installed component on a bicycle is called a `bikepart'. Just one manufacturer supplies each component, but one manufacturer may supply many components to the store, so there is a 1:M relationship between Manufacturer and Component. There is a self-referencing relationship in Component, as one component may be made up of many other components.

5. PRELIMINARIES TO RESULTS

5.1 A Wrong Answer versus No Attempt

Suppose for a given question, one third of students answered the question correctly, one third attempted the question but never answered it correctly, and one third never attempted the question – what should we consider the success rate to be? There are two options:

- 33%, if non-attempts are treated as incorrect answers.
- 50%, if non-attempts are ignored.

In the results presented below, we have supplied both options when reporting success rates for one of the seven types of questions. In cases where we are comparing two different types of questions, the issue arises as to whether a student did not attempt one of the two questions because the student knew they could not do it, or because they ran out of time. For that reason, when comparing performance between question types, we have applied the option where non-attempts are ignored.

5.2 Differences in a Pool of Variations

In the online test, students are presented with seven questions. Each question requires a different type of SQL "select" statement (e.g. a "group by", a natural join, etc). Each of the seven questions presented to a specific student is chosen at random from a small pool. In this paper, to avoid confusion between the actual questions presented to a student, and the questions in a pool, we henceforth refer to the questions in a pool as "variations".

In this section we discuss differences in success rates among variations within a single pool. To make the discussion more clear, we will focus on a concrete example – the four "group by" variations of the Bicycle Database. We chose this example as these four "group by" variations exhibited the greatest differences in success rate of any pool. The performance of each of these "group by" variations is shown in Figure 2.

In Figure 2, the line beginning "Baseline" provides the aggregate statistics when the data from all four variations are combined. That baseline row shows that a total of 986 students were assigned one of these four variations. Of those 986 students, 742 students successfully provided a correct answer, which is 75% of the total number of students. While 986 students is a large sample, if we collected data from another set of students, of similar ability, the success rate for that new sample is likely to be at least a little different. Equally, Figure 2 shows that the success rate for Variation 1 is 87% and 70% for Variation 2, but if we collected data from more students, how likely is it that the success rate for Variation 1 would drop below the success rate for Variation 2? To address those sorts of issues, we assumed a normal distribution of student abilities and estimated a 95% confidence interval for the success rate of the baseline and each of the four variations, using a well known statistical technique – the offset to the upper and lower bounds of the confidence interval is 1.65 × Standard Eror (SE), where the SE is estimated as:

$$SE = \sqrt{\frac{Success\ Rate(1 - Success\ Rate)}{Total}}$$

For instance, the lower and upper bounds of the Success Rate for the baseline shown in Figure 2 are 72% and 78% respectively (i.e. the offset was estimated as being 3%). That confidence interval and the confidence intervals for all four variations are also represented graphically in Figure 2. The colors in the confidence intervals of the variations show the proportion of those confidence intervals that are outside the baseline confidence interval.

In Figure 2, the column headed "p-value" indicates the probability that the difference in the success rate of a variation from the baseline is due to the existing sample being atypical. Those p-values indicate that the success rates of variations 1 and 3 are likely to remain higher than the baseline, and variation 4 remain lower, if another sample of data was collected from an equivalent group of students. The column headed "improvement" shows the difference in success rate between each variation and the baseline. The 95% confidence interval shown in that "improvement" column was calculated the same way as above.

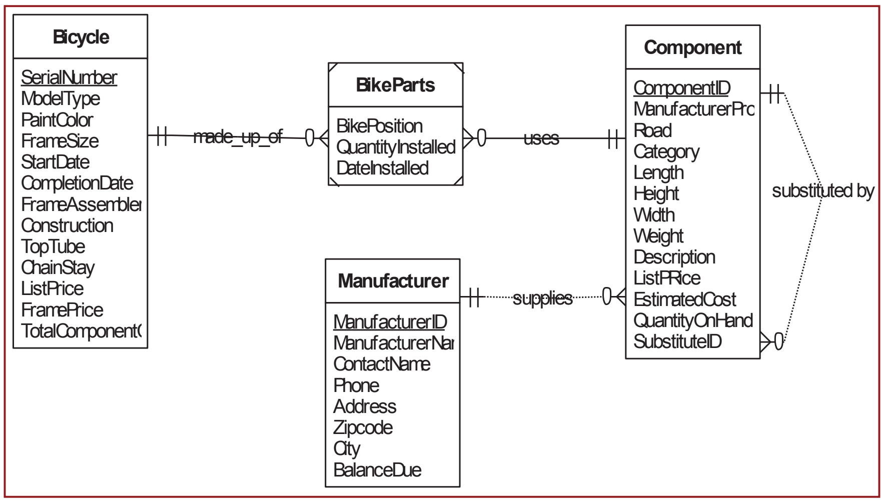
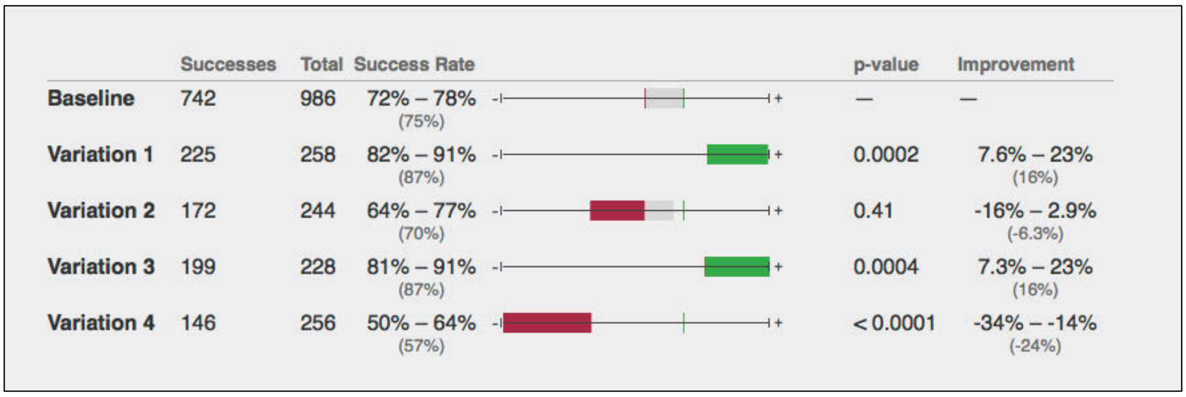


Figure 1. Bicycle Database ERD.



	Successes	Total	Success Rate	p-value	Improvement
Baseline	742	986	72% – 78% (75%)	—	—
Variation 1	225	258	82% – 91% (87%)	0.0002	7.6% – 23% (16%)
Variation 2	172	244	64% – 77% (70%)	0.41	-16% – 2.9% (-6.3%)
Variation 3	199	228	81% – 91% (87%)	0.0004	7.3% – 23% (16%)
Variation 4	146	256	50% – 64% (57%)	< 0.0001	-34% – -14% (-24%)

Figure 2. Variation in Success Rate for the "group by" pool.

The analysis illustrated in Figure 2 indicates that, even among questions considered to be variations that test the same fundamental SQL concept (i.e. "group by"), the success rates of the variations can be quite different. (Recall, however, that the "group by" example illustrated here manifested the greatest differences of any set of pool variations.) The reason for those differences is an issue we will return to in the "Results" section.

6. RESULTS

Table 1 summarizes the performance of students on the Bicycle database. In AsseSQL, the order in which the seven questions are presented on the computer screen to the students is fixed. (Recall, however, that students may attempt the questions in any order.) That same ordering is used in Table 1. Our choice of this ordering reflects our *a priori* intuition (i.e. when AsseSQL was built), of the relative difficulty of the seven query types.

As a general rule of thumb, based upon the estimation of the 95% confidence interval described in the previous section, the success rate percentages in Table 1 that differ by more than 5% can be considered to be significantly different. Table 1 shows that, in general, our intuition was correct – the success rate tends to fall from question type 1 to 7. Our intuition proved to be incorrect at the bottom of the table, as students were substantially less successful on self-joins then they were on correlated sub-queries.

Table 1. Query Success rates for various types of queries on the Bicycle database.

Type of "Select" statement required in answer		No. of question variations	Success Rate: non-attempts wrong	Success Rate: non-attempts ignored	Non-attempts
1	Simple, one table	6	89%	90%	1%
2	"group by"	4	74%	75%	2%
3	"group by" with "Having"	4	58%	61%	5%
4	Natural Join	3	57%	61%	6%
5	Simple subquery	4	53%	58%	9%
6	Self-join	3	18%	24%	23%
7	Correlated subquery	6	39%	46%	16%

6.1 Simple Queries on a Single Table

We begin a more detailed analysis by considering the simplest queries in the system: queries made on a single table, using only the reserved words "select", "from", "where" and "and". Table 1 summarizes student performance on these queries for the Bicycle database (see row 1). For that particular database, there are six variations. AsseSQL randomly assigned one of these six queries to each student.

As Table 1 shows, 90% of the students were able to provide a correct query. That students did so well is not surprising, given the simplicity of this type of query, but it does establish that at least 90% of the students were able to understand the English-language instructions given by AsseSQL (which could not be taken for granted, since many of our students have English as a second language) and that 90% of the students are competent users of AsseSQL.

6.2 "Group By"

Table 1 shows that around 75% of students were able to answer a question that required "group by". However, as described in the method section, the "group by" pool exhibited the greatest differences in success rate of any set of variations.

One possible explanation for the difference in the success rate is the linguistic complexity of the four variations, since English is the second language of many of our students. However, Table 2 shows that there is no clear relationship between success rate and linguistic complexity, when linguistic complexity is estimated by the number of words in each variation. In fact, the variation with the lowest success rate also has the lowest word count.

Table 2. Success rates for the "group by" variations.

Variation	**Success Rate**: non-attempts considered wrong	Word count	Signal words
1	87%	17	"average"
2	70%	24	"average total"
3	87%	19	"average"
4	57%	16	"number of"

On inspection of the text of the four questions represented in Figure 2, another explanation suggests itself as to why variation 4 was substantially harder. This explanation is consistent with Reisner's (1981) observation that the subjects in her study had difficulty knowing when to use "group by", but they could successfully use "group by" when explicitly told to do so. With the exception of variation 4, all the variations use the word "average", which is a clear signal to the student that the aggregate function "avg" is required, and hence "group by" is probably required. In contrast, the use of "number of" in variation 4 does not transparently signal that a specific aggregate function is required. In variation 2 the use of "average total" may have confused students, as those words signal two possible aggregate functions, which perhaps explains that variation's middling success rate.

This analysis of differences in the success rates of the "group by" variations demonstrates the pedagogical value of looking at the data collected by our "particle accelerator" (i.e. AsseSQL). Because of this analysis, our teaching team was led to discuss exactly what it is that we are looking to test when we ask a "group by" question. Is our goal to (1) test whether a student recognizes that "group by" is required, or (2) merely test whether a student can actually write such a query when they know that "group by" is required? If our goal includes the former, then variations 1-3 need to be reworded or replaced. If the latter is our goal, then variation 4 needs to be replaced.

6.3 "Group By" Queries with "Having"

Table 3 compares the performance of students on two types of "group by" queries – queries with "having" and queries without "having" (i.e. the queries in rows 2 and 3 of Table 1). Table 3 shows that these two types of queries are significantly correlated ($p < 0.0001$), but only moderately so (phi = 0.49). The 19% in the top right of Table 3 may understate how much difficulty students have with "having", as that is a percentage of all 986 students represented by the entire table. When the 186 in the top right cell is expressed as a percentage of the 742 in the top row, the figure is 25%. That is, a quarter of all students who could provide a correct "group by" without a "having" could not provide a correct "group by" that required a "having".

Table 3. Comparison of "group by" with and without "having". (N =986; phi correlation 0.49; $\chi 2$ test $p < 0.0001$)

	with "having" right	with "having" wrong
"group by" right	556 (56%)	186 (19%)
"group by" wrong	49 (5%)	197 (20%)

6.4 Natural Join and Self-join

Table 4 shows that the correlation between natural joins and self-joins is a moderate 0.41. The 38% in the top right of Table 4 is a percentage of all 986 students represented by that table. When the 371 in that top right cell is expressed as a percentage of the 599 students in the top row, the figure is 62%. That is, 62% of all students who could answer a natural join could not provide a correct self-join.

Table 4. Comparison of natural join and self-join. (N = 599; phi correlation 0.41; $\chi 2$ test $p < 0.0001$)

	self-join right	self-join wrong
natural join right	228 (23%)	371 (38%)
natural join wrong	9 (1%)	380 (38%)

Table 5. Comparison of simple and correlated sub-queries. (N =986; phi correlation 0.49; $\chi 2$ test $p < 0.0001$)

	Correlated right	Correlated wrong
Simple right	387 (39%)	189 (19%)
Simple wrong	71 (7%)	341 (35%)

6.5 Simple and Correlated Sub-queries

Table 5 shows, perhaps unsurprisingly, that the correlation between simple and correlated sub-queries is a moderate 0.49. The 189 (19%) in the top right of Table 5 is a percentage of all 986 students represented by that table. Expressed as a percentage of the 576 students in the top row, this figure is 33%. That is, one third of all students who could provide a correct simple sub-query could not provide a correct correlated sub-query.

6.6 Generalizing to Other Databases

We also collected data from students for two other databases, of similar complexity to the Bicycle database. Figure 3 compares the success rates for each query type in the Bicycle database with the success rates for those other two databases. (The numbers on the horizontal axis refer to the row numbers in Table 1.) All three databases show approximately the same pattern. The success rate for self-joins (point 6 on the horizontal axis) is the lowest success rate for all three databases. The only major difference between the three databases is that the "having" questions (point 3 on the horizontal axis) were especially difficult in "Bicycle2".

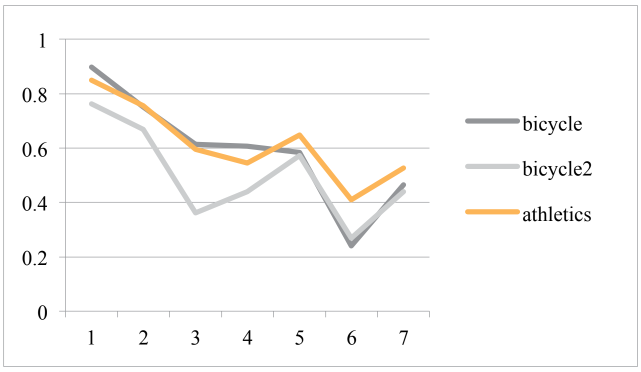


Figure 3. Success rates of the Bicycle database compared with two other databases.

6.7 Number of Attempts

In the "real world", nobody knows the desired result table before they write their query. In that sense, AsseSQL and most other online SQL testing systems are unnatural environments, as these systems provide the student with the desired table. We have therefore wondered whether a large portion of students may have answered questions correctly, particularly the easier questions, by brute force – that is, by making many attempts at the question, with quasi-random changes.

Analysis shows that this does not appear to be the case. For all three databases, the correlation (Pearson) between the median number of attempts at a question and that question's success rate is between -0.6 and -0.7, which is a strong negative correlation. The average number of attempts also exhibited a strong negative correlation between -0.6 and -0.7 for all three databases.

For the Bicycle database, this negative correlation is illustrated in Figure 4. The middle plot in that figure indicates the median number of attempts by the students who eventually provide a correct answer. A surprising aspect of Figure 4 is that students required relatively few attempts for the correlated sub-query. Note, however, that in this figure (and throughout this section of the paper) we are only considering students who eventually provide a correct answer. Thus, the figure merely shows that 39% of students (as shown in Table 1) who did provide a correct correlated sub-query were able to do so in relatively few attempts. The speed of those students on correlated sub-queries further emphasizes the difficulty of self-joins, since only 18% of the students could provide a correct self-join, half of whom needed 9 or more attempts, with a quarter requiring 16 or more attempts.

Figure 5 compares the median number of attempts needed to answer questions correctly in each of the three databases. All three databases show approximately the same pattern. There are two major differences across the three databases: (1) the number of attempts for simple sub-queries (i.e. point 5 on the horizontal axis) is relatively low in the "Bicycle2" and "athletics" databases, and (2) the number of attempts for self-joins (i.e. point 6 on the horizontal axis) is relatively low in the "athletics" database.

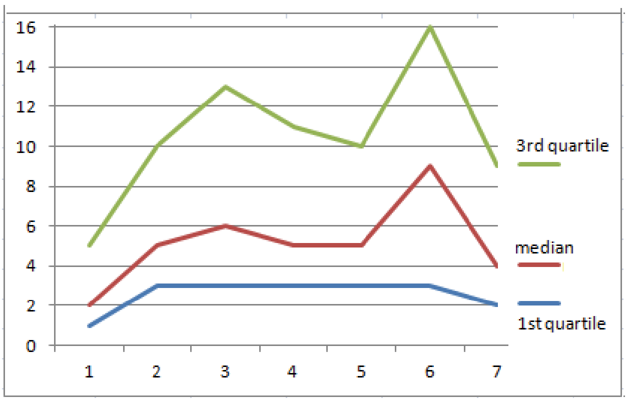


Figure 4. The distribution of number of attempts needed to answer each question type correctly in the Bicycle database.

In AsseSQL, we do not distinguish between attempts that are wrong because of a syntactic error, and attempts that are syntactically correct but return an incorrect table. It would be interesting to study data where that distinction is made.

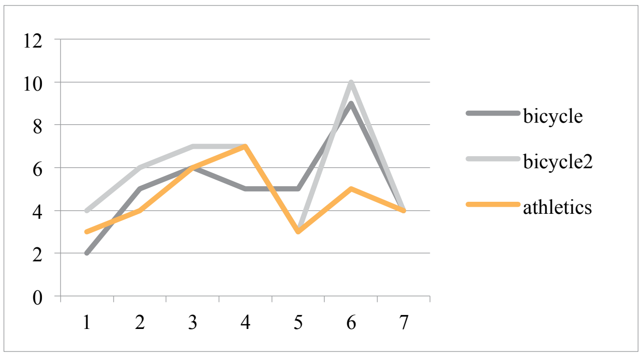


Figure 5. The median number of attempts needed to answer questions correctly in each of the three databases.

7. DISCUSSION

Many of our quantitative results are probably consistent with the intuitions of experienced database educators. It is important, however, that intuitions are tested. (After all, to return to Agarwal's metaphor quoted in the introduction, the Large Hadron Collider was expected to find the Higgs' Boson).

Perhaps our most surprising result – it was certainly a surprise to the authors of this paper – is that students found self-joins to be the most difficult of the query types we tested. Our intuition from many years of teaching databases was that correlated sub-queries were the most difficult type of query. Furthermore, we were surprised that our students found self-joins to be far harder than simple sub-queries, when these two types of queries can often be used interchangeably. A slightly surprising result was the extent of the difficulty of "having". While we suspected from experience that "having" troubled many students, we were surprised that a quarter of our students who could answer a "group by" that did not require "having" could not also correctly use "having".

Why are self-joins and (to a lesser extent) "having" troublesome for so many students? Our students have an equal opportunity to practice all the query types, so we are less inclined to believe that the trouble lies with insufficient practice. (But the prevalence in the database education literature of online tutoring systems may imply that the many educators believe that the problem is a lack of practice.) We are more inclined to believe that self-joins and "having" expose a conceptual problem for students. In the traditional teaching of SQL queries, the early emphasis is on the concept of a table. This leads to students thinking of operations on database tables as being much like the "real world" tables, such as spreadsheets. But self-joins and "having" have no common "real world" analog, so those operations are troublesome for many students. Reisner (1981) hypothesized a similar explanation, specifically about "group by". She wrote "*We suspect that ...* [students] *... adopt an "operations-on-tables" strategy ...* [which] *... does not work for the "group by" function, which requires users to think in terms of partitioning a table into subgroups*". As many database educators know, the foundational data structure for databases is the row, not the table, and database educators need to communicate that knowledge explicitly to their students. Furthermore, while we agree with Sadiq *et al*. (2004) that the declarative nature of SQL is a problem for some students, we suspect that it is especially a problem in the very early stages of learning SQL, and after students have begun to acquire that declarative understanding on simpler SQL queries, they are then taught "having", self-joins and correlated sub-queries, which do require a procedural grasp of SQL.

Part of the contribution of our work is the establishment of a method for studying the difficulties students have with SQL queries. Our principal methodological contribution is our focus on the relative difficulties students have with SQL queries. For example, of our students who could provide a correct simple sub-query, one third could not provide a correct correlated sub-query, but it would be absurd to claim that this same ratio (i.e. one third) is universal. Clearly, that ratio will vary from database to database, from institution to institution, and even semester to semester within an institution. What we are claiming is that, when a cohort of students manifests significantly different success rates at (for example) simple and correlated sub-queries, then it will be the correlated sub-query that exhibits the lower success rate. (And likewise for other pairs of query types.) Note that our claim is not negated by cohorts that do not display any difference in success rates on simple and correlated queries. Clearly, an immature or low achieving cohort will find both simple and correlated sub-queries to be difficult, while a mature or high achieving cohort will find both simple and correlated sub-queries to be easy. What our quantitative results suggest are developmental stages in learning SQL – for example, competence at simple sub-queries precedes competence at correlated sub-queries. (And likewise for other pairs of query types.)

As another methodological issue, we advocate that prior to studying the relative difficulty of two different query types, a third query type be used as a screening test. In our study, a simple select on a single table (i.e. row 1 of Table 1) served that purpose. The purpose of the screening test is to establish that students have an interesting minimum level of knowledge. In our case, since 90% of our students met the minimum requirement, and since students in AsseSQL can answer questions in any order, we elected not to remove that 10% from our analysis. However, in future studies, it may be useful to conduct the screening test as a pre-test, and remove from the data those students who fail the screening test, especially when the percentage of students who fail the screening test is much greater than 10%.

Our analysis of the four variations in the "group by" pool points to one interesting future research direction. In that pool, three of the four variations clearly signaled that a "group by" was required, while the fourth variation did not. It would be interesting to verify and extend upon Reisner's (1981) observation, that her subjects had difficulty recognizing when they should use "group by", but they could successfully use "group by" when explicitly told to do so – does that observation hold for any other query types?

8. CONCLUSION

At the back end of many internet applications there is a database. However, the prevalence and importance of databases is not reflected in the computing education literature. There certainly is education literature on databases, but nothing like the literature on (for example) learning to program. Furthermore, most of the existing database education literature focuses on the architecture of online tutorial and assessment systems. There is very little literature on what students find difficult about writing database queries. This paper is the first published quantitative study of the relative difficulty for novices of different types of SQL queries. This paper provides a quantitative and methodological foundation upon which further studies may be built.

9. REFERENCES

Brusilovsky, P., Sosnovsky, S., Lee, D., Yudelson, M., Zadorozhny, V., and Zhou, X. (2008) *An open integrated exploratorium for database courses*. ITiCSE '08. pp. 22-26. http://doi.acm.org/10.1145/1384271.1384280

Brusilovsky, P., Sosnovsky, S., Yudelson, M. V., Lee, D. H., Zadorozhny, V., and Zhou, X. (2010) Learning SQL Programming with Interactive Tools: From Integration to Personalization. *Trans. Comput. Educ.* 9, 4, Article 19 (January 2010). http://doi.acm.org/10.1145.1656255.1656257

Kearns, R., Shead, S. and Fekete, A. (1997) *A teaching system for SQL*. ACSE '97. pp. 224-231. http://doi.acm.org/10.1145/299359.299391

Mitrovic, A. (1998). *Learning SQL with a computerized tutor*. SIGCSE '98, pp. 307-311. http://doi.acm.org/10.1145/273133.274318

Mitrovic, A. (2003) *An Intelligent SQL Tutor on the Web*. Int. J. Artif. Intell. Ed. 13, 2-4 (April 2003), pp. 173-197.

Post, G.V. (2001) *Database management systems: designing and building business applications*. McGraw-Hill.

Prior, J., and Lister, R. (2004) *The Backwash Effect on SQL Skills Grading*. ITiCSE 2004, Leeds, UK. pp. 32-36. http://doi.acm.org/10.1145/1007996.1008008

Prior, J. (2014) *AsseSQL: an online, browser-based SQL skills assessment tool*. ITiCSE 2014. pp. 327-327. http://doi.acm.org/10.1145/2591708.2602682

Reisner, P. (1981) *Human Factors Studies of Database Query Languages: A Survey and Assessment*. ACM Comput. Surv. 13, 1 (March), pp. 13-31. doi.acm.org/10.1145/356835.356837

Sadiq, S., Orlowska, M., Sadiq, W., and Lin, J. (2004) *SQLator: an online SQL learning workbench*. ITiCSE '04. pp. 223-227. http://doi.acm.org/10.1145/1007996.1008055

Stokes, P. (2013) *The Particle Accelerator of Learning*. Inside Higher Ed. https://www.insidehighered.com/views/2013/02/22/look-inside-edxs-learning-laboratory-essay

7.3 Discussion

In this chapter I performed correlation analysis on the mark of different questions of an online SQL test. The findings of this chapter demonstrated the validity of the statistical approach we used in Chapter 5 in another context. However, before I move on to the application of machine learning tools in predicting struggling students in (Chapter 9, I attempt to better understand the data to see what the number of attempts and correctness mean in this context. These are what I learned from the findings in Chapter 6.

Chapter 8

Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries

8.1 Introduction

The aim of this chapter is to better understand the meaning of the features extracted and analyzed from the source code snapshots collected from SQL novices *SELECT*. Given my discussion on the results presented in the paper allocated to Chapter 6, it is important to understand the data before inputting it to the machine learning algorithm. This helps us to have a *sense* of the data. In this Chapter, I demonstrate what the precise definition of an *attempt* is in this context and how it could relate to the ways the novices complete the construction of the SQL statement, including specific aims and hypotheses.

8.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;

2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;

3. there are no other authors of the publication according to these criteria;

4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and

5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries

Conference: ACM Conference on Innovation and Technology in Computer Science Education

URL: http://dl.acm.org/citation.cfm?id=2899464&CFID=862572254&CFTOKEN=60379727

Status: Presented, July 2016

TABLE 8.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 8

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Julia Prior				✓
Vahid Behbood	✓	✓		
Raymond Lister	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi
Name Signature Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister

Name	Signature	Date

8.2 PDF of the Published Paper

Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries

Alireza Ahadi, Julia Prior, Vahid Behbood and Raymond Lister
University of Technology, Sydney, Australia

{Alireza.Ahadi, Julia.Prior, Vahid.Behbood, Raymond.Lister}@uts.edu.au

ABSTRACT
Computer science researchers have studied extensively the mistakes of novice programmers. In comparison, little attention has been given to studying the mistakes of people who are novices at writing database queries. This paper represents the first large scale analysis of students' semantic mistakes in writing different types of SQL SELECT statements. Over 160 thousand snapshots of SQL queries were collected from over 2300 students across nine years. We describe the most common semantic mistakes that these students made when writing different types of SQL statements, and suggest reasons behind those mistakes. We mapped the semantic mistakes we identified in our data to different semantic categories found in the literature. Our findings show that the majority of semantic mistakes are of the type "omission". Most of these omissions happen in queries that require a JOIN, a subquery, or a GROUP BY operator. We conclude that it is important to explicitly teach students techniques for choosing the appropriate type of query when designing a SQL query.

Categories and Subject Descriptors
H.2.3 [**Database Management**]: Language – query languages.

General Terms
Performance, Human Factors.

Keywords
Online assessment; databases; SQL queries.

1. INTRODUCTION
The Structured Query Language (SQL) is the standard language for relational and object-oriental databases, as well as the industry standard language for querying databases. As with other computer languages, SQL queries can be semantically or syntactically wrong. However, limited attention has been given to understanding novice programmers' challenges in writing correct SQL queries [5]. A deep understanding of the common semantic mistakes that novices make when writing SQL queries will improve teaching and learning outcomes.

In this paper, we use data collected over nine years from ~2300 students taking online SQL exams. One of the ways that we have analyzed this data is to qualitatively study the semantic mistakes committed by these students. We review these mistakes in seven different types of SQL queries and investigate the reasons behind them. More specifically, we map these mistakes to proposed mistake categories introduced in the literature, and explain why students are likely to make these mistakes.

In section 2, we review the literature on analysis of errors in SQL. In Section 3, we describe the data collection and analysis of the data to explore the types of SQL query errors made by students. In section 4, we review our findings on common semantic mistakes of novices in writing different SQL SELECT statements. Section 5 expands and discusses these findings, before our conclusions are given in Section 6.

2. RELATED WORK
Most computing education researchers who have studied database education have focused on tutoring and/or assessment tools. Most of that work has been concerned with the functionality of the tool itself, or on how the system supports a certain pedagogical model [1-4]. There has been relatively little work on novice errors and misconceptions when using SQL.

A few studies have investigated these challenges with programming in SQL. Reisner performed the first experimental study investigating SEQUEL, the predecessor of SQL [7]. In that study, a series of psychological experiments were conducted on college students to investigate learnability of the language, as well as the type and frequency of errors made by subjects. Reisner categorized students' mistakes into "intrusion", "omission", "prior-knowledge", "data-type", "consistency" and "over-generalization".

Welty and Stemple [8] explored users' difficulty in writing queries in SQL compared to TABLET. Their comparison revealed that constructing difficult queries in more procedurally-oriented languages was easier than less procedurally-oriented languages. They categorized SQL statements into "correct", "minor language error", "minor operand error", "minor substance error", "correctable", "major substance error", "major language error", "incomplete" and "unattempted", where the first four of those categories were considered *essentially correct* and the other five categories were classified as *incorrect*. Their categorization of the SQL statements was based on Reisner's categorization [7]. A few years later, Welty ran an experiment on a small group to test how assistance with error correction would affect user performance

ITiCSE '16, July 09-13, 2016, Arequipa, Peru
© 2016 ACM. ISBN 978-1-4503-4231-5/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2899415.2899464

[9]. In that study, he categorized subject responses into “correct”, “minor error in problem comprehension”, “minor syntactic”, “complex errors”, “group by”, “s-type error”, “incorrect” and “unattempted”.

Buitendijk [10] introduced a classification of natural language questions, as well as possible errors within each class which resulted in four general groups of logical errors including “existence”, “comparison”, “extension” and “complexity”. This categorization is not only based on SQL anomalies, but also focuses on user mistakes.

Smelcer [11] developed a model of query writing that integrated a GOMS-type analysis of SQL query construction with the characteristics of human cognition. This model introduced four common cognitive causes of JOIN clause omission and resulted in the categorization of common mistakes in writing SQL queries to “omitting the join clause”, “AND/OR difficulties”, “omitting quotes”, “omitting the FROM clause”, “omitting qualifications”, “misspellings” and “synonyms”. Brass [12] reports an extensive list of conditions that are strong indications of semantic mistakes. However, none of these studies analysed student mistakes in large datasets.

3. METHOD

3.1 Snapshot Collection

The data collected in this study forms a total number of ~161000 SQL SELECT statement snapshots from ~2300 students. Each snapshot is of one student attempt at a particular test question. The students in this study were all novice undergraduate students enrolled in an introductory database course. The tool used to collect the data is a purpose-built online assessment system named AsseSQL. Further details on the tool and how it was used to test the students can be found in prior publications [5, 6]. These snapshots were generated during supervised 50 minute online tests in which students attempted to answer seven SQL questions based on a given case study database. Students were provided with the case study, which included the description of the database, the Entity Relationship Diagram (ERD), and the CREATE statements corresponding to the database tables. Each question tests a student’s ability to design a SELECT statement that covers a specific concept. Table 1 shows the concepts covered in the online test and statistics of snapshots related to each concept. A more detailed explanation on the nature of these concepts and their relative difficulty levels can be found in an earlier publication [13].

Table 1. Different SQL concepts and the number of snapshots.

Concept	Snapshot count
Group by with having	~32k (20%)
Self-join	~27k (17%)
Group by	~25k (15%)
Natural join	~24k (15%)
Simple subquery	~19k (12%)
Simple, one table	~18k (11%)
Correlated subquery	~16k (10%)

3.2 Snapshot Categorization

In order to produce the execution result of the collected snapshots, all snapshot SQL statements were re-executed in PostgreSQL and the output of each snapshot was obtained. Depending on the output returned by the PostgreSQL and the marking results of AsseSQL, each snapshot was tagged as *correct*, *syntactically wrong* or *semantically wrong*. We categorized each snapshot as a) correct if its result set was exactly the same as desired solution for the question corresponding to the snapshot, b) syntactically wrong when an error message was returned by the PostgreSQL, or c) semantically wrong when the execution of snapshot resulted in either an empty result set or a result set which was not exactly the same as the desired solution for the question corresponding to the snapshot. In this study, a student’s snapshot is considered to be semantically incorrect if the output generated by the snapshot is different from the correct output. The categorization of the snapshots and its breakdown for each concept is shown in Table 2. While some of the snapshots in each level are correct, the majority of snapshots introduce an error (Figure 1). A detailed exploration of the reasons behind the syntactic errors is available in an earlier publication [14].

Table 2. Categorization of snapshots based on their output and their breakdown for different SQL concepts.

Concept	Correct	Syntactically wrong	Semantically wrong
Group by with having	4%	58%	37%
Self-join	2%	37%	61%
Group by	7%	63%	30%
Natural join	4%	64%	32%
Simple subquery	5%	61%	34%
Simple, one table	11%	48%	41%
Correlated subquery	6%	52%	42%
Among all snapshots	6%	54%	40%

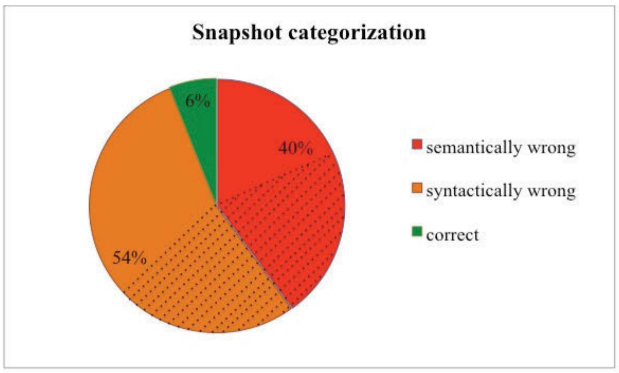


Figure 1. Categorization of collected SQL snapshots (N = ~161k). Snapshots in the hatched area were generated by students who subsequently fixed their error.

3.3 Semantic Mistake vs. Syntactic Error

As could be seen in Table 2 above, a considerable number of snapshots introduce an error, with the majority of snapshots falling into the syntactic error category. As the number of syntactic errors observed in students’ attempts is higher than number of semantic mistakes, one could argue the importance of syntactic errors over semantic mistakes. However, in this study we chose to investigate the snapshots with semantic mistakes for the following reasons. Firstly, our analysis suggests that syntactic errors are more likely to be the result of lack of practice or carelessness. This is supported by the fact that among all syntactic error snapshots, ~69% of them are due to a typing errors in the SELECT statement (N = ~61000). These typing errors are due to wrong column names, wrong table names, or wrong syntax in one or more clauses of the SELECT statement. Excluding the syntax errors due to typing mistakes, the majority of mistakes made by novices are semantic. The second reason is related to the last snapshots generated by the unsuccessful students, which reflect the point where students stopped trying to get a query right. The number of students' last attempts with semantic mistakes is almost three times more than the number of last attempts that are syntactically incorrect (including typing mistakes). Another reason supporting our decision is that a SQL query that suffers from a syntactic error might already have a semantic mistake encapsulated in it. Our preliminary result suggests that a considerable number of *fixed* syntactic error snapshots are not correct as they include semantic mistakes (N = ~21000). Finally, the error code and the error message returned by PostgreSQL are usually enough to indicate the reason for a syntax error. In contrast, the output of a query that suffers from a semantic mistake is not as easily diagnosed as a syntactic error. This makes a semantic mistake much harder to fix.

3.4 Selection of Database Case Study

The set of ~161000 snapshots in this study are based on three different database case studies that are used in the online tests. The ERD structure as well as data complexity of these three databases are very similar. They consist of four to five tables, including one associate relation and three to four relationships, one of which is a unary relationship. The success rates of these case studies are only slightly different (Figure 2). The relative numbers of syntactic errors generated by students among these case studies are similar, however, the relative number of semantic mistakes made by students differs from one database case study to another. To make the result of our work less dependent on the comparative difficulty level of these case studies, we decided to limit our investigation dataset to the set of snapshots collected from only one database case study. We selected database case study 'Bicycle', which is based on a database from Post [15]. This database case study has the highest median success rate among different SQL query types, which reflects its lower level of difficulty relative to the other two case studies. For this case study we have ~45000 snapshots collected from ~700 students.

3.5 Primary Cause of Semantic Errors

According to Figure 1, the vast majority (94%) of snapshots do not generate the correct result, due to either a syntactic error or a semantic mistake. Almost half of these incorrect snapshots (46%) are from students who eventually were able to correct the errors and answer the question correctly. Figure 3 shows the frequency of incorrect snapshots **as a function of the attempt number** for these students who did eventually answer a question correctly. Among ~3200 cases where a student was able to answer a question correctly, 57% of students constructed at least one semantically incorrect SQL statement before they produced the correct SQL snapshot. Hence, we conclude that not all semantically wrong snapshots/student attempts are unfixable. This is also reported by Ogden *et al.* [16]. As a result, we elected to identify and investigate only those semantic mistakes that students were not able to correct. We limited our investigation to the set of snapshots (N = 551) that are final attempts with a semantic error. Those snapshots were generated by 321 students.

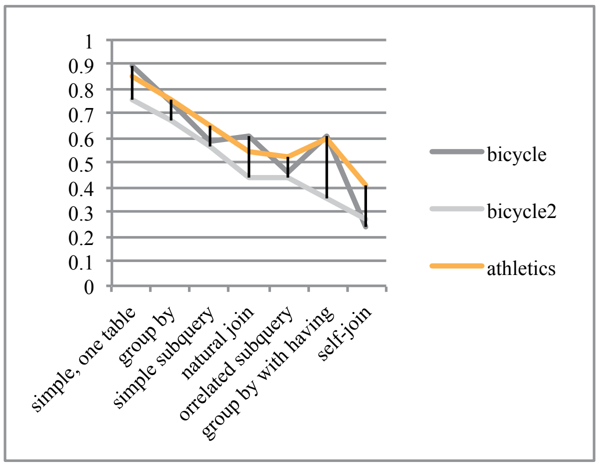


Figure 2. Success rates of different database case studies.

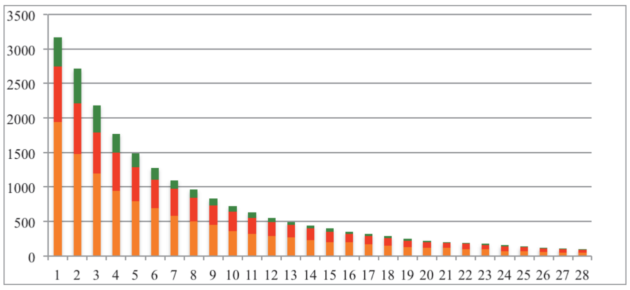


Figure 3. Distribution of correct (green), semantically incorrect (red) and syntactically incorrect (orange) snapshots, as a function of the number of attempts. X axis represents the nth attempt and Y axis represents number of snapshots.

A snapshot may contain multiple semantic mistakes. We focused upon the primary semantic mistake. Our process for manually identifying the primary semantic mistake in the 551 snapshots is illustrated in the following example:

List the given name, family name, address, the employee ID and the number of managers of those employees who are less than 25 years and have been managed by more than one manager.

The correct query for such a question could be:

SELECT *EMP_ID*, *FNAME*, *LNAME*, *ADDRESS* , COUNT(*EMPID*) AS *NUMBER_OF_MANAGERS FROM EMP_DETAILS* NATURAL JOIN *EMP_MANAGER* WHERE *EMP_AGE* < 25 GROUP BY *EMPID* HAVING COUNT(*EMP_ID*) > 1;

A student's attempt for this question might be:

SELECT * FROM *EMP_DETAILS* WHERE *EMPAGE* < 24;

This SELECT statement has multiple semantic mistakes; however, the most important mistake is the absence of the EMP_MANAGER relation in this query. Even if all other aspects of the given query related to EMP_DETAILS table are corrected, this query will not produce the correct result set unless the EMP_MANAGER table is included in the query. Thus the principle semantic error is the omission of the table.

4. RESULTS

Queries of type *self-join* have the highest number of semantically wrong last attempts (178 snapshots). Questions of type *natural join*, *group by with having* and *correlated subqueries* have the second highest frequency. The *simple with one table* and *group by* categories have the lowest frequencies. In all categories, the main reason behind the semantic mistake is students' lack of skill in identifying the required type of query (as listed in Table 2). Table 3 reviews different clauses of a SELECT statement and the mistakes allocated to those clauses.

Table 3. Clause based categorization of principle semantic mistakes.

Clause	Mistake/s	Concepts
where (46%)	Missing/wrong condition	Simple, self-join, correlated subquery, join
from (26%)	Self-join not used	Self-join
having (13%)	Missing group by or having clause, use of wrong column	Having
order by (5%)	Missing order by clause, incorrect/incomplete column	Simple, group by
select (5%)	Missing/extra column	Simple, group by
group by (5%)	Missing group by clause, use of wrong column	Group by, group by with having

4.1 Simple Query with One Table

Construction of simple queries with one table is the first concept that novices learn at the authors' institute. The students investigated in this study have the highest success ratio for questions of this type. However, snapshots that belong to this category also suffer from semantic mistakes. The most common mistakes observed for this query type includes missing/unnecessary columns in a SELECT clause or an ORDER BY clause. The absence of an appropriate condition or the presence of irrelevant condition in a WHERE clause is also a common mistake. In a few cases, the GROUP BY clause was irrelevantly used. The majority of these snapshots fall into Reisner's category of "essentially correct", as they are mostly the result of carelessness.

4.2 Group By

Four common mistakes were identified, including mistakes related to the GROUP BY clause, SELECT clause, WHERE clause or ORDER BY clause. The most frequent mistakes include missing the GROUP BY clause entirely or including unnecessary columns in the GROUP BY clause. Wrong conditions in the WHERE clause and a missing ORDER BY clause were also among the most frequent mistakes for GROUP BY queries. Also, a missing aggregate function in the SELECT clause was a common mistake, which has direct correlation with the absence of a signal word in the question (e.g. "average" or "sum"). This was originally observed in Reisner's experiments and further supported by the findings of Ahadi et al. [13].

4.3 Simple Subquery

More than 80% of the investigated snapshots in this category did not use a subquery at all. This could indicate that either students were not able to identify the skill required to answer these questions, or they did not have the skill required to construct such a query.

4.4 Correlated Subquery

More than 70% of snapshots in this category lacked the structure of a correlated subquery. In other words, the identification by students of the need for a correlated subquery was the most common problem. A further 13% of snapshots followed the syntax of a simple subquery. The rest of the snapshots included a wrong condition in the WHERE clause of the inner query.

4.5 Join

More than 80% of the snapshots in this category lacked the JOIN clause, that is, either the ON clause or the JOIN clause in the WHERE clause was missing. This is perhaps explained by the fact that the use of "NATURAL JOIN" in a SQL SELECT does not require the specification of the joining keys in the query. As a result, students who wrote the join queries may have forgotten that, unlike the NATURAL JOIN, the INNER JOIN requires the indication of the keys involved in the join. Unnecessary use of aggregate functions and incorrect joining conditions in the WHERE clause were also observed.

4.6 Group By with Having

Around 75% of the snapshots in this category lacked a HAVING clause and 15% of them did not even include the GROUP BY clause. Upon closer inspection, we noticed that the condition that is supposed to appear in the HAVING clause was often mistakenly written in the WHERE clause. Around 15% of the snapshots included the wrong conditions in the HAVING clause.

4.7 Self-join

Around 75% of the snapshots in this category demonstrated that students did not understand that they needed a self-join. While the remaining snapshots had a self-join, the self-join lacked a condition in the WHERE clause by which the result set is limited to non-overlapping sections of the main table and its replicate. In rare cases, unnecessary/incorrect conditions were included in the WHERE clause.

4.8 Mapping Semantic Mistakes

We mapped the 551 snapshots to the categorizations provided by Reisner, by Welty and Stemple, by Welty, and by Buitendijk [7-10]. Note, however that most error categorizations in the literature are a mix of semantic and syntactic mistakes.

4.8.1 Psychological experiments and error categorization

Reisner has done most of the work in experimental analysis of learnability of SQL [7]. She categorized students' mistakes into "intrusion", "omission", "prior-knowledge", "data-type", "consistency" and "over-generalization". More than half of the observed semantic mistakes in the snapshots analyzed in this study are of type omission. This error usually happens when a signal word such as "average" is not given in the question. Our results support Reisner's experiment. While omissions were reported by Reisner, her observations were limited to GROUP BY omission, and also AND clause omissions. Welty's "incomplete" category is (to some extent) similar to Reisner's omission error category. Table 4 reviews different omission errors observed among the set of 551 snapshots in our study.

Table 4. Omission errors in semantically incorrect snapshots. The second number in each row represents the percentage among all omission errors.

category	Frequency
Omitting JOIN clause	218 (48%)
Omitting SUBQUERY	115 (25%)
Omitting HAVING clause	58 (12%)
Omitting ORDER BY clause	23 (5%)
Omitting GROUP BY clause	17 (3%)
Omitting aggregate function	8 (1%)
Omitting WHERE clause	6 (1%)
Omitting column in SELECT	3 (<1%)
Omitting DISTINCT	3 (<1%)
Omitting column in ORDER BY	2 (<1%)

4.8.2 Categorization based on ease of error correction

Welty's experiment classified subject's responses into multiple categories according to the amount of effort needed to correct their mistakes [9]. In our data, snapshots with a semantic mistake of type "minor spelling errors" were frequent. Most of the time, in our data, a misspelling resulted in a syntactic error – e.g. typing "SELCT" instead of "SELECT" – but there are cases where a misspelling results in a semantic mistake, for example, using the string 'hawaii' instead of 'Hawaii' when referring to data stored in a table.

A similar error to Welty's "minor spelling error" is the "overgeneralization" error proposed by Reisner. For a given question "List the names of employees…", an overgeneralized solution would be:

SELECT *NAMES* FROM *EMPLOYEE ... etc.*

The overgeneralization error happens when the information given in the question is directly extracted and used to construct the query, in the case above the *names* and *employees*. Reisner's overgeneralization category overlaps with Buitendijk's "comparisons" error category.

In some cases, snapshots of this type appeared immediately after a syntactically wrong attempt where the name of the columns or the tables were directly extracted from the question itself. Reisner classifies this kind of syntactic error as "intrusion error". The result of syntactic error analysis investigated by Ahadi *et al.*[14] reported that this type of mistake was common. In that study, syntactic errors due to prior knowledge – e.g. using AVERAGE instead of AVG – were also reported to be common.

Some semantic mistakes fall into Welty and Stemple's "major substance" error [8]. This error happens when the query is syntactically correct but the query answers a different question. Such errors are hard for students to detect and fix. Examples of these types of errors are a missing second column in an ORDER BY clause, an unnecessary condition in a WHERE clause, or the use of an incorrect aggregate function. This error was not common in our snapshots.

4.8.3 Natural Language based Categorization

Proposed by Buitendijk [10], this categorization of logical errors introduces "existence", "comparison", "extension" and "complexity" errors. Among our data, over one quarter of the semantic mistakes fall into the complexity category. An example of such cases is when more than one subquery needs to be included in the query.

Semantic mistakes due to the complexity of the written query are hard to detect. A query is regarded as complex if the translation the natural language question to the SQL answer is difficult. Interestingly, there is no relationship between the complexity of the given question and its corresponding answer. For example, longer questions do not necessarily require a complex query. However, longer queries are usually more complex and as a result more likely to contain errors.

5. DISCUSSION

Ogden *et al.* [16] categorized the knowledge of query writing into knowledge of the data, knowledge of the database structure and knowledge of the query language. The lack of knowledge of the first two categories is best reflected in syntactic errors, particularly those errors due to misspelling or referring to undefined columns or tables. On the other side, lack of knowledge of the query language is better reflected in semantic mistakes. Our findings suggest that the primary reason behind semantic mistakes is students' poor skill in selecting the right technique to design the query needed to answer the test question. The presence of a signal word such as "average" in the question seems to be helpful; however, students' dependency on a such "clue" in the question is not ideal.

Semantic mistakes also have implications for students' development in writing queries. Reisner's [7] model for the development of a query consists of three phases; generation of a template, transformation of English words to database terms, and insertion of database terms into the template. However, we noticed that the majority of our students who abandoned the question due to a semantic mistake had problems with the generation of the initial template. The first step in generating the template is to identify which technique – e.g. natural join, simple subquery, self-join, etc. – is most suitable. Our results suggest that there should be a greater emphasis on this matter when teaching students how to write SQL queries.

Our results show that the majority of semantic mistakes occur in the WHERE clause of the SELECT statement. This error may occur when the capacity of a student's working memory is surpassed [17]. For example, a high number of conditions in the WHERE clause could result in working memory overload, especially when many WHERE conditions are required to join two or more tables. This has been previously shown in the experiment performed by Smelcer [11]. This problem could perhaps be avoided by putting greater emphasis in teaching on following a systematic, step-by-step procedure in segmenting the question and formulating the correct answer in SQL.

6. CONCLUSION

This study attempts to both qualitatively and quantitatively investigate the semantic mistakes made by students. The results of our analysis show that syntactic mistakes are more common than semantic mistakes. However, semantic mistakes are much harder to correct, even among successful students. Our findings show that the majority of semantic mistakes are of type omission, indicating that students have difficulty with selecting the correct type of query, and they also lack a systematic approach to formulating the query.

The majority of omission errors happen in queries that require a JOIN, or a subquery, or a GROUP BY. This has implications for the way that we teach SQL query design, in that we should emphasize techniques that deal with the identification of the type of query necessary to return the requested information, e.g. a JOIN or a subquery. Furthermore, we believe that the selection of the right terminology by the teacher in articulating the question will decrease the chance of students' semantic mistakes that are due to complexity or the comprehension of the question. We are carrying out additional research to confirm this assertion.

7. REFERENCES

[1] Brusilovsky, P., Sosnovsky, S., Lee, D., Yudelson, M., Zadorozhny, V., and Zhou, X. (2008). *An open integrated exploratorium for database courses*. ITiCSE '08. pp. 22-26. http://doi.acm.org/10.1145/1384271.1384280.

[2] Brusilovsky, P., Sosnovsky, S., Yudelson, M. V., Lee, D. H., Zadorozhny, V., and Zhou, X. (2010) *Learning SQL programming with interactive tools: From integration to personalization*. Trans. Comput. Educ. 9, 4, Article 19 (January 2010). http://doi.acm.org/10.1145.1656255.1656257

[3] Mitrovic, A. (1998) *Learning SQL with a computerized tutor*. SIGCSE '98, pp. 307-311. http://doi.acm.org/10.1145/274790.274318

[4] Mitrovic, A. (2003) *An intelligent SQL tutor on the web.* Int. J. Artif. Intell. Ed. 13, 2-4 (April 2003), pp. 173-197.

[5] Prior, J., and Lister, R. (2004) *The Backwash Effect on SQL Skills Grading*. ITiCSE 2004, Leeds, UK. Pp. 32-36. http://doi.acm.org/10.1145/1007996.1008008

[6] Prior, J. (2014) *AsseSQL: an online, browser-based SQL Skills assessment tool*. ITiCSE 2014. Pp. 327-327. http://doi.acm.org/10.1145/2591708/2602682

[7] Reisner, P. (1977) *Use of psychological experimentation as an aid to development of a query language*. IEEE Trans. Softw. Eng. SE-3, 3, 218-229. http://doi.acm.org/10.1145/1103669.1103673

[8] Welty, C., and Stemple, D. W. (1981) *Human factors comparison of a procedural and a nonprocedural query language.* ACM Transactions on Database Systems (TODS) 6.4:626-649. http://doi.acm.org/10.1145/319628.319656

[9] Welty, C. (1985) *Correcting user errors in SQL*. International Journal of Man-Machine Studies 22(4): 463-477.

[10] Buitendijk, R. B. (1988) *Logical errors in database SQL retrieval queries*. computer Science in economics and management 1, 79-96. http://doi.acm.org/10.1007/BF00427157

[11] Smelcer, J. B. (1995) *User error in database query composition*. Int. J. Human-Computer Studies 42, 353-381. http://doi.acm.org/10.1006/ijhc.1995.1017

[12] Brass, S. and Goldberg, C. (2006) *Semantic error in SQL queries: A quite complete list.* The Journal of Systems and Software 79, 630–644. http://doi.acm.org/10.1016/j.jss.2005.06.028

[13] Ahadi, A., Prior, J., Behbood, V., and Lister, R. (2015) *A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries*. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 201-206. ACM, 2015.

[14] Ahadi, A., Behbood, V., Vihavainen, A., Prior, J., & Lister, R. (2016, February). *Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success.* In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 401-406). ACM.

[15] Post, G. V. (2001) *Database management systems: designing and building business applications*. McGraw-Hill.

[16] Ogden, W. D., Korenstein, R., Smelcer, J. B. (1986) *An Intelligent Front-End for SQL*, IBM, San Jose, CA.

[17] Miller, G. A. (1994). *The magical number seven, plus or minus two: Some limits on our capacity for processing information.* Psychological review, *101*(2), 343.

8.3 Discussion

In this Chapter, I reviewed the semantic mistakes the novices make through the construction of SQL *SELECT* statements. My understandings from the data analysis led me to hypothesize that making semantic mistakes is an inevitable matter. Those novices which have a higher abstraction skill complete the construction of the correct SQL statement in less attempts and those who are challenging with this construction require more time/steps/effort to do it. I also hypothesize that given the short capacity of the working memory, it is recommended that the novices construct the SQL statement in a step by step manner, starting with an initial template and adding different bits of the statement to different clauses one at a time. This walks hand in hand with the idea of chunking and how it establishes the development of coding skills of the novices in programming tasks. The fundamental message to be taken along to the next chapter is the solid understanding of what the *attempt* means in this context: a sequence of attempts represent the path through which the database novice walks to complete the construction of the correct SQL statement. Challenged novices take more steps (generate more snapshots) while more advanced students complete the question in less attempts. Hence, the semantically incorrect SQL *SELECT* statement is best interpreted as incomplete, not necessarily incorrect. Understanding the ideal path (what is an ideal path for construction of a SQL statement?) that a less challenging (If the novice is not really challenging much with the code, then is he or she a novice in the first place?) novice would take however, is beyond the scope of this study.

Chapter 9

Students’ Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students’ Success

9.1 Introduction

In the previous chapter, I gave an explanation for the definition of an "attempt" in the context of SQL *SELECT* statements for database novices. In this chapter, I’ll take into account that information used to construct machine learning models which best predict the performance in the course. At this stage, I have awareness of the definition of an "attempt", hence giving the data to the machine learning algorithm will also produce some sense of the data. As I show in the paper, the rule based machine learning, the error code of the syntactic mistakes and the performance on different SQL questions can altogether be strong predictors of other assessments in the course.

9.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;

2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;

3. there are no other authors of the publication according to these criteria;

4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and

5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success

Conference: ACM Technical Symposium on Computing Science Education

URL: http://dl.acm.org/citation.cfm?id=2844640&CFID=862572254&CFTOKEN=60379727

Status: Presented, February 2016

TABLE 9.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 9

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Vahid Behbood				✓
Arto Vihavainen				✓
Julia Prior				✓
Raymond Lister	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi

Name Signature Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister		
Name	Signature	Date

9.2 PDF of the Published Paper

Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success

Alireza Ahadi
University of Technology Sydney
Australia
Alireza.Ahadi@uts.edu.au

Vahid Behbood
University of Technology Sydney
Australia
Vahid.Behbood@uts.edu.au

Arto Vihavainen
University of Helsinki,
Finland
Arto.Vihavainen@cs.helsinki.fi

Julia Prior
University of Technology Sydney
Australia
Julia.Prior@uts.edu.au

Raymond Lister
University of Technology Sydney
Australia
Raymond.Lister@uts.edu.au

ABSTRACT

The computing education community has studied extensively the errors of novice programmers. In contrast, little attention has been given to student's mistake in writing SQL statements. This paper represents the first large scale quantitative analysis of the student's syntactic mistakes in writing different types of SQL queries. Over 160 thousand snapshots of SQL queries were collected from over 2000 students across eight years. We describe the most common types of syntactic errors that students make. We also describe our development of an automatic classifier with an overall accuracy of 0.78 for predicting student performance in writing SQL queries.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages – query languages.

General Terms

Management, Measurement, Human Factors.

Keywords

Online assessment; databases; SQL queries; Machine learning.

1. INTRODUCTION

The Structured Query Language (SQL) is the standard language for relational and object-relational databases. A better understanding of student SQL errors and misconceptions would improve the teaching and learning of SQL. It would also serve the writing of textbooks and other instructional materials. As with any other computer language, SQL queries may contain semantic or syntactic errors. The focus of the previous studies of novices in writing SQL queries has been on the nature of semantic errors. There has been little attention to analyzing the syntactic mistakes made by students when writing SQL queries.

In this paper we use data collected over eight years, from 2300 students, to quantitatively study the syntactic errors committed by students at the authors' institution. We review these mistakes among seven different types of SQL queries and compare syntactic errors of students who were successful versus students who were unsuccessful in writing a correct query. We also show how this information can be used to train a rule-based classifier to predict student success at writing an SQL query.

This paper is structured as follows. In section 2, we review the literature on analysis of semantic errors in SQL. In section 3, we describe how the data was collected and how the classifier was trained. In section 4, we review our findings on syntactic error analysis and demonstrate how the information obtained from student snapshots can be used to predict their performance in writing SQL queries under exam condition. Section 5 expands and discusses our findings. Section 6 presents our conclusions.

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA
© 2016 ACM. ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844640

2. RELATED WORK

Many reports have been published about online SQL tutoring/assessment tools. However, most of those reports focus on the functionality of the tool itself, or on how the system supports a certain pedagogical model [1-6]. Those reports do not analyze the data collected by those systems to determine the syntactic errors that students face when writing SQL queries.

There are some papers in which authors review different semantic errors encountered in writing SQL queries. Reisner [8] categorized queries generated by subjects into three categories: "correct", "minor error(s) only", and "major error(s). Wetly and

Stemple [7] and used a more elaborate categorization scheme: "correct, "minor language error", "minor operand error", "minor substance error", "correctable", "major substance error", "major language error", "incomplete" and "unattempted". Wetly and Stemple studied how subjects faired at writing SQL queries in comparison to a a more procedural query language (TABLET). Later in 1985, Welty [9] ran an experiment on a small number of subjects (N=39) to test how assistance with error correction would affect SQL user performance. In that study, errors are categorized into "minor syntactic", "complex errors", "group by", "semantic error", "incorrect" and "unattempted". Buitendijk [10] categorized SQL errors into the categories of "existence", "comparison", "extension" and "complexity". In that work, a classification of natural language questions was introduced to give insight into possible errors in SQL queries. Smelcer [11] reports seven different type of common mistakes in writing SQL queries, which are "omitting the join clause", "AND/OR difficulties", "omitting quotes", "omitting the FROM clause", "omitting qualifications", "misspellings" and "synonyms". In that work, a model of query writing is developed that integrates a GOMS-type analysis of query writing with the characteristics of human cognition. Brass [12] reports an extensive list of conditions that are strong indications of semantic errors. Although all those works give a fundamental understanding of semantic errors in SQL, none of them reviewed syntactic errors.

3. METHOD

3.1 Data Collection

The data collected in this study forms a total number of ~161000 SQL SELECT statements from ~2300 students. We collected our data in an online assessment system, AsseSQL [5, 6]. The students in this study were all novice undergraduate students enrolled in an introductory database course at the authors' university. Most of the students enrolled in this course were studying for a Bachelor degree in Information Technology or Software Engineering. Each semester there is an online SQL test assessing student's performance in writing SQL SELECT statements. In the online test, students were allowed 50 minutes to attempt seven SQL questions. Each question examines the students' ability to write a SELECT statement which covers a fundamental concept. Table 1 represents these question and the total number of snapshots collected from students' attempts. Each of the seven questions presented to a specific student is chosen at random from a small pool of questions.

Table 1. Number of snapshots generated by students for different SQL SELECT statements.

Type of SELECT statement required in answer	Number of SQL statements collected from students
Group by with having	31484 (20%)
Self-join	27350 (17%)
Group by	24422 (15%)
Natural join	24248 (15%)
Simple subquery	18860 (12%)
Simple, one table	18440 (11%)
Correlated subquery	15856 (10%)

3.2 Error Categorization

In order to collect the execution result of students' attempts, all the students' SQL statements were re-executed in PostgreSQL and the execution results generated by the DBMS were collected. Based on the execution message returned by DBMS, we define a *syntactic error* as an error message which is returned by PostgreSQL engine. In contrast, a *semantic error* is produced by a successfully running query that does not produce the correct answer. For the sake of clarity, we redefine *syntax error* (Error code 42601) as a type of syntactic error which is due to either a typo or the exclusion of a semicolon at the end of the query. Table 2 reviews these error codes and their frequency among students' queries.

Table 2. PostgreSQL error codes and their frequency in student's attempts

PSQL Error code	Description	Frequency
Error 42601	Syntax error	34504 (21%)
Error 42703	Undefined column	20689 (13%)
Error 42803	Grouping error	15442 (10%)
Error 42P01	Undefined table	5548 (3%)
Error 42702	Ambiguous column	4844 (3%)
Error 42883	Undefined function	2534 (2%)
Error 42804	Data type mismatch	1262 (1%)
Error 22008	Date time field overflow	975 (0.61%)
Error 22007	Invalid text representation	849 (0.53%)
Error 22P02	Invalid date time format	536 (0.33%)
Error 3F000	Invalid schema name	457 (0.28%)
Error 42704	Undefined object	134 (0.08%)
Error 42712	Duplicate alias	108 (0.07%)
Error 42P10	Invalid column reference	96 (0.06%)
Error 42P02	Undefined parameter	47 (0.03%)
Error 42725	Ambiguous function	21 (0.01%)

In many cases, an error might arise for multiple reasons. As a result, the error code alone might not be sufficiently self-explanatory to express exactly where the problem lies within the corresponding SQL statement. Therefore, we categorized those error codes which might arise due to different reasons into sub-categories that are summarized in Table 3.

3.3 Classification

In order to investigate the predictive value of the syntactic errors of students' attempts, we trained a classifier to see to what extent this information can be used to distinguish between successful students and unsuccessful students. To that end, information on syntactic errors (i.e. the top eight syntactic errors in Table 2), the number of semantic errors and the total number of attempts of 480 students were used to build the training set. AsseSQL selects questions for each student on a random basis from a question pool. As a result, we decided to train the classifier on the proportion of students who were assigned with the same identical question. This training set included 240 students who were successful in answering a GROUP BY question correctly (Positive set) as well as 240 students who were not able to answer a GROUP BY question (Negative set). We chose to study the GROUP BY question as it had a particularly high ratio of syntactic errors in students' attempts. The classifier trained in this study is PART, a rule based classifier which, in each iteration, builds a partial C4.5 and re-expresses the best leaf as a rule [13]. The PART classifier was selected for two main reasons. First PART handles missing values caused by student queries which no syntactic errors of a given kind. Second, PART provides a clear explanation of the rules generated within the C4.5 iterations. Feature selection and classifier evaluation was performed using the WEKA Data Mining toolkit [14].

Table 3. PostgreSQL error codes and underlying reasons.

PSQL Error code	Reason	Occurrence
42601	Wrong syntax	33482 (97%)
	Chunk of code not closed	681 (2%)
	Invalid subquery syntax	268 (<1%)
	Other rare syntactic errors	73 (<1%)
42803	Aggregate function may not be used in GROUP BY clause	13019 (84%)
	Aggregate function may not be used in WHERE clause	2294 (14%)
	Aggregate function may not be nested	104 (2%)
42883	Operator does not exist	1628 (64%)
	Function does not exist	906 (36%)
42804	Argument of AND must be type Boolean	547 (44%)
	Argument of OR must be type Boolean	437 (35%)
	Argument of HAVING must be type Boolean	271 (21%)

4. RESULTS

4.1 Syntactic Errors Have a High Frequency Ratio Among Students' Attempts

Among ~161000 students' attempts, more than half of those attempts result in a syntactic error (54%). Around 40% of all executions do not include a syntactic error, but include semantic errors. Thus only the remaining 6% of executions result in a successful execution capable of producing the correct result table. A low percentage of successful executions is to be expected, as a student stops attempting a question in the online system as soon as they have generated a correct answer. However, this low percentage of correct answers does reflect: (1) the long sequence of incorrect queries, both syntactic and semantic, typically generated by students before arriving at the correct answer, and (2) the many students who never generate a correct answer to some questions.

Table 4 reviews the frequency of incorrect SQL statements written by students. As can be seen, the number of syntactic errors in most categories is more than the number of semantic errors. Examining the last attempt of unsuccessful students revealed that half (51%) of unsuccessful students abandoned the question when they were not able to fix a syntactic error. As shown in Table 2, error 42601 ("syntax error") is the biggest category of syntactic errors. The main cause of this error is typos. The second largest category of encountered errors is error 42703 which is generated when the referenced column does not exist (which may also be caused by a typo).

Table 4. Incorrect SQL statements and syntactic errors. Second column represents the percentage of incorrect queries among all queries collected for each query type. Third column represents the percentage of incorrect queries among all incorrect queries per query type where the incorrectness is due to syntactic errors.

SELECT statement type	Percentage of unsuccessful statements	Unsuccessful due to syntactic errors
Simple, one table	89%	54%
Group by	93%	68%
Group by with having	96%	61%
Natural join	94%	66.%
Simple subquery	92%	64%
Self-join	98%	38%
Correlated subquery	93%	55%

4.2 Syntactic errors in different types of SQL SELECT statements

To better understand which syntactic errors students encounter in writing different types of SQL statements, we investigated the error codes generated by their attempts in answering seven different types of SQL questions. We chose these seven types of queries because the relative difficulty of these seven types has been studied and established by Ahadi *et. al*. [15]. As can be seen in Table 5, a limited number of error codes form the majority of the total population of most encountered errors in the seven different query types. However, the frequency of these errors varies between the seven query types. For each query type, with the exception of the self-join, more than half of the students who did not get a right answer gave up when they were not able to fix the syntactic error produced by their SELECT statement.

4.3 Successful vs. Unsuccessful: Syntactic Error Comparison

To characterize students as "successful" or "unsuccessful" according to their syntactic errors, for each SQL query type we selected an equal number of students (N >300) and compared the total number of semantic and syntactic errors (Table 6). As an example of how this table was constructed, consider the 81% figure shown in the top left of Table 6. For error code 42601, from the total number of queries generated by successful and unsuccessful students, 81% are from unsuccessful students.

According to the information presented in Table 6, unsuccessful students tend to have more syntactic and semantic errors compared to successful students. Although the number of encountered syntactic errors differs from query type to another, the majority of syntactic and semantic errors in each error category are from unsuccessful students.

Table 5. Most encountered errors in different SQL statements.

SQL type	1st error	2nd error	3rd error	All Other Errors
Simple, one table	42601 (46%)	42703 (19%)	42803 (13%)	22%
Group by	42601 (52%)	42803 (26%)	42703 (11%)	11%
Group by with having	42601 (45%)	42803 (29%)	42703 (15%)	11%
Natural join	42703 (38%)	42601 (27%)	42P01 (12%)	23%
Simple subquery	42703 (33%)	42601 (28%)	42803 (13%)	26%
Self-join	42601 (37%)	42703 (23%)	42P01 (11%)	29%
Correlated subquery	42601 (35%)	42703 (32%)	42P01 (11%)	22%

4.4 Syntax Error Based Prediction of Unsuccessful Students

As we were able to characterize successful and unsuccessful students according to their syntactic errors, we decided to see to what extent a student's degree of struggle would be a good predictor of student's success in eventually writing the correct query. We therefore trained the PART classifier on an equal number (240) of successful and unsuccessful students in answering GROUP BY questions (i.e. N=480). On a 10-fold cross validation training mode, the classifier was able to correctly classify 77% of students correctly. Table 7 provides further details on the performance of the classifier.

To reduce the number of overlapping features, reduce over fitting, and to improve predictive accuracy of the feature set, feature selection was performed. We used correlation-based feature subset selection, where the individual predictive ability of each feature along with the degree of redundancy between the features was evaluated using three methods; (1) genetic search, (2) best first method and (3) greedy stepwise method. After feature selection, four features were used to train this classifier. Those features were: (1) how many times a student's queries contained error 42803, (2) contained error 42601, (3) a semantic error, and (4) total number of attempts student at the question.

Using those four features, five rules were generated by PART. Inspection of the rules identified features distinguishing successful and unsuccessful students. For example, two of the rules generated by our classifier are:

- If the total number of attempts is higher than 30 and there is at least one syntactic error of code 42803, then the student is not successful.
- If the student has not encountered a semantic error in answering the question, then the student is not successful. (A complete absence of semantic errors usually indicates that all the student's attempts contained syntactic errors.)

We discuss these rules, and other findings, in the next section.

5. DISCUSSION

According to our analysis, the majority of students' mistakes in writing SQL SELECT statements are either syntax errors or an incorrect referred column in different clauses of the SELECT statement. While the former seems to be a result of lack of practice, the second category gives us some insights into student's understanding. The most common mistake among students' attempts in writing a simple join query is an incorrect column name in either the SELECT or WHERE clause. In some cases, this is simply due to a typing error. In cases where the incorrect column name is not a typing error, perhaps the complexity of the provided entity relationship diagram leads students to make a mistake in choosing the right field name.

Table 6. Distribution of errors for unsuccessful students compared to successful students in different query types. The values in the table represents the proportion of unsuccessful students from the total mistake for each error and cross different query types.

PSQL Error code	Simple, one table	Group by	Group by with having	Natural join	Simple subquery	Self-join	Correlated subquery
Error 42601	81%	70%	71%	73%	70%	73%	72%
Error 42702	63%	90%	75%	70%	69%	46%	75%
Error 42703	73%	81%	68%	70%	61%	58%	72%
Error 42803	84%	69%	71%	70%	72%	73%	61%
Error 42804	89%	76%	76%	85%	77%	76%	50%
Error 42883	85%	85%	71%	74%	88%	60%	69%
Error 42P01	77%	86%	78%	65%	74%	64%	78%
semantic error	75%	61%	61%	61%	53%	50%	59%

Table 7. PART classifier's performance in classifying successful and unsuccessful students in writing a GROUP BY query.

TP Ratio	FP Ratio	Precision	Recall	F-Measure	ROC Area	Class
0.84	0.30	0.74	0.84	0.79	0.84	Unsuccessful
0.70	0.16	0.81	0.70	0.75	0.84	Successful
0.77	0.23	0.77	0.77	0.77	0.84	Weighted Avg.

We were surprised to find that error 42803 is the third most common mistake in writing a simple SELECT statement on one table, accounting for 13% of all syntax errors. When this error occurred, it almost always occurred in the WHERE clause. A simple SELECT doesn't require a GROUP BY clause, which probably indicates a serious misconception.

Inspecting the top three most encountered syntactic mistakes among seven different categories of SQL queries, error 42P01 (i.e. undefined table) is observed in four different types of queries:

a) When there is a need to perform a natural join between two relations.

b) When writing a simple subquery

c) When there is a need to join a table to itself.

d) When writing a correlated subquery.

What these four categories of queries share is the presence of either more than one table in the SELECT statement or the presence of more than one SELECT clause. This might be an indicator of students' lack of skill in identifying the correct tables from which to extract the desired information. On the other hand, investigating a small number of their queries shows that the main mistake they make is mistyping the correct name of the table, even though the number of tables used in each entity relationship diagram is small.

The results from different machine learning algorithms with the same goal can be extremely context-dependent. Even with the same algorithm, variation can even occur with variations in the exact training data used. We trained a wide range of models to classify student's performance in the online test and for each classifier, we obtained a prediction accuracies ranging from 60% to 79%. Each of these classifiers has a different set of generated rules and selected features

6. CONCLUSION

Our study goes a small way to addressing that imbalance in the computing education literature between the study of novices writing programs and novices writing SQL queries. Our data is from a relatively large data set (over 161000 SQL queries collected from ~2300 students) for a range of different syntactic errors that novices make in writing SQL statements. Our findings show that, in general, students make more syntactic errors than semantic errors. Furthermore, a syntactic error and not a semantic error is what in most cases causes a student to abandon answering a question. Syntax errors in different clauses of the SELECT statement, undefined referred column errors, and grouping errors are the most common mistakes that novices make in writing their SQL statements. The prevalence of syntactic errors, and in particular the fact that syntactic errors is what leads students to abandon attempting a question, indicates that the teaching of SQL queries needs to place greater emphasis on syntax and syntax errors. While we believe that semantic errors are the more intellectually demanding errors, students will not come to grips with semantic errors while their query formulation remains dominated by syntax errors.

7. REFERENCES

[1] Brusilovsky, P., Sosnovsky, S., Lee, D., Yudelson, M., Zadorozhny, V., and Zhou, X. 2008. An open integrated exploratorium for database courses. *ITiCSE* '08. pp. 22-26.

[2] Brusilovsky, P., Sosnovsky, S., Yudelson, M. V., Lee, D. H., Zadorozhny, V., and Zhou, X. 2010. Learning SQL programming with interactive tools: From integration to personalization. Trans. Comput. Educ. 9, 4, Article 19 (January 2010).

[3] Mitrovic, A. 1998. Learning SQL with a computerized tutor. SIGCSE '98, pp. 307-311.

[4] Mitrovic, A. 2003. An intelligent SQL tutor on the web. Int. J. Artif. Intell. Ed. 13, 2-4 (April 2003), pp. 173-197.

[5] Prior, J., and Lister, R. 2004. The backwash effect on SQL skills grading. ITiCSE 2004, Leeds, UK. pp. 32-36.

[6] Prior, J. 2014. AsseSQL: an online, browser-based SQL skills assessment tool. ITiCSE 2014. pp. 327-327.

[7] Wetly, C. and Stemple, D. 1981. Human factors comparison of a procedural and a nonprocedural query language. ACM Transactions on Database Systems (TODS) 6;4:626-629

[8] Reisner, P. 1977. Use of psychological experimentation as an aid to development of a query language. IEEE Trans. Softw. Eng. SE-3, 3 (1977), 218-229.

[9] Wetly, C. 1985. Correcting user errors in SQL. International Journal of Man-Machine Studies 22(4): 463-477.

[10] Buitendijk, R. B., 1988. Logical errors in database SQL retrieval queries. Computer Science in Economics and Management 1 (1988) 79-96

[11] Smelcer, J. B. 1995. User error in database query composition. Int. J. Human-Computer Studies (1995) 42, 353-381

[12] Brass, S. and Goldberg, C. 2006. Semantic error in SQL queries: A quite complete list. The Journal of Systems and Software 79 (2006) 630–644.

[13] Frank, E. and Witten, I. H. 1998. Generating accurate rule sets without global optimization. (Working paper 98/2). Hamilton, New Zealand: University of Waikato, Department of Computer Science.

[14] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten. I. H. 2009. The WEKA data mining software:

an update. ACM SIGKDD explorations newsletter, 11(1):10-18, 2009.

[15] Ahadi, A., Prior, J., Behbood, V. and Lister, R. 2015. A quantitative study of the difficulty for novices of writing seven different types of SQL queries. ITiCSE (2015) Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education 201-206.

9.3 Discussion

The result of this chapter and the result of Chapter 6 show that the "attempt" in both contexts can be interpreted as the degree to which a novice struggles with the given task. In both contexts, the more struggling novices generate a greater number of snapshots, spending more time on the code, and showing poorer performance in the course outcome. In both contexts, I used the "number of attempts" and "correctness ratio" to construct the prediction models. To investigate the applicability of the analysis of the proposed features extracted from the source code snapshot data, I set to explore the consistency of my findings in the next chapter by examining the data collected during different stages of the semester.

Chapter 10

Performance and Consistency in Learning to Program

10.1 Introduction

In this Chapter, I investigate the application of the machine learning tools, on the data collected during different stages of the semester. I show that correlation between performance and the number of attempts with the course outcome is different when the analyzed data is collected from different time periods of the same semester within the same institute.

10.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;
2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;
3. there are no other authors of the publication according to these criteria;
4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and
5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: Performance and Consistency in Learning to Program

Conference: Australian Computing Education

URL:

Status: Presented, January 2017

TABLE 10.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 10

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Shahil Lal				✓
Juho Leinonen				✓
Raymond Lister	✓			
Arto Hellas	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi

Name	Signature	Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister

Name	Signature	Date

10.2 PDF of the Published Paper

Performance and Consistency in Learning to Program

Alireza Ahadi and
Raymond Lister
University of Technology,
Sydney
Australia
alireza.ahadi@uts.edu.au
raymond.lister@uts.edu.au

Shahil Lal
University of Sydney, Australia
shahil.lal7@gmail.com

Juho Leinonen and
Arto Hellas
Department of Computer
Science
University of Helsinki
Finland
juho.leinonen@helsinki.fi
arto.hellas@cs.helsinki.fi

ABSTRACT
Performance and consistency play a large role in learning. Decreasing the effort that one invests into course work may have short-term benefits such as reduced stress. However, as courses progress, neglected work accumulates and may cause challenges with learning the course content at hand.

In this work, we analyze students' performance and consistency with programming assignments in an introductory programming course. We study how performance, when measured through progress in course assignments, evolves throughout the course, study weekly fluctuations in students' work consistency, and contrast this with students' performance in the course final exam.

Our results indicate that whilst fluctuations in students' weekly performance do not distinguish poor performing students from well performing students with a high accuracy, more accurate results can be achieved when focusing on the performance of students on individual assignments which could be used for identifying struggling students who are at risk of dropping out of their studies.

Keywords
source code snapshot analysis; educational data mining; CS1

1. INTRODUCTION
Researchers have sought to determine whether factors such as gender, age, high-school performance, ability to reason, and the performance in various aptitude tests correlate with the ability to create computer programs [6]. As many of these factors are static and only rarely account for *what the students do* while programming, studies that analyze the learning process have started to emerge [11].

In these studies, researchers study features extracted from programming process recordings at various granularity [1, 5, 12, 24]. Process data has been collected also in other contexts, e.g. during in-class peer instruction [15]. With such research, in the future, *data analytics* and support tools can be regularly applied to provide instructors with greater insight into what is actually occurring in the classroom, opening up new opportunities for identifying individual student needs, providing targeted activities to students at the ends of the learner spectrum, and personalizing the learning process [11].

In our work, we are interested in how students' performance evolves during the course and how their performance and consistency contribute to the course outcomes. More specifically, we study how students' performance, measured through correctness of snapshots taken from students' programming process during an introductory programming course, evolves over time and analyze whether students consistently perform on the same level. The analysis is based on observing students' average performance, and seeks to determine whether the weekly performance fluctuates due to some unobserved variables. Furthermore, we also study students' performance using non-parametric and parametric clustering approaches with the goal of detecting those students who could benefit from a teaching intervention.

This article is organized as follows. In the next Section, we discuss related work. In Section 3, we describe the research questions and data, and in Section 4, we outline the methodology and results. The results are further discussed in Section 5 and drawn together in Section 6.

2. RELATED WORK
2.1 Predictors for Success
Traditional predictors for students' success include past academic performance [2], previous exposure to programming [7], and demographic factors such as age and gender [25]. In addition, variables such as students' expectations for course and for their grade correlates with the final course outcomes [16]. However, as pointed out by Watson et al. [23], many of the traditional predictors are sensitive to the teaching context, and generalize relatively poorly.

There are studies that indicate that there is no significant correlation between gender and programming course outcomes [3, 21, 25], as well as studies that indicate that the gender may explain some of the introductory programming course outcomes [2]. Similarly, when considering past mathematics performance, some studies indicate no significant correlation between programming course outcomes and mathematics outcomes [23, 25], while others have found referential correlations [19].

Even the connection between introductory programming course outcomes and past exposure to programming is controversial. Whilst a number of studies have reported that past programming experience helps when learning to program [4, 7, 26], contradictory results also exist. For example,

ACE '17, January 31-February 03, 2017, Geelong, VIC, Australia
© 2017 ACM. ISBN 978-1-4503-4823-2/17/01...$15.00
DOI: http://dx.doi.org/10.1145/3013499.3013503

Bergin and Reilly found that students with no previous programming experience had a marginally higher mean overall score in introductory programming [2]. Watson et al. [23] also found that while students with past programming experience had significantly higher overall course points than those with no previous programming experience, programming experience in years had a weak but statistically insignificant negative correlation with course points.

A more recent approach to identifying students who may or may not succeed is related to the use of data that is recorded from the environment in which the students are learning to program. We focus on such studies next.

2.2 Work Patterns and Effort

In-class behavior has been explored by Porter et al. [15] who observed that the proportion of correct clicker answers early in a course was strongly correlated with final course outcomes. Similarly, interaction with web-based programming environments has been studied [18]; Spacco et al. noticed statistically significant but weak relationships between the final exam score and students' effort.

Sequential captures of programming states have been used to analyze students' working patterns [11]. Jadud found a high correlation with the course exam score and the students' ability to create and fix errors [12], and observed that the less errors a student makes, and the better she solves them, the higher her grade tends to be [12].

The approach proposed by Jadud has been extended by Watson et al. [24] to include the amount of time that students spend on programming assignments [24]. More recently, a similar approach that analyzes subsequent source code snapshots was also proposed by Carter et al. [5], who analyzed the errors that students encounter in more detail.

Programming states have been used to elicit finer information from the programming process. For example, Vihavainen et al. [22] analyzed the approaches that novices take when building their very first computer programs. They observed four distinctive patterns, which were related to typing the program in a linear fashion, copy-pasting code from elsewhere, using auto-complete features from the programming environment, and using programming environment shortcuts. Similar work was conducted by Heinonen et al. [8], who analyzed problem solving patterns of two populations in an introductory programming course; those who failed and those who passed. In the study, Heinonen et al. noticed that a number of the students who failed had a *programming by incident* -approach, where they sought a solution through a seemingly random process – a behavior that has also observed in the past (see e.g. [17]).

One stream of research merges snapshots into states, and analyzes students' progress through those states. Piech et al. [14] clustered students' approaches to solving a few programming tasks, and found that the solution patterns are indicative of course midterm scores. Similarly, Hosseini et al. [9] analyzed how students' progress towards correct solutions in a course, and noticed that some students were more inclined to build their code step by step from scratch, while others started from larger quantities of code, and progressed towards a solution through reducing and altering the code.

Students have also been grouped based on features describing how they work on programming assignments (lines changed, added, removed, and so on), followed by an evaluation of how these features change over a number of assignments [27]. Overall, students in different groups may differ in the way how they benefit from help [27].

Our research is closely related to the work by Worsley et al. [27] and Hosseini et al. [9]. However, whilst both have focused more on change sizes and other similar metrics, our focus is on changes in correctness of code. We also evaluate new methodologies and visualizations for the task at hand, and study the students' performance across a course instead of over a smaller set of assignments.

3. RESEARCH DESIGN

3.1 Research Questions

Our research questions are as follows:

- **RQ1: How does students' performance evolve during a programming course?**
- **RQ2: How does the performance change across course weeks? That is, how consistently do students perform during the course?**
- **RQ3: To what extent can students' consistency be used to predict students' course outcomes?**

With research question one, we aim to both validate previous results by Spacco et al. [18] in a new context, and to extend the work by analyzing students' performance changes from week to week. These changes are discussed in the light of the content taught in the course. With research question number two, we seek to determine whether working consistency – i.e. does student perform similarly over the course – affects course outcomes, and with the third research question, we revisit multiple works where authors have sought to determine those at risk of failing the introductory programming course.

3.2 Context and Data

The data for the study comes from a six-week Java introductory programming course organized at University of Helsinki during Spring 2014. One of the authors of this article is responsible for organizing the course under study. In the course, half of the grade comes from completing programming assignments, and the rest of the grade comes from a written exam, where students are expected to answer both essay questions as well as programming questions. The course has a set limit for the pass rate: at least half of the overall points as well as half of the exam points need to be attained to pass, and the highest grade is attained with over 90% of course points.

The course was based on a blended online textbook, had a single two-hour lecture, and tens of hours of weekly support in open labs. The course provides students a view to both procedural and object-oriented programming. Course assignments start easy and small, helping students to first focus on individual constructs before they are combined with others. After the students have become familiar with the basic tools and commands, variables and conditional statements are introduced, followed by looping. Students start constructing their own methods during the second week in the course, and start working with lists during the week three. Principles of object-oriented programming are introduced during the latter parts of the third week of the course, and students start building their own objects during the fourth week. Overall, during the course, the students slowly proceed towards more complex assignments and topics.

For the purposes of this study, students were asked to install a software component that records source code snapshots with metadata on assignment-specific correctness. Overall, 89 students agreed to provide their data for the purposes of our study.

3.3 Measuring Performance and Consistency

Throughout this work, students' performance and consistency is measured through their progress and work on the course assignments. For each assignment in the course, there are automatic unit tests, which are used to both support the students as they work on the assignment, and to provide information on the degree of correctness on the assignment. As the students work on the assignments, data is gathered for analysis. Finally, at the end of the course, the students take an exam. In this work, when seeking to predict course outcomes, we focus on predicting the outcomes of the exam.

Students' *average performance* in an assignment means the average correctness of snapshots that have been recorded when the students have either saved, run, tested, or submitted the code that they are working on. The average is calculated for each assignment for all students who worked on the assignment. Average performance provides a measure of the degree to which students' struggled on an assignment and indicates overall performance throughout the course.

Students' *consistency* is measured through whether they perform on the same level throughout the course. More specifically, for each week, we place students' into weekly performance quantiles and measure whether they perform on the same level (i.e. stay in the same quantile), or whether their performance level fluctuates (i.e. changes over the weeks). The consistency is used both as a way to measure the weekly struggle as well as the effort that the student invests into the assignment at a specific state; if a student is always in the upper quantile, most of the effort is invested in complete or nearly complete assignments.

4. METHODOLOGY AND RESULTS

4.1 Students' Performance over the Course

First, we performed statistical analysis on the assignment correctness data gathered from the students' programming process. Average performance (Figure 1) was in focus to analyze how students' performance evolves during the programming course.

A first order (i.e. linear) regression analysis of students' performance throughout the course shows that average performance declines throughout the course. Second order regression shows that the average performance of students increases until the middle of the course, and then declines towards the end.

Overall, students performance decreases throughout the course. This observation is partially explainable by the incremental nature of programming. Students proceed towards more complex assignments and topics and new content is added continuously. At the same time, the results show that the peak average performance is reached at the middle of the course when students start to work on object-oriented programming.

4.2 Students' Performance over Course Weeks

After analyzing the overall trend, we study whether students perform consistently throughout the course, or whether their performance varies. This analysis was performed using weekly average performance quantiles.

In Figure 2, students are placed into four categories based on their average performance throughout each week. If the average performance of a student during the whole week was

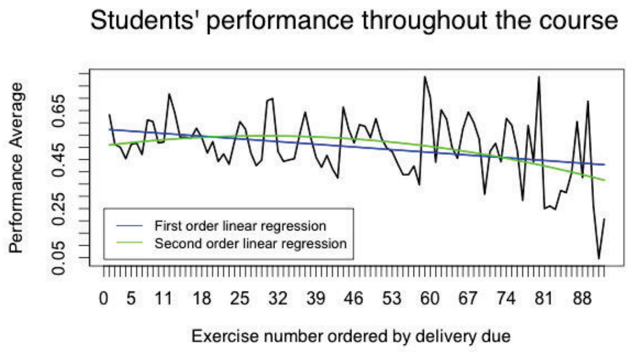


Figure 1: First order and second order regression plot of the mean of performance per exercise.

over 75%, she was in the high-performing quantile (green in Fig. 2), whilst if the student's average performance was less than or equal to 25%, she was in the low-performing quantile (red in Fig. 2). As the semester progresses, there is a noticeable decline in the number of students who belong to the high-performing quantile. At the end of the course, nearly 50% of the students are in the lowest performing quantile, and only a very small number of students remained in the high-performing quantile.

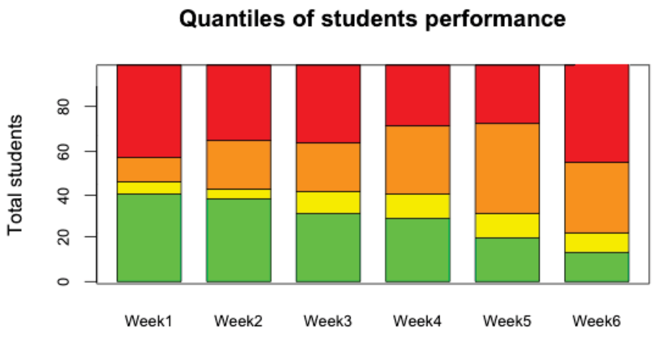


Figure 2: Frequency of students in different quantiles per week in the course. The colors green, yellow, orange and red represent 1st, 2nd, 3rd and 4th quantiles respectively.

We then set out to analyze the extent to which a student's performance during one week indicates performance during the next week. To visualize this, a state transition diagram for the student's performance transition from one week to the next week was constructed. These transitions are calculated using the previous weekly performance quantiles, where a student's average performance in all weekly assignments could fall in one of the four quantiles.

When analyzing the transitions (Figure 3), the majority of students show neither progression or retrogression. That is, a large part of the students perform somewhat consistently between any two weeks of the course. At the same time, early in the course, majority of the students fell either into the upper quantile or the lower quantile, whilst at the end of the course, majority of the participants had a low performance average.

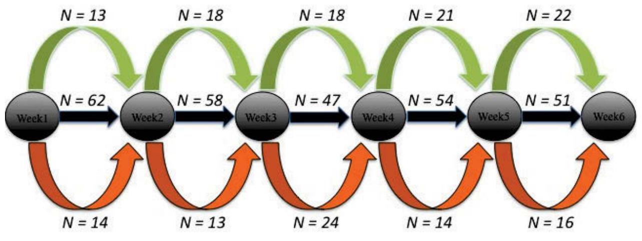


Figure 3: State transition diagram of students' performance throughout the course.

4.3 Consistency and Course Outcomes

Using students' consistency to predict course outcomes is performed in two parts. We first use the consistency over the weeks for the course outcome prediction, and then, in the next subsection, we delve into assignment-specific performance and course outcomes.

The analysis was performed using t-Distributed Stochastic Neighbor Embedding (t-SNE) clustering [20], which is a technique for dimensionality reduction suitable for high-dimension datasets. For the analysis, three separate datasets were used. The first dataset contained students' average performance per week, the second dataset contained students' weekly performance quantiles, and the third contained students' state transitions between performance quantiles.

For each dataset, an analysis of a limited range of weeks was performed to evaluate whether the clustering method could be used as an early indicator of performance, as well as to identify weeks where students' performance varied the most. Kullback-Leibler divergence was used as a measure for the difference between the clustering outcome and the course outcome for each dataset. Values of the analysis are summarized in Table 1. Here, the consistency dataset (dataset 3) has entries only in the rows with more data as transitions between any two weeks did not yield meaningful results.

Table 1: t-SNE Kullback-Leibler divergences of three different datasets.

period	dataset 1	dataset 2	dataset 3
Weeks one and two	0.25	0.36	NA
Weeks two and three	0.27	0.36	NA
Weeks three and four	0.2	0.25	NA
Weeks four and five	0.23	0.29	NA
Weeks five and six	0.18	0.28	NA
Weeks one to three	0.25	0.33	0.38
Weeks two to four	0.24	0.35	0.35
Weeks three to five	0.26	0.24	0.34
Weeks four to six	0.28	0.25	0.35

The Kullback-Leibler divergence of the students' state transition data (dataset 3) is higher when compared to both weekly performance (dataset 1) and weekly quantiles (dataset 2). That is, the average weekly performance performs better as an indicator of final exam result than the transitions between the performance quantiles.

The result of t-SNE clustering on students' average performance per week is shown in Figure 4 – here, we have used the data from weeks four, five and six with Kullback-Leibler divergence 0.28. T-SNE does not require an initial value for the number of desired clusters, that is, it is a non-parametric approach. One of the major benefits of non-parametric models is that the parameters are determined by the data, not the chosen model.

As could be seen in Figure 4, t-SNE was able to cluster students based on their performance into meaningful clusters – few indicating mostly passing students, one with mostly dropouts, and one with mixed data. The labels in the figure depict those students who pass the subject (`Y`), those who failed the subject (`N`), and those who drop out from the course (`Z`).

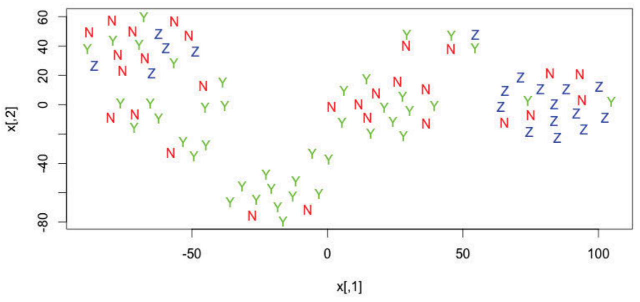


Figure 4: t-SNE based visualization of students final course outcomes using data from weeks four, five and six of students' average performance.

4.4 Assignment-specific Performance and Course Outcomes

Thus far, students were clustered based on their weekly performance. We observed that performance fluctuations between weeks is not as indicative of performance in the final exam as the average weekly performance. We now move from analyzing weekly performance to analyzing assignment-specific performance and its relationship with course outcomes. As the number of assignments is high, dimensionality reduction over the assignments using principal component analysis (PCA) [10] was performed.

Whilst performing PCA on the assignment data, we also performed k-means clustering [13] to identify a meaningful amount of groups into which the data could be split into. Upon analysis of the k-means clustering results over k between 1 and 10, we observe that the projection of our data could be best explained in a two-dimensional space with two principal components. We then used those two principal components to divide the students into two groups. As a result, 78% of students who failed or dropped the subject are placed into the correct cluster.

The same analysis was then performed on three datasets similar to the previously used datasets: (a) the quartile of each student's performance on each exercise; (b) the state transition data based on the score of each exercise where the student's performance from one week to the next week could be either progress, retrogress or no change; and (c) the state transition data based on the quartile of the obtained score of the student per exercise where the student's performance from one week to the week after could be either progress, retrogress or no change.

When performing clustering on the three datasets, we observed that the used k-means clustering does not show a good separation of successful and unsuccessful students. The result of our analysis suggests that at-risk students could best be predicted by the scores or quantile of exercises (i.e. datasets a and b, being able to correctly identify 78.72% and 76.60% of the at-risk students respectively), compared to the quartile state of student's performance or performance transition between exercises. However, the predictive information stored in state transitions seem to be more effective in identifying students who perform well in the final exam of the course (identifying 66.67% of well-performing students correctly).

5. DISCUSSION

5.1 Overall performance in the course

Overall, in line with previous studies, we observed that students' performance decreases during the course. This is in line with both the general knowledge in introductory programming as well as the literature where students' performance in the online programming environment has been analyzed [18]. However, when contrasting our results to those in Spacco et al. [18], students in our context are working in a traditional programming environment, and the majority of the course assignments expect that the students implement more than a single function – thus, generalizing the previous results to a new context.

Upon analysis of the average performance, in Fig. 1, we observed through second order regression that the performance first increases, and then decreases. We observe that students in the course perform reasonably well until the introduction of object-oriented programming, after which the assignments become more complex.

Overall, course topics build on each other, and learning each new topic depends on whether the student understands the previous topic to a sufficient detail. If a student struggles on a topic in the course and fails to grasp it, he or she will likely struggle with upcoming topics as well. Second, as programming courses progress, it is typical that the complexity of the programming assignments increases.

5.2 Changes in performance

When analyzing the students performance transitions in the course, we observe that for the majority of the students, the performance stays nearly the same. When analyzing performance through quantiles, nearly none of the students perform at a level where their solutions would, on average be over 75% correct. What we likely also observe here is that as the course progresses, students become less likely to tinker with their solutions after they have finished the solution, leading to a smaller amount of snapshots in states with high correctness.

At the same time, as the assignments are more complex, starting an assignment becomes more challenging, and approaching a correct solution takes more steps than before. This means that as students struggle, they perform worse in the light of our metric. At the same time, struggling does not necessarily indicate poor *learning*, only that the assignments are more challenging.

We observe that fluctuations in average performance does not transfer well to fluctuations in course performance. Our results are in line with those in [27], indicating that students' performance does not drastically change during the semester.

5.3 Clustering and at-risk students

When comparing the clustering results to the results previously presented in the literature, the outcomes are mediocre, and we do not observe a good distinction between the students who perform well and those who do not. Recent studies that have focused on selecting features that best determine students who are at risk of dropping out have reported high correlations. For example, a recent result from [1] reported $MCC = 0.78$, when using only the first week of data in a course and seeking to predict whether students fall under or over the class median, a significantly better result than that from our study. At the same time, the data, features and methodology was different to ours.

One of the reasons for the lack of performance in the clustering is that distinguishing between those students who pass and those who fail is not trivial. There is a clear separation between clusters of students who are far from the fail/pass -border, while the course outcome for a student who receives 49/100 (i.e. a fail in our system) and a student who receives 50/100 (i.e. a pass in our system) is large in terms of grading, there is very little difference in how they fared in the course and in the exam.

5.4 Limitations of the study

Our study has several limitations. First, the sample size in our dataset was only 89, which is relatively small for clustering and PCA. Second, the exam results come from a pen-and-paper exam, which does not fully reflect the activities that students perform during the course. It is possible that students handle programming tasks well, but are, for example, not able to properly explain what they are doing using pen and paper. On the other hand, students may get lower course marks because they are not able to complete the more complex assignments, but that does not necessarily mean that a student did not learn the topic and would not be able to do well in an exam.

Third, when analyzing performance transitions, we limited our work to quartiles. It is possible that better results would have been achieved with different settings such as a different interval for the quartiles.

Finally, we do not know if students do their best on the assignments as it is possible that some students only seek to pass the course. Additional details should be incorporated to the clustering to take such students into account.

6. CONCLUSIONS

In this work we analyzed students performance and consistency in an introductory programming course. For the analysis, we used sequential source code snapshots that described students progress with the programming assignments. We investigated course-level changes in students' performance, analyzed whether the performance varied over different weeks, and evaluated clustering to identify students at risk.

Overall, to answer research question one, *"How does students' performance evolve during a programming course?"*, we observe that students' performance declines throughout the course. This result is in line with previous studies (see e.g. [18]), and is explainable through the incremental nature of programming. To proceed in a course, the student has to learn the topics from the previous week.

Whilst the overall performance declined over the course, at the same time, we observe that the majority of the students have a constant performance from week to week. That is, to answer the research question two, *"How does the performance change across course weeks?"*, the performance mostly stays the same. This means that the majority of the students perform similarly throughout the course, and that the overall decline in the course performance is likely explainable by the increasing complexity of the course assignments – a topic for future study.

When evaluating this performance as a predictor of final course outcomes, we noticed that fluctuation in course performance across weeks is a relatively poor metric for final course performance. The most accurate clustering was achieved using average performance. To answer the research question three, *"To what extent can students' performance be used to identify those that may drop out from a course?"*, we posit that one could use the clusters built from average performance. However, at the same time, one should seek to verify that there is a sufficient number of study samples, and that the measured outcome depicts the students' working process.

7. REFERENCES

[1] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 121–130, New York, NY, USA, 2015. ACM.

[2] S. Bergin and R. Reilly. Programming: factors that influence success. *ACM SIGCSE Bulletin*, 37(1):411–415, 2005.

[3] P. Byrne and G. Lyons. The effect of student attributes on success in programming. In *ACM SIGCSE Bulletin*, volume 33, pages 49–52. ACM, 2001.

[4] B. Cantwell Wilson and S. Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. In *ACM SIGCSE Bulletin*, volume 33, pages 184–188. ACM, 2001.

[5] A. S. Carter, C. D. Hundhausen, and O. Adesope. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 141–150, New York, NY, USA, 2015. ACM.

[6] G. E. Evans and M. G. Simkin. What best predicts computer proficiency? *Comm. of the ACM*, 32(11):1322–1327, 1989.

[7] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? *ACM SIGCSE Bulletin*, 32(3):25–28, 2000.

[8] K. Heinonen, K. Hirvikoski, M. Luukkainen, and A. Vihavainen. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 229–234, New York, NY, USA, 2014. ACM.

[9] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a Java programming course. In *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group*, 2014.

[10] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[11] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ITICSE-WGR '15, pages 41–63, New York, NY, USA, 2015. ACM.

[12] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84, New York, NY, USA, 2006. ACM.

[13] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[14] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[15] L. Porter, D. Zingaro, and R. Lister. Predicting student success using fine grain clicker data. In *Proceedings of the tenth annual conference on International computing education research*, pages 51–58. ACM, 2014.

[16] N. Rountree, J. Rountree, and A. Robins. Predictors of success and failure in a cs1 course. *ACM SIGCSE Bulletin*, 34(4):121–124, 2002.

[17] J. Spacco. *Marmoset: a programming project assignment framework to improve the feedback cycle for students, faculty and researchers.* PhD thesis, 2006.

[18] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall. Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 18–23, New York, NY, USA, 2015. ACM.

[19] M. V. Stein. Mathematical preparation as a basis for success in CS-II. *Journal of Computing Sciences in Colleges*, 17(4):28–38, 2002.

[20] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[21] P. R. Ventura Jr. Identifying predictors of success for an objects-first CS1. 2005.

[22] A. Vihavainen, J. Helminen, and P. Ihantola. How novices tackle their first lines of code in an ide: analysis of programming session traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 109–116. ACM, 2014.

[23] C. Watson, F. W. Li, and J. L. Godwin. No tests required: Comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 469–474, New York, NY, USA, 2014. ACM.

[24] C. Watson, F. W. B. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 319–323, Washington, DC, USA, 2013. IEEE Computer Society.

[25] L. H. Werth. *Predicting student performance in a beginning computer science class*, volume 18. ACM, 1986.

[26] S. Wiedenbeck, D. Labelle, and V. N. Kain. Factors affecting course outcomes in introductory programming. In *16th Annual Workshop of the Psychology of Programming Interest Group*, pages 97–109, 2004.

[27] M. Worsley and P. Blikstein. Programming pathways: A technique for analyzing novice programmers' learning trajectories. In *Artificial intelligence in education*, pages 844–847. Springer, 2013.

10.3 Discussion

The findings of this chapter show that the data collected from different stages of the semester do not show consistent features. This gives implications and clues to the importance of considering variations in the context. Investigating the quantity of the difference is beyond the scope of this PhD research, however the finding of this paper shows the importance of the researcher's level of awareness from the data and its features. In the next chapter, I set out to investigate the degree of context specificity of the data analysis result via a minimal variation made to the course structure. The finding of this chapter and the following chapter demonstrates the need for more context-independent methods of data analysis.

Chapter 11

On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions

11.1 Introduction

The result of the previous chapters showed that the information extracted from the source code snapshot can be used to perform data analytics with the goal of identification of students who are likely to struggle with the learning objectives of the course. I demonstrated that two main features including *attempts* and *correctness ratio* alone can predict the course outcome in a different context. However, since the models could be trained at different stages of the semester, the variances of the data used in the training set might in fact change the prediction performance of the prediction model. Hence, there is a need for a machine learning based model which relies on these two features while performing in a less context dependent fashion. This chapter represents a new method for analyzing the source code snapshot data which is based on contingency tables. A 2 by 2 contingency table can be interpreted as a confusion matrix, representing the association between two variables. The proposed method is easy to implement, more suitable to fit the real time data analysis tools, and is adaptable in different contexts.

11.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;
2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;
3. there are no other authors of the publication according to these criteria;
4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and
5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions

Conference: ACM Conference on Innovation and Technology in Computer Science Education

URL: http://dl.acm.org/citation.cfm?id=2899452&CFID=862572254&CFTOKEN=60379727

Status: Presented, July 2016

TABLE 11.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 11

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Raymond Lister	✓			✓
Arto Vihavainen				✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi		
Name	Signature	Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister		
Name	Signature	Date

11.2 PDF of the Published Paper

On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions

Alireza Ahadi and Raymond Lister
University of Technology, Sydney
Australia
alireza.ahadi@uts.edu.au
raymond.lister@uts.edu.au

Arto Vihavainen
Department of Computer Science
University of Helsinki
Finland
arto.vihavainen@cs.helsinki.fi

ABSTRACT

This paper explores the relationship between student performance on online programming exercises completed during semester with subsequent student performance on a final exam. We introduce an approach that combines whether or not a student produced a correct solution to an online exercise with information on the number of attempts at the exercise submitted by the student. We use data collected from students in an introductory Java course to assess the value of this approach. We compare the approach that utilizes the number of attempts to an approach that simply considers whether or not a student produced a correct solution to each exercise. We found that the results for the method that utilizes the number of attempts correlates better with performance on a final exam.

Keywords

Programming; educational data mining; learning analytics

1. INTRODUCTION

For decades, students who were learning to program submitted their assignments on paper. Thus, the only artifact available for analysis was a student's final program. But in recent years, with the advent of web-based systems for teaching programming, it is now possible to study the student behaviour that culminates in the final program.

Jadud [2] studied the sequence of source code *snapshots* generated by students, where a snapshot is collected each time a student compiles their code. Jadud introduced a metric that quantifies how students fix errors, which he called the *error quotient*. Using a version of the error quotient, Rodrigo et al. [4] found a strong correlation between error quotient and midterm score in an introductory programming course. Watson et al. [5] proposed an improvement to the error quotient called *Watwin*. Watson et al. also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with final course score.

ITiCSE '16, July 09-13, 2016, Arequipa, Peru
© 2016 ACM. ISBN 978-1-4503-4231-5/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2899415.2899452

1.1 Motivation for this Study

While the aforementioned research work on the error quotient and other measures is very promising, in this paper the authors study students in a simpler way, without directly analyzing student code, for the following reasons:

1. As practising teachers, we often find ourselves focusing on two simpler aspects of a student's performance on practical programming tasks, before we look at the student's code: (1) whether or not the student provided a piece of code that generates correct answers, and (2) how long it took the student to write the code.
2. As practising teachers, we also want ways of assessing the quality of the exercises we are giving the students.
3. As education researchers, we feel the work on the error quotient and other measures currently lacks a suitable benchmark, a simpler approach, upon which those more complicated approaches are incumbent to improve.

In the next section, we describe our simpler approach, which does not directly analyze student code. We then present and discuss results generated with this approach.

2. METHOD AND DATA

2.1 Educational Context and Data

The data for this study was collected from students enrolled in a 6 week introductory Java programming course at the University of Helsinki. In the course, 50% of the overall mark comes from completing online exercises during the 6 week semester, while the other 50% comes from a final exam. Furthermore, to pass the course, students must achieve at least half of the available marks in both the exercises and the final exam.

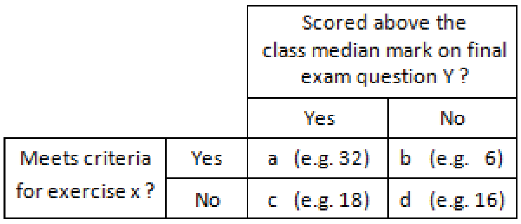
There are 106 online exercises available for inspection at http://mooc.fi/courses/2013/programming-part-1/material.html but only 77 of those exercises are used in the course. Students were allowed to make multiple submissions of an exercise, henceforth referred to as "attempts". Attempts could fail for both syntactic and semantic reasons, and the online system provides feedback to students. There were lab sessions where students could seek assistance with the exercises from teaching staff, but students were also allowed to work on and submit the exercises at any other time and place of their choosing. More information about the educational context and the data could be found in [1]. As data for this paper, each student's performance on each exercise was recorded as two values:

1. A dichotomous variable; 0 if the student did not succeed in answering the exercise correctly, or 1 if the student did provide a correct answer.
2. An integer; the number of attempts a student submitted for the exercise (irrespective of whether or not the student provided a correct answer).

The three final exam questions analyzed in this paper are provided in an appendix to this paper. As data for this paper, a student's performance on each of those three questions was recorded as a dichotomous variable; 0 if the student's mark for the question was below the class median mark for that question, or 1 if it was above the median.

2.2 Construction of Contingency Tables

A 2×2 contingency table is illustrated in Figure 1. The four variables in Figure 1, "a", "b", "c" and "d" represent the number of students who satisfy each of the four possible combinations of the two dichotomous variables, for exercise X and final exam question Y. The simplest example of a criterion for an exercise X is that students answered the exercise correctly. Given that criterion, then according to the example value in Figure 1 for "a", 32 students answered exercise X correctly and also scored above the class median mark on final exam question Y.



		Scored above the class median mark on final exam question Y ?	
		Yes	No
Meets criteria for exercise x ?	Yes	a (e.g. 32)	b (e.g. 6)
	No	c (e.g. 18)	d (e.g. 16)

Figure 1: A 2×2 Contingency Table for Exercise X and Final Exam Question Y.

2.2.1 *Number of Student Attempts*

More complicated criteria for exercise X in Figure 1 were constructed by combining whether or not students answered an exercise correctly with the number of attempts. That is, for each combination of exercise X and final exam question Y, a number of contingency tables were generated, where the dichotomous criteria for exercise X was that students answered the exercise ...

1. Correctly
2. Correctly, and in $<2^n$ attempts
3. Correctly, and in $\geq 2^n$ attempts
4. Incorrectly, and in $<2^n$ attempts
5. Incorrectly, and in $\geq 2^n$ attempts
6. In $<2^n$ attempts
7. In $\geq 2^n$ attempts

The largest number of attempts by a student on any exercise was 513, so "n" in each of the list items takes on all values from 0 to 10.

2.3 Measures of Performance

We use two measures to describe the relationship between an exercise and a final exam question, in terms of the values within the contingency table, which are described below.

2.3.1 *Accuracy*

Accuracy, which ranges from 0 to 1, is the fraction of occasions when a student either (1) meets the criteria on exercise X and is in the upper half of the class on final exam question Y, or (2) did not meet the criteria on exercise X and is in the lower half on exam question Y. Formally, in terms of the contingency table values shown in Figure 1, accuracy is defined as:

$$acc = \frac{a+d}{a+b+c+d} \tag{1}$$

2.3.2 *Phi Correlation Coefficient*

The phi correlation coefficient is a standard measure of the correlation of two binary variables. The Pearson correlation coefficient for two binary variables is equivalent to the phi coefficient. The phi coefficient ranges from 1 (where the two binary variables are always equal), through zero (where the two binary variables are not related), to -1 (where the two binary variables are never equal). The phi coefficient is computed for a 2 by 2 contingency table as follows:

$$phi = \frac{ad - bc}{\sqrt[2]{(a+b)\ (c+d)\ (a+c)\ (b+d)}} \tag{2}$$

2.4 Contingency Table Pruning Rules

To select the statistically significant and most useful contingency tables, the following pruning rules were used to eliminate some contingency tables. In these pruning rules, where a rule refers to two contingency tables, the two contingency tables are for the same exercise and same final exam question. Also, of those two contingency tables, the table with a wider range of student attempts is the more general table (e.g. $\geq$8 attempts is more general than $\geq$16 attempts).

1. Contingency tables with any cell value of less than 5 were pruned. This is a well known and widely used criterion for ignoring a contingency table, which reduces the likelihood of over-fitting a model to data, for the reasons explained in [6].
2. Contingency tables with a negative phi were pruned. Such contingency tables always have "mirror image" contingency table with a positive phi.
3. Contingency tables with $p > 0.01$ (χ^2 test) were pruned.
4. If the phi values of two contingency tables differed by less than 0.01, then the less general bin was pruned.
5. If the phi values of two contingency tables differed by more than 0.01, and the phi value of the more general contingency table was higher than the phi value of less general table, then the less general table was pruned.

6. If the phi value of a more general contingency table was lower than the phi value of another table, but a statistical test for significant difference (z score transformation [3]) revealed no significant difference ($p<0.05$), then the less general bin was pruned.

3. RESULTS

3.1 When number of attempts is ignored

Table 1 shows the exercises with the highest correlation to final exam questions 2, 3 and 4, when the number of attempts by students is ignored. The final exam question numbers are designated in the table's second column, the column headed "ExamQ". Questions 2, 3 and 4 from the final exam are provided as an appendix to this paper.

For final exam question 2, the three highest correlating exercises are 59, 70 and 61, as shown in the third column, headed "Ex". These three exercises (and all other exercises) are available for inspection at http://mooc.fi/courses/2013/programming-part-1/material.html. The column with the heading "Week" indicates that all the exercises are from weeks 3 and 4 of the 6 week semester.

The columns headed "Q1", "Q2" and "Q3" show the quartile boundaries for the number of attempts made by students. For example, row 1.1 of the table shows that 25% of the students made 10 attempts or less on the exercise, 50% of the students made 15 attempts or less, and 50% of the students made between 10 and 20 attempts. Note that these quartile boundaries are for all students, irrespective of whether or not they answered the exercise correctly.

The values in the column headed "Correct" indicate that, for the contingency table used to construct each row of this table, answering each exercise correctly is the sole "criteria on exercise X" in Figure 1.

The columns headed "Phi" and "Accuracy" show the measures of performance as defined earlier. For each exam question, the rows of Table 1 are ordered on "Phi", from highest to lowest. Both "Phi" and "Accuracy" in each row are calculated using a contingency table as shown in Figure 1. In Table 1, the columns headed "a", "b", "c" and "d" show the values for each of these contingency tables. For each row of Table 1, "Meet criterion on exercise x" in Figure 1 is "yes" if a student answered the exercise correctly. For example, column "a" in row 1.1 of Table 1 shows that 32 students answered exercise 59 correctly and also scored above the median mark on final exam question 2. The values recorded for "a", "b", "c" and "d" in that row of Table 1 are the four example values shown in Figure 1. The column "sum" is simply the sum of the values in columns "a", "b", "c" and "d", which show that this table used data from approximately 70 students. The sum values in that column vary as sometimes a student did not attempt an exercise. The column headed "p" shows the statistical significance of the contingency table for each exercise, using the standard χ^2 test. All exercises shown in this table easily meet the traditional $p<0.05$ criteria for statistical significance.

3.2 When number of attempts is considered

Table 2 shows the exercises with the highest correlation to final exam questions 2, 3 and 4, when the number of attempts made by students on an exercise is considered. Most of the columns in this table contain the same type of information as the previous table. The differences between the previous table and this table are:

- The column headed "Attempts" describes the range of the number of attempts a student must have made on an exercise as part of meeting the criteria on exercise X (as shown in Figure 1). For example, the "$\geq$8" in row 2.1 indicates that a student needed to make at least 8 attempts at the exercise to be counted within either cell "a" or "b" of the contingency table. In rows 2.6 and 2.7, "<16" indicates that a student needed to make less than 16 attempts to meet the criteria.
- Some rows in the column headed "Correct" contain an asterisk. Each of those asterisks indicates that whether a student answered the exercise correctly is irrelevant; the sole criterion for including a student in cell "a" or "b" of the contingency table is the number of attempts the student made on the exercise.
- Rows 2.8 and 2.9 contain "correct" in the column headed "Correct", indicating there are two criteria that need to be met for a student to be counted in either cell "a" or "b" of the contingency table: (1) the student must have answered the exercise correctly, and (2) the student must have done so in the number of attempts specified in the "Attempts" column.

4. DISCUSSION

There are differences between Tables 1 and 2, which illustrate the utility of considering the number of attempts students make, rather than focusing on correctness alone:

- Table 1 shows no exercise correlated significantly with final exam question 3, but Table 2 shows several exercises correlated with that question.
- For final exam question 2, the phi correlation and accuracy of each exercise is much higher in Table 2 than in Table 1.
- In Table 1, all the exercises listed are from weeks 3 and 4 of the 6 week semester. The authors believe that it is counter-intuitive that exercises from mid-semester would always correlate better with a final exam question than exercises done late in semester. That intuition is supported by Table 2, where 6 out of 10 exercises listed in Table 2 are from weeks 5 and 6.

Before generating the results in the tables, the authors had thought that the column "Attempts" in Table 2 would be dominated by criteria that placed an upper bound on the number of attempts, not a lower bound (i.e. we thought there would have been more "<" symbols in the "Attempts" column, not the "$\geq$" symbols that actually dominate). Our intuition was that stronger students would consistently complete exercises in fewer attempts than weaker students. Our explanation as to why "$\geq$" symbols dominate is three-fold:

1. Students do the exercises at any time or place of their choosing. Students who complete some exercises in an unusually small number of attempts may be receiving too much assistance from someone else.
2. The reader may recall one of the pruning rules for contingency tables; that the smallest value in any cell of

Table 1: The exercises with the highest correlation to final exam questions 2, 3 and 4, where the sole criterion is whether a student answered the exercise successfully; the number of attempts prior to success is ignored. No exercises correlated significantly ($p < 0.05$) with final exam question 3.

Row No.	ExamQ	Ex	Week	Q1	Q2	Q3	Correct	Phi	Acc	a	b	c	d	sum	p
1.1	2	59	3	10	15	20	correct	0.33	0.66	32	6	18	16	72	0.004
1.2	2	70	4	15	40	70	correct	0.32	0.63	29	5	21	17	72	0.005
1.3	2	61	3	11	14	20	correct	0.28	0.61	27	5	23	17	72	0.01
1.4	3	—	—	—	—	—	———	—	—	—	—	—	—	—	——
1.5	4	52	3	16	24	35	correct	0.34	0.66	30	6	18	17	71	0.004
1.6	4	59	3	10	15	20	correct	0.33	0.66	31	7	17	17	72	0.004
1.7	4	55	3	19	26	35	correct	0.31	0.65	30	7	18	17	72	0.007

Table 2: The exercises with the highest correlation to final exam questions 2, 3 and 4, when the number of attempts by students is considered.

Row No.	ExamQ	Ex	Week	Q1	Q2	Q3	Correct	Attempts	Phi	Acc	a	b	c	d	sum	p
2.1	2	92	5	4	53	91	*	$\geq$ 8	0.57	0.82	43	7	6	15	71	<0.001
2.2	2	102	6	2	113	199	*	$\geq$ 16	0.48	0.77	38	6	10	15	69	<0.001
2.3	2	103	6	1	86	190	*	$\geq$ 4	0.46	0.76	38	6	11	15	70	<0.001
2.4	3	92	5	4	53	91	*	$\geq$ 4	0.43	0.77	44	10	6	11	71	<0.001
2.5	3	93	5	2	40	69	*	$\geq$ 16	0.36	0.72	38	8	12	13	71	0.002
2.6	3	28	2	7	9	14	*	$<$ 16	0.35	0.75	46	12	6	9	73	0.003
2.7	3	49	3	12	14	18	*	$<$ 16	0.34	0.70	37	8	13	13	71	0.004
2.8	4	52	3	16	24	35	correct	$\geq$ 8	0.38	0.68	30	5	18	18	71	0.001
2.9	4	59	3	10	15	20	correct	$\geq$ 8	0.37	0.68	31	6	17	18	72	0.002
2.10	4	103	6	1	86	190	*	$\geq$ 16	0.37	0.70	34	7	14	15	70	0.002

the contingency table must be at least 5. Given that the data is only from approximately 70 students, many contingency tables that place an upper bound on the number of attempts are pruned.

3. The occurrence of a "$\geq$" symbol in the "Attempts" column is an indication that an exercise is non-trivial. We note that in rows 2.6 and 2.7, where "$<$" symbols appear, the values in columns "Q1", "Q2" and "Q3" indicate that most students required relatively few attempts to complete the exercise. If the desire of the instructor is to provide students with a set of exercise in which the level of difficulty increases slowly, then the dominance of "$\geq$" symbols may be an indication that some exercises need to be added to the pool to reduce sudden jumps in difficulty.

On initial consideration, the many asterisks in the "Correct" column of Table 2 might be thought to indicate that the value for students in doing the exercises resides in the effort of doing the exercises, more so than getting the exercises right. However, there is also a more prosaic explanation, which is related to the contingency table pruning rule that the smallest value in any cell of the contingency table must be at least 5. For example, consider column "b" in row 2.1 of Table 2. The value in that column is 7, so the associated contingency table only narrowly avoided being pruned. Adding the extra criterion that students must also get exercise 92 right would shift some of the students from cells "a" and "b" to cells "c" and "d". In doing so, the value in cell "b" is likely to drop from 7 to below 5. We note that in rows 2.8 and 2.9 of Table 2, where the selection criteria includes getting the exercise right, the values in the "b" column are 5 and 6, so the associated contingency tables only narrowly avoided being pruned. If data became available from many more students, it is the authors' suspicion that fewer asterisks would appear in the "Correct" column of Table 2.

Our method can identify gaps in a set of exercises. For example, the relatively low values of phi and accuracy for question 4, for all six exercises in both Tables 1 and 2, may indicate that the exercises did not prepare students well for this exam question – perhaps the exercises do not cover Object-Oriented concepts adequately.

Note that neither Table 1 or Table 2 show all the exercises that correlate significantly with each final exam question. Only the highest correlating exercises are shown.

4.1 Over-fitting

With data from only 70 students, a natural concern for any analysis is the danger of over-fitting. That is, there is a danger that the exercises selected for Table 2 exploit unrepresentative patterns in the relatively small data set; patterns that would not be present in a much larger data set. The specific problem with our method is that, for each pair of final exam question and exercise, there is only one contingency table that ignores the number of attempts, but there are several contingency tables that consider the number of attempts. For example, consider row 2.2 of Table 2. The column "Q3" indicates that a quarter of the students made 199 attempts or more at this exercise. If we assume that the highest number of attempts by any student was less than 512, there are 9 attempt ranges to consider: 1 attempt, 2-3 attempts, 4-7 attempts, 8-15 attempts ... 256-511 attempts. Furthermore, for each of those attempt ranges, there are two contingency tables: one that considers correctness and another that ignores correctness. It might therefore be argued that the reason why the exercises selected for Table 2 have higher phi and accuracy values is simply because there are more contingency tables to choose from when constructing Table 2. There are at least two reasons to discount that argument, which we describe in the remainder of this section.

The primary reason for discounting the danger of overfitting is the pruning rule that all values in a contingency table must be ≥ 5. Table 3 shows the number of contingency tables after pruning for each of the final exam questions, across all exercises, for both when the number of attempts at each exercise are ignored and when the number of attempts are considered. For final exam questions 2 and 4, the small ratio between contingency tables when attempts are considered and contingency tables when attempts are ignored (as shown in the final column of Table 3) is unlikely to be large enough to explain the consistent superiority of phi and accuracy values in Table 2 over Table 1.

The second reason for discounting the above argument about over-fitting is that the argument incorrectly assumes statistical independence among all contingency tables. Consider two contingency tables for a given exercise and final exam question, where both contingency tables either ignore correctness or both consider it. Furthermore, assume that one of the contingency tables is for the case where the number of attempts is $\geq 2^n$ and the other contingency table is for the case where the number of attempts is $\geq 2^{n+1}$. The students who meet the criteria for the latter contingency table also meet the criteria for the former contingency table, so the two tables are not statistically independent.

Table 3: The number of contingency tables after pruning for final exam questions 2, 3 and 4, across all exercises, when the number of attempts at each exercise are ignored and considered.

Row No.	ExamQ	Attempts Ignored	Attempts Considered	Ratio
3.1	2	10	35	3.5
3.2	3	0	16	—
3.3	4	15	34	2.3

5. CONCLUSION

Our method can be used to benchmark more sophisticated methods for analyzing student performance on coding exercises. But practising teachers can also use this approach to identify weaknesses in a set of exercises, and identify students who may need help. Furthermore, the information that emerges from our method is simple enough to provide to students, as a guide to how many attempts it might take them to complete an exercise. Doing so might calm some students who are slow to understand that programming is an iterative process. It might also act as an indication to other students that they either need to become more systematic in their approach, or they need to seek help from teaching staff.

6. REFERENCES

[1] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 121–130, New York, NY, USA, 2015. ACM.

[2] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[3] A. Papoulis. *Probability and Statistics*. Prentence-Hall International Editions, 1990.

[4] M. M. T. Rodrigo, E. Tabanao, M. B. E. Lahoz, and M. C. Jadud. Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science*, 138(2):177–190, 2009.

[5] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 319–323. IEEE, 2013.

[6] F. Yates. Contingency tables involving small numbers and the ÏĞ2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.

7. APPENDIX

The exam questions discussed in detail in this paper:

Question 2, part a
Create a program that outputs (using a loop statement such as while or for) all integers divisible 2, starting with 1000 and ending in 2. The output must occur so that 5 integers are printed on each row, and that each column must be aligned. The program output should look like this:

```
1000 998 996 994 992
 990 988 986 984 982
 980 978 976 974 972
      (lots of rows)
  10   8   6   4   2
```

Question 2, part b
Create a program where the input is integers representing the exam points gained by students. The program starts by reading the numbers of points from the user. The reading of the points stops when the user enters the integer -1.

The number of points must be an integer between 0 and 30. If some other integer is input (besides -1 that ends the program), the program ignores it.

After reading the numbers of points, the program states which number of points (between 0 and 30) is the greatest. Out of the number of points, the integers under 15 are equivalent to the grade *failed*, and the rest are *passed*. The program announces the number of passed and failed grades.

Example:

```
Enter numbers of exam points, -1 ends the program:
20
12
29
15
-1
best number of points: 29
passed: 3
failed: 1
```

In the above example, 12 points failed and the points 20, 29 and 15 passed exams. Thus, the program announces that 3 students passed and 1 student failed.
Please note that the program must ignore all integers outside 0-30. An example of a case where there are integers that have to be ignored among the input numbers:

```
Enter numbers of exam points, -1 ends the program:
10
100
20
-4
30
-1
best number of points: 30
passed: 2
failed: 1
```

As shown, the points -4 and 100 are ignored.

Question 3, part a
Create the method `public static void printInterval(int edge1, edge2)` that prints, in ascending order, each integer in the interval defined by its parameters.

If we call `printInterval(3, 7)`, it prints

```
3 4 5 6 7
```

The methods also works if the first parameter is greater than the second one, i.e. if we call `printInterval(10, 8)`, it prints

```
8 9 10
```

Thus, the integers are always printed in ascending order, regardless of which method parameter is greater, the first one or the second one.

Question 3, part b
Create the method `public static boolean bothFound(int[] integers, int integer1, integer2)`, which is given an integer array and two integers as parameters. The method returns true if both integers given as parameters (`integer1` and `integer2`) are in the array given as method parameter. In other cases the method returns false.

If the method receives as parameters for example the array [1,5,3,7,5,4], and the integers 5 and 7, it returns true. If the method received the array [1,5,3,2] and the integers 7 and 3 as parameters, it would return false.

Create a main program, as well, which demonstrates how to use the method.

Note! If you don't know how to use arrays, you can create `public static boolean bothFound(ArrayList <Integer> integers, int integer1, int integer2)`, where the method is given as parameters an ArrayList containing the integers and the integers to be found.

Question 4.
Create the class Warehouse. The warehouse has a capacity, which is an integer, and the amount of wares stored in the warehouse is also stored as an integer. The warehouse capacity if specified with the constructor parameter (you can assume that the value of the parameter is positive). The class has the following methods:

- `void add(int amount)`, that adds the amount of wares given in the parameter to the warehouse. If the amount is negative, the status of the warehouse does not change. When adding wares, the amount of wares in the warehouse cannot grow larger than the capacity. If the amount to be added does not fit into the warehouse completely, the warehouse is filled and the rest of the wares are 'wasted."
- `int space()`, that returns the amount of empty space in the warehouse.
- `void empty()`, that empties the warehouse.
- `toString()`, which returns a text representation of the warehouse status, formulated as in the example below; observe the status when the warehouse is empty!

Next is an example that demonstrates the operations of a warehouse that has been implemented correctly:

```
public static void main(String[] args) {
  Warehouse warehouse = new Warehouse(24);
  warehouse.add(10);
  System.out.println(warehouse);
  System.out.println("space in warehouse "
    + warehouse.space());
  warehouse.add(-2);
  System.out.println(warehouse);
  warehouse.add(50);
  System.out.println(warehouse);
  warehouse.empty();
  System.out.println(warehouse);\\
```

if the class has been implemented correctly, the output is
capacity: 24 items 10
space in warehouse 14
capacity: 24 items 10
capacity: 24 items 24
capacity: 24 empty

11.3 Discussion

In this Chapter, I presented a new method for the analysis of the source code snapshot data with the goal of identification of the students who are likely to struggle in the subject. This method has two primary differences with the work presented in Chapter 6. Firstly, this method is rooted more in statistical data mining rather than machine learning. Secondly, it deploys multiple statistical measurements where each of the measurements correspond to a specific research question. Therefore, we are not talking about a single target class of *fail* or *pass* labels, and are focused on multiple aspects of the learning progress. In the following Chapter, I extend this method to include more statistical measurements to provide a way to find the answers to the more frequently asked research questions.

Chapter 12

A Contingency Table Derived Method for Analyzing Course Data

12.1 Introduction

As reviewed in the previous chapter, there are a considerable number of quantitative metrics which could be calculated from a single contingency table. These metrics can be used to answer different research questions and each have an interest in looking at the data from a particular perspective. This chapter represents the extended version of the data analytics methodology presented in previous chapters. In this chapter, new metrics are introduced, implemented and interpreted with an aim to answer different questions of the computer science education researcher.

12.1.1 Statement of Contribution of Co-Authors

The authors listed below have certified that:

1. they meet the criteria for authorship in that they have participated in the conception, execution, or interpretation, of at least that part of the publication in their field of expertise;

2. they take public responsibility for their part of the publication, except for the responsible author who accepts overall responsibility for the publication;

3. there are no other authors of the publication according to these criteria;

4. potential conflicts of interest have been disclosed to (a) granting bodies, (b) the editor or publisher of journals or other publications, and (c) the head of the responsible academic unit; and

5. they agree to the use of the publication in the student thesis and its publication on the QUT ePrints database consistent with any limitations set by publisher requirements.

In the case of this chapter:

Title: A Contingency Table Derived Methodology for Analyzing Course Data

Journal: ACM Transactions on Computing Education

URL:

Status: Submitted, October 2017

TABLE 12.1: Authors' Area of Contribution for The Paper Corresponding to Chapter 12

Contributor	Area of contribution (See appendices A and B)			
	(a)	**(b)**	**(c)(i)**	**(c)(ii)**
Alireza Ahadi	✓	✓	✓	✓
Raymond Lister	✓			
Arto Hellas	✓			✓

Candidate confirmation:

I have collected email or other correspondence from all co-authors confirming their certifying authorship and have directed them to the principal supervisor.

Alireza Ahadi

Name Signature Date

Principal supervisor confirmation:

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Raymond Lister

Name	Signature	Date

12.2 PDF of the Submitted Paper

A Contingency Table Derived Method for Analyzing Course Data

ALIREZA AHADI, University of Technology, Sydney
ARTO HELLAS, University of Helsinki
RAYMOND LISTER, University of Technology, Sydney

We describe a method for analyzing student data from online programming exercises. Our approach uses contingency tables that combine whether or not a student answered an online exercise correctly with the number of attempts that the student made on that exercise. We use this method to explore the relationship between student performance on online exercises done during semester with subsequent performance on questions in a paper-based exam at the end of semester. We found that it is useful to include data about the number of attempts a student makes on an online exercise.

CCS Concepts: • **Mathematics of computing** → **Contingency table analysis**; • **Information systems** → **Data mining**; • **Social and professional topics** → **Computer science education**;

Additional Key Words and Phrases: Data mining, programming novices, Java

ACM Reference Format:
Alireza Ahadi, Arto Hellas, and Raymond Lister. 2017. A Contingency Table Derived Method for Analyzing Course Data. *ACM Trans. Comput. Educ.* 17, 3, Article 13 (August 2017), 19 pages.
https://doi.org/10.1145/3123814

13

1 INTRODUCTION

Recent studies of novice programmers have used dynamically accumulated online data, such as source code snapshots [2, 16, 22, 41]. The methods used to analyze such data range from statistical analysis of a single variable constructed from the programming process—such as the *error quotient* [22] or the *watwin-score* [41]—to the use of a multiple variables combined by machine learning methods [2].

In this article, we use contingency tables to represent student behavior in online exercises and explore the relationship between that behavior and student performance on final exam questions. For the purposes of this study, we have extracted log data from students' programming processes that contains the number of attempts and the degree of achieved correctness for each online exercise.

Authors' addresses: A. Ahadi, Faculty of Engineering and Information Technology, University of Technology, Sydney, 15 Broadway, Ultimo NSW 2007, Australia; email: Alireza.Ahadi@uts.edu.au; A. Hellas, Department of Computer Science, Ernst Lindelöfin katu 1, 00560 Helsinki Finland; email: Arto.Hellas@cs.helsinki.fi; R. Lister, Faculty of Engineering and Information Technology, University of Technology, Sydney, 15 Broadway, Ultimo NSW 2007, Australia; email: Raymond.Lister@uts.edu.au.

https://doi.org/10.1145/3123814

Using our method based on contingency tables, we have studied the following research question:

RQ: Does the number of attempts students make on an online programming exercise during semester relate to student performance on final exam questions?

This article is organized as follows. In Section 2, we review the related literature on analyzing students' performance and provide an overview of contingency tables and their statistical measures of performance. This is followed by a description of the data used for this study, as well as the method used for constructing the contingency tables. In Section 4, we apply the method to our data and study how well the contingency tables identify different student populations. In Section 5, we discuss the results, and finally, in Section 6, we conclude the article with suggestions for future work.

2 BACKGROUND

Early work on attempting to identify success factors for programming focused on variables that preceded the period in which the novice learns to program. Such variables included demographic profiles, high school achievement, prior programming experience, cognitive styles, and generic problem-solving abilities. Evans and Simkin [19] described these variables as approximations for *programming aptitude*. With the development of recent online technologies, there has been a move to studying data collected during the period in which the novice is learning to program [22, 29, 39, 41]. In this article, we focus on such data.

2.1 Continuously Accumulating Data

Continuously accumulating data are generated as a byproduct of students using various learning management systems. One technologically unsophisticated example is the analysis of course transcripts to discover sequences that are often associated with course failure [17].

Another example, which uses more sophisticated technology, is the analysis of in-class clicker behavior. Porter et al. [29] studied students' clicker responses in a peer instruction setting and identified that the percentage of correct clicker answers from the first three weeks of a course was strongly correlated with overall course performance.

While the previous examples are related to coarser grained data, a stream of research on continuously accumulating programming process data has recently become more popular [21]. In 2006, Jadud suggested quantifying a student's ability to fix errors in a sequence of source code snapshots (collected at code compilations) and found that there was a high correlation with final exam score [22]. In essence, this suggests that the less programming errors a student makes, and the more successful she is in solving them, the higher her grade will tend to be [22, 34]. The method has also been explored in other institutional contexts, with the same BlueJ programming environment and Java programming language, and with similar results [34, 41], while studies with a different programming environment have reported somewhat different results [2, 16].

Watson et al. [41] improved on Jadud's results by including the amount of time that students spent on programming assignments [41]. Watson et al. also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with programming course scores [41]. More recently, a similar approach that analyzes programming states was proposed by Carter et al. [16]. Piech et al. studied students' approaches to solving programming tasks and found that the solution patterns are indicative of course midterm scores [28]. Different types of patterns were also studied by Hosseini et al., who analyzed students' behaviors within a programming course: some students were more inclined to build their code step by step, while others started from larger quantities of code and then reduced their code to reach a

Table 1. A 2 by 2 Contingency Table

		V_1		
		Category=α	Category=β	row totals
V_2	Category=γ	a	b	r_1
	Category=δ	c	d	r_2
	column totals	c_1	c_2	n

solution [20]. A separate and more recent stream of research has sought to model students' understanding of fine-grained source code-based concepts using source code snapshots [31, 46].

2.2 Approaches for Data Analysis

The approaches used for analyzing student data can be broadly categorized into two main groups: (1) single- and multivariate regression analysis and (2) machine learning and data mining approaches.

2.2.1 Single- and Multivariate Regression. The use of single- and multivariate regression is the primary analysis method in the literature. Numerous articles discuss the connection between a specific variable and introductory programming course outcomes [6–8, 14, 26, 36, 37, 42–44], whilst less attention has been invested in multivariate analysis [9, 15, 33, 35, 38, 47].

2.2.2 Machine-Learning and Data Mining Approaches. This approach is becoming more common as it provides insight into data regardless of dimensional complexity. Some uses of this approach are described in the remainder of this subsection.

Data mining techniques have been used to predict course outcomes using data from enrollment forms [23] and also data from students' self-assessment [24]. The mining techniques used range from basic machine learning algorithms to more advanced techniques such as decision trees [30]. There have also been experiments in using real-time data to detect student failure [4].

Machine learning and data mining techniques have also been used in computer science education. Ahadi et al. [1] described the common syntactic mistakes of novices writing SQL "select" statements. Using that data, they trained decision trees to predict student outcomes in a database course. Most other studies have focused on the novice programmer. Lahtinen [25] used statistical cluster analysis to place novice programmers into clusters based on Bloom's Taxonomy. Berland [10] mined source code snapshots to categorize novice programmers into "planners" and "tinkerers." In a follow-up study, Berland determined that students progress through three stages, from "exploration," through "tinkering," to "refinement" [11]. Blikstein [13] described an automated technique to assess, analyze, and visualize the behavior of novice programmers.

2.3 Contingency Tables

In statistics, a contingency table is a matrix that displays the frequency distribution of two categorical variables. In this article, we restrict ourselves to dichotomous variables, V_1 and V_2, which is illustrated as a 2×2 contingency table in Table 1. The two values for each variable are shown in Table 1 as α, β, γ, and δ. In Table 1, the values in the four cells (a, b, c, and d) represent the frequency of each category. For example, a, is the frequency for V_1 equals α and simultaneously V_2 equals γ. The values r_1, r_2, c_1, and c_2 correspond to the row and column sums; for example, r_1 is the sum of a and b, and c_1 is the sum of a and c. The sample size $n = r_1 + r_2 = c_1 + c_2$.

V_1 and V_2 may in reality be numeric variables, in which case the binary categorization is determined by a threshold value. The choice of threshold value is theoretically arbitrary, but intuitive choices are the mean or median value.

In the context of computing education research, examples of variables V_1 and V_2 could be whether or not a student answered an exercise correctly, the time spent on answering the exercise, the number of compilation errors while answering the exercise, or the number of attempts needed to answer the exercise; this last example is a variable of particular interest in this article, where an attempt is any action leading to an indication of whether or not the student's answer is correct, such as a compilation, a test run by the student, or an upload to a testing system.

Statistical measures of performance associated with contingency tables are summarized in Table 2, where the variables a, b, c, and d on the right-hand side of the equations are the same as in Table 1.

3 METHOD

In this section, we describe the data we used and the method we devised for constructing and selecting the contingency tables.

3.1 Data

The data for this study is from 96 students in an introductory programming course conducted in 2014 at the University of Helsinki. The course topics are similar to many introductory programming courses around the world and include input and output, variables, loops, lists, and objects. The course is taught in Finnish, but English translations (albeit dated) of the course materials are available online.[1] The students used an integrated development environment (IDE), and their programming process data was captured using a plugin called TestMyCode [40]. For each exercise, the IDE plugin stored the details of every key press made by a student.

Each of the 77 exercises is graded using a suite of unit tests, which have been constructed to provide step-wise feedback to students as they work on the exercises. For the purposes of this study, the correctness of each exercise has been mapped to either 1 if a student's score on an exercise is equal to or above the median score of the class, and 0 if it is not. Moreover, when counting the number of attempts made by a student, we only counted running, testing, and submitting the exercise.

A student's overall course grade is composed of the online exercises (worth 50%) and a paper-based final exam consisting of five questions (the remaining 50%). The first question in the final exam required students to explain course concepts, while the other four questions were program writing tasks. That first exam question was excluded from analysis in this article. The remaining questions (Q2–Q5), which are given in Appendix A, focused on the following topics: (Q2) algorithmic problem solving with loops, on small problems, (Q3) the use and construction of methods that include algorithmic problem solving, (Q4) the creation and use of classes, where the students must write a class for a specified domain, and (Q5) the creation and use of a composite object (i.e., an object that contains a list of other objects).

3.2 Notation

In our approach, we combine whether or not a student answered an online exercise correctly with the number of attempts that the student made on that exercise. To describe such combinations precisely but concisely, we use the notation described in this section.

[1] http://mooc.fi/courses/2013/programming-part-1/material.html.

Table 2. Statistical Measures of Performance

Measure	Explanation and Equation, where *a*, *b*, *c*, and *d* are from Table 1
Precision	Also known as Positive Predictive Value (PPV) is the proportion of predicted positive cases that are actually positive [32]. $PPV = \frac{a}{a+b}$
Recall	Also known as sensitivity [3] measures the portion of actual positives that are correctly identified as such. $TPR = \frac{a}{a+c}$
F1-Score	Is the harmonic mean of precision and recall [18]. It can also be interpreted as a weighted average of precision and recall. $F1 - Score = \frac{2a}{2a+b+c}$
Specificity	Also known as True Negative Ratio [3] measures the portion of actual negatives that are correctly identified as such. $SPC = \frac{d}{b+d}$
Negative Predictive Value	The ratio of actual negatives to actual and false negatives [32]. $NPV = \frac{d}{d+c}$
Accuracy	Also known as Overall Fraction Correct, is defined as the quotient of the number of correct classifications and the sample size [12]. $ACC = \frac{a+d}{n}$
Matthews Correlation Coefficient	Also known as the "phi coefficient" (Φ), it is a measure of the degree of association between two binary variables derived from the Pearson product-moment correlation coefficient [5]. $MCC = \Phi = \frac{ad - bc}{\sqrt{r_1 * r_2 * c_1 * c_2}}$

Table 3. The Seven Variables Used and Associated Research Questions

Contingency table	Sample research question
`X(x)C(1)A(*)`	How does performing well on exercise x relate to performance on final exam question y?
`X(x)C(*)A(<2`^{n}`)`	How does the number of attempts on exercise x, regardless of correctness, relate to performance on final exam question y?
`X(x)C(*)A(>2`^{n}`)`	(similar to above)
`X(x)C(1)A(<2`^{n}`)`	How does performing well on exercise x within a certain number of attempts influence the performance in final exam question y?
`X(x)C(1)A(>2`^{n}`)`	(similar to above)
`X(x)C(0)A(<2`^{n}`)`	How does attempting but not performing well on exercise x relate to performance on final exam question y?
`X(x)C(0)A(>2`^{n}`)`	(similar to above)

Let `X(x)` denote exercise x. Let `C(a)` denote the correctness a for that exercise, where $a = 1$ refers to a score on the exercise that is greater than or equal to the median score for the whole class, and $a = 0$ means a score lower than the median. Let `A(n)` denote the number of attempts n for that exercise. For example, `X(23)C(1)A(4)` refers to those students who answered exercise 23 with a score equal to or greater than the median score, in exactly four attempts.

We bucket the number of attempts students make into intervals based on powers of two. Let $A(< 2^n)$ refer to any number of attempts less than 2^n, and let $A(> 2^n)$ refer to any number of attempts greater than 2^n.

Sometimes we do not combine correctness and the number of attempts. This is indicated by an asterisk. For example, `X(23)C(*)A(<2`^{3}`)` refers to those students who, on exercise 23, made less than 8 attempts (irrespective of correctness) and `X(23)C(1)A(*)` refers to those students who had a score higher than median score on exercise 23 (irrespective of the number of attempts).

3.3 Selecting the Variables of Interest

Table 3 describes the seven variables we used in the construction of the contingency tables and provide examples of research questions that each of these contingency tables could be used to answer. In the examples, we assume that the other variable forming the contingency table is one of the code writing final exam questions, $2 \leqslant y \leqslant 5$.

The maximum number of different contingency tables generated for each combination of online exercise and final exam question is 61. However, a combination of online exercise and final exam question often generated fewer contingency tables, because we only generate attempt buckets up to the smallest power of two that exceeds the maximum number of attempts made by any student on that exercise.

3.4 Pruning the Contingency Tables that Contain Little Information

To select a limited number of contingency tables that represent a general and statistically significant association between investigated variables, the following pruning rules were applied:

(1) Contingency tables with a cell value (i.e., a, b, c, or d) of less than 5 are pruned, since the chi-square is suspect if values are less than 5 [45].
(2) Contingency tables with a non-significant chi-square p-value ($p > 0.01$) are pruned.

(3) If the phi coefficient is the same for two attempt buckets related to the same exercise, then the less general of the attempt buckets is pruned. That is, the attempt bucket that covers a wider range of number of attempts is kept, and the other bucket is discarded.
(4) If the phi coefficients of two attempt buckets related to the same exercise are different, and if the phi value of the more general bucket is higher than the phi value of the less general bucket, then the less general bucket is pruned.
(5) For a given exercise, if the phi value of the more general bucket is less than the phi value of the less general bucket, a test of the significance of the difference between the two phi values is performed (using z score transformation [27]). If there is no significant difference between the two phi values, then the less general bucket is pruned.
(6) Contingency tables with an Accuracy or a F1-score of less than 0.5 are pruned.

3.5 Statistical Measures of Performance for the Proposed Variables

In this section, we briefly describe again the statistical measures of performance described in Table 2, but here in the context of student performance on the exercises and exam questions:

- The **precision** of a given contingency table `X(x)C(1)` indicates to what degree students who performed well on exercise x scored above the median on a final exam question.
- The **recall** of a given contingency table `X(x)C(1)` indicates to what degree students who scored above the median on a final exam question also performed well on exercise x.
- The **specificity** of a given contingency table `X(x)C(1)` indicates to what degree students who scored less than the median score on a final exam question also performed poorly on exercise x.
- The **negative predictive value** indicates to what degree students who scored less than the median score on a final exam question also performed poorly on a given exercise.
- The **accuracy** of a contingency table `X(x)C(1)` could be interpreted as the percentage of the students whose performance on a final exam question could be directly identified from their performance on exercise x. (This definition, however, applies to only those contingency tables for which the number of positive and negatives samples are equal.)
- With respect to the **Matthews Correlation Coefficient**, the closer $|\Phi|$ is to one, the stronger the association between student performance on an exercise and the final exam question.

4 ANALYSIS AND RESULTS

We analyzed data from the introductory programming course outlined in Section 3.1. The maximum number of attempts by a student on any of the 77 online exercises was 513.

Approximately half (55.4%) of the contingency tables were pruned, because the table had a cell containing a number less than 5. Of the remaining contingency tables, only 4.9% ($N = 410$) had $p < 0.01$. After the remaining pruning rules were applied, 330 contingency tables remained. Given that there are $4 \times 77 = 308$ pairings of exam questions to online exercises, 330 contingency tables is an approximate average of only one contingency table for each pair (cf. up to 61 tables prior to pruning). Such a low average after pruning suggests that the results we present below are not simply due to the overfitting of models to data.

4.1 Exercises and Final Exam Questions with the Highest Correlation

Identification of those exercises that have the highest correlation with final exam questions can be determined using the Matthews Correlation Coefficient. Table 4 shows the top four exercises with highest correlations to each final exam question. For most of these highest correlating exercises,

Table 4. The Four Exercises with Highest Correlation to Each Final Exam Question

Exercise	Correctness	Attempts	Exam question	MCC
92	*	>7	Q2	0.56
102	*	>15		0.48
103	*	>3		0.46
93	*	>15		0.46
92	*	>3	Q3	0.43
93	*	>15		0.36
28	*	<15		0.35
49	*	<15		0.34
52	1	>7	Q4	0.38
59	1	>7		0.37
103	*	>15		0.36
44	*	>7		0.36
70	*	>15	Q5	0.54
102	*	>3		0.52
70	*	>7		0.50
92	*	>15		0.47

Table 5. Number of Exercises with Significant Correlation to Final Exam Questions

Exam question	No. significant tables	No. exercises in those tables	Phi mean
Q2	45	29	0.30
Q3	16	15	0.29
Q4	49	30	0.28
Q5	60	33	0.31

correctness is not an issue; just the number of attempts. Therefore, the answer to our research question is "yes." That is, the number of attempts a student makes on a programming exercise does contain useful information; specifically, there is a relationship between the number of attempts on a programming exercise and a student's performance on a final exam question.

Some caution should be exercised, however, before concluding that getting an online exercise correct is less important than the number of attempts on that exercise. Inspection of some contingency tables suggests that the pruning of tables with a cell value less than 5 may favour tables where correctness is not considered. Consider two contingency tables for the same exercise, the same exam question, and same attempt bucket. Let one table be for $C(*)$ and the other for $C(1)$. It may be the case that the two tables have very similar values for a, b, c, and d, but one cell in the table for $C(1)$ has a value just under 5 and is therefore pruned, but the value in corresponding cell of the table for $C(*)$ is just above 5 and is therefore not pruned.

Some of the high-ranked exercises in Table 4 appear more than once. Exercise 92 appears in three out of four questions. That exercise has been included in Appendix B.

4.2 Number of Exercises Associated with Each Final Exam Question

Table 5 shows, for each final exam question, the number of online exercises with a significant correlation to that exam question. This information indicates which online exercises are especially helpful in preparing students for each of the final exam questions.

Table 6. The Precision, Recall and F1-Score of given Contingency Tables and Final Exam Question 2

Exercise	Correctness	Attempts	Precision	Recall	F1 score
92	*	>7	0.86	0.88	0.87
70	*	>15	0.80	0.86	0.83
102	*	>15	0.86	0.79	0.83
93	*	>15	0.85	0.80	0.82
103	*	>3	0.86	0.78	0.82
99	*	<32	0.76	0.88	0.82
28	*	<16	0.76	0.88	0.81
69	*	>7	0.75	0.90	0.81
29	*	<16	0.74	0.90	0.81
82	*	>3	0.76	0.86	0.81
59	1	*	0.84	0.64	0.73
52	1	<64	0.85	0.59	0.70
55	1	<64	0.83	0.60	0.70
70	1	*	0.85	0.58	0.69
52	1	>7	0.83	0.59	0.69
52	1	<128	0.83	0.59	0.69
56	1	*	0.81	0.60	0.69
55	1	*	0.81	0.60	0.69
23	1	<64	0.82	0.59	0.68
56	1	<64	0.83	0.58	0.68

4.3 Identifying Students who Are Likely to Perform Well

Table 6 represents Precision, Recall, and F1-score of the twenty highest ranking contingency tables (out of 45) that correlate with final exam question two. The rows of the table are ordered on F1-score, the harmonic mean of precision and recall.

Note that the F1-Score of all $C(*)$ rows in the table are higher than the F1-Score of all $C(1)$ rows. Thus once again the answer to our research question is "yes." That is, the number of attempts a student makes on a programming exercise does contain useful information. However, once again some caution should be exercised before concluding that getting an online exercise correct is less important than the number of attempts on that exercise, due to the pruning of tables with a cell value less than 5.

4.4 Identifying Students who are Likely to Perform Poorly

Precision, Recall, and F1-Score are measures that focus on positive cases; in our case, on students who perform well. None of those measures capture information on negative cases; in our case, on students who perform poorly.

The statistical measure dealing with unsuccessful students is specificity. We inspected the top ten contingency tables according to their specificity in predicting the students' performance on Q2 of the final exam. In all cases, however, the negative predictive value of those contingency tables was less than 0.5, indicating that these contingency tables performed at a level less than chance at identifying poor students. That we could not, from the exercises, identify students who would subsequently perform poorly on Q2 is probably attributable to the nature of the exercises and/or the exam question, rather than our method.

Table 7. Ten Contingency Tables with Highest Accuracy for Q2 of Final Exam

Exercise	Correctness	Attempts	ACC	MCC
92	*	>7	0.81	0.56
102	*	>15	0.76	0.48
93	*	>15	0.76	0.46
103	*	>3	0.75	0.46
70	*	>15	0.75	0.38
28	*	<15	0.72	0.31
99	*	<31	0.72	0.29
82	*	>3	0.72	0.32
69	*	>7	0.71	0.26
21	*	>15	0.71	0.29

4.5 General Evaluators of Students' Performance

Whilst the previous metrics have focused on identifying high-performing students and low-performing students, it is also meaningful to consider metrics that would be useful in identifying performance in general. Here, accuracy is an appropriate choice, assuming that the categorization of variables is balanced. Accuracy characterizes the increase in probability of a student being in the top 50% on the final exam question, for those answering the exercise correctly, and the decrease in probability for those answering the exercise incorrectly. Table 7 represents the accuracy and Matthews Correlation Coefficient of the ten contingency tables with the highest accuracy for Q2 in the final exam.

In all the rows of Table 7, correctness does not matter. The accuracy of all the statistically significant contingency tables related to Q2 (not just the top ten shown in Table 7) range from 0.57 to 0.81. Both the mean and median of those contingency tables where correctness does not matter is 0.72, which is relatively higher than the mean and median of those contingency tables where correctness does matter, 0.61 and 0.62, respectively. Once again, the number of attempts a student makes on a programming exercise does contain useful information. But also once again some caution should be exercised before concluding that getting an online exercise correct is less important than the number of attempts on that exercise.

The MCC values in the rows of Table 7 are not high, implying that individual exercises are not highly accurate predictors of performance on Q2 of the final exam.

5 DISCUSSION

Before we produced our results, our hypothesis was that the exercises that would separate low- and high-achieving students would be the final exercise for each week, as these exercises essentially divided the students into those who worked on all the exercises and those who did not. However, our results show this is not the case. The last exercise of a specific week were typically not in the list of exercises that separate low and high achieving students. Thus, it was not the sheer number of exercises that students did that led to above-median performance on exam questions.

5.1 What Makes for a Highly Predictive Exercise?

For some contingency tables, we observed high accuracies (up to 81% with 0.56 MCC). The exercises associated with these contingency tables were inspected. Many of the high-correlating exercises were from late in the course and combined multiple concepts (e.g., both object-oriented programming and algorithmic thinking). Also, these exercises tended to contain concepts that were the culmination of a sequence of concepts. For example, consider exercise 92 (see Appendix A), which correlated relatively highly with three of the four analyzed exam questions. This exercise requires the student to augment a "Date" class to include a method that calculates the difference between the date within an object and the date within another object passed as a parameter. To complete the exercise, the student needed to be able to work with the notion of "this," also an object passed as a parameter, as well as being able to perform calculations.

5.2 The Number of Attempts

To better understand the importance of the number of attempts at an exercise, we compared those contingency tables that only used information about the number of attempts (i.e., X(x) C(*) A(either $< 2^n$ or $> 2^n$), to those contingency tables that ignored the number of attempts, and only used information about correctness (i.e., X(x) C(1) A(*)). We observed that none of the contingency tables from the latter category (i.e., ignored attempts) correlated significantly with the final exam question related to constructing methods, while several contingency tables from the former category showed a significant and high correlation. This may indicate that for the student to succeed in the exam question related to methods, it is important that she practices, but she does not necessarily have to get every exercise right. Moreover, the timing of the practice is important—the majority of exercises in the contingency tables of the former category are from weeks 3 and 4 of the 6-week course—when the exercises first focus on building simple methods.

5.3 Data Granularity and Validity

Different tools collect data at different levels of granularity [21]. While some tools record each keystroke, other tools may generate snapshots on line edit, compile, run, or save actions or only when students submit the program.

For any study similar to this study, it is important to have a clear definition of an attempt. If snapshots are generated on each compile, and the number of compilations is used as the measure of attempts, then what exactly does the snapshot count represent? A straightforward metric for effort may be the number of key strokes. If that was our metric, suppose that two students both answered an exercise correctly with a single compile, but one student's program was much longer than the other student's program—then would it be appropriate to conclude that the student with the longer program had put in more effort? In our study, we used actions that are related to saving, running, testing and compiling the code—actions that have the virtue of being easily collected by several tools—but exactly what does a count of those actions mean?

5.4 Limitations and Concerns

The quality of data could be sensitive to subtle institutional settings. For example, in a programming course in which answering an exercise correctly is a must for unlocking the next exercise, it is likely that some struggling students will answer an exercise "correctly" by cheating. In the class from which we collected our data, students needed to attain 85% of the points from one week's batch of exercises before they could access the following week's exercises.

Accuracy has good predictive value as it is based on all cells of the contingency table. However, accuracy is a less useful metric when the number of instances in each class ($a + c$ versus $b + d$) are not balanced. To address this problem with accuracy, we selected for our data an equal number of successful students and unsuccessful students.

The data for this study is from 96 students, which is a relatively small sample. A small sample size automatically raises the issue of over-fitting. That is, the results found with these 96 students might not be present in a larger dataset. It might be argued that the reason why some contingency tables have high correlation and accuracy values is simply because there were up to 61 tables to choose from for each pair of exercise and final exam question. However, there are at least two reasons to discount that argument. The primary reason for discounting the danger of over-fitting are the pruning rules, especially the rule that the value in each of the four cells of a contingency table must be at least five. As we have already explained in the results section, after pruning there was approximately an average of only one contingency table for each pairing of exam question to online exercise. Such a low average is a strong argument against overfitting.

The second reason for discounting the above argument about over-fitting is that the argument incorrectly assumes statistical independence among the contingency tables. To illustrate why all the contingency tables are not statistically independent, consider two contingency tables for a given exercise and final exam question, where both contingency tables either ignore correctness or both consider it. Furthermore, assume that one of the contingency tables is for the case where the number of attempts is $>2^n$ and the other contingency table is for the case where the number of attempts is $>2^{n+1}$. The students who meet the criteria for the latter contingency table also meet the criteria for the former contingency table, so the two tables are not statistically independent.

6 CONCLUSION

The results presented in this article demonstrate that there can be a statistically significant relationship between the number of attempts on a programming exercise and performance on a final exam question. Almost half of our exercises had a moderate to strong predictive value for a final exam question, when the number of attempts at the exercise was taken into account.

The pedagogical benefit of our approach is twofold. First, students who are making an inordinate number of attempts at exercises could be identified early and offered help. Second, exercises that require a high average number of attempts for success could be analyzed to identify topics that are vital to success but hard to understand.

One of the key advantages of the proposed method is its strength in handling small datasets, such as the ones that researchers typically gather in computing education. Given the small number of features used in the construction of the contingency table, and our pruning rules, the likelihood of over-fitting the data is low.

We are extending this analysis to include students' background details, such as their course major. We will also analyze the highly predictive exercises in more detail to tease out the factors related to students' performance.

Our method of using contingency tables could be used in many contexts. We are considering making our method of data analysis publicly available via a web-server. The server could not only be then used by researchers, but non-research educators might also use the server to evaluate their own exercises.

APPENDIX

This appendix first provides the four final exam questions that were used to construct the second variable of the contingency tables explored as the case study. Then, we provide a sample exercise—here 92—that outlines the way the exercises are presented to the students.

A EXAM QUESTIONS

Question 2

Part A (3p)

Create a program that outputs (using a loop statement, such as while or for) all integers divisible with 2, starting with 1000 and ending in 2. The output must occur so 5 integers are printed on each row, and each column must be aligned. The program output should look like this:

```
 1000   998   996   994   992
  990   988   986   984   982
  980   978   976   974   972
             (lots of rows)
   10     8     6     4     2
```

Part B (4p)
Create a program where the input is integers representing the exam points gained by students. The program starts by reading the numbers of points from the user. The reading of the points stops when the user enters the integer −1.

The number of points must be an integer between 0 and 30. If some other integer is input (besides −1 that ends the program), then the program ignores it.

After reading the numbers of points, the program states which number of points (between 0 and 30) is the greatest. Out of the number of points, the integers under 15 are equivalent to the grade *failed*, and the rest are *passed*. The program also announces the number of passed and failed grades.

Example:

```
Enter numbers of exam points, -1 ends the program:
20
12
29
15
-1
best number of points: 29
passed: 3
failed: 1
```

In the above example, 12 points failed and the points 20, 29, and 15 passed exams. Thus, the program announces that three students passed and one student failed.

Please note that the program must ignore all integers outside 0–30. An example of a case where there are integers that have to be ignored among the input numbers:

```
Enter numbers of exam points, -1 ends the program:
10
100
20
-4
30
-1
best number of points: 30
passed: 2
failed: 1
```

As shown, the points −4 and 100 are ignored.

Question 3

Part A (3p)
Create the method `public static void printInterval(int edge1, edge2)` that prints, in ascending order, each integer in the interval defined by its parameters.

If we call `printInterval(3, 7)`, then it prints

```
3 4 5 6 7
```

The method also works if the first parameter is greater than the second one, that is, if we call `printInterval(10, 8)`, then it prints

```
8  9  10
```

Thus, the integers are always printed in ascending order, regardless of which method parameter is greater, the first one or the second one.

Part B (3p)

Create the method `public static boolean bothFound(int[] integers, int integer1, integer2)`, which is given an integer array and two integers as parameters. The method returns true if both integers given as parameters (`integer1` and `integer2`) are in the array given as method parameter. In other cases, the method returns false.

If the method receives as parameters, for example, the array [1,5,3,7,5,4], and the integers 5 and 7, then it returns true. If the method received the array [1,5,3,2] and the integers 7 and 3 as parameters, then it would return false.

Create a main program, as well, which demonstrates how to use the method.

Note! If you don't know how to use arrays, then you can create `public static boolean bothFound (ArrayList<Integer> integers, int integer1, int integer2)`, where the method is given as parameters an ArrayList containing the integers and the integers to be found.

Question 4 (6 points)

Create the class Warehouse. The warehouse has a capacity, which is an integer, and the amount of wares stored in the warehouse is also stored as an integer. The warehouse capacity if specified with the constructor parameter (you can assume that the value of the parameter is positive). The class has the following methods:

- `void add(int amount)`, that adds the amount of wares given in the parameter to the warehouse. If the amount is negative, then the status of the warehouse does not change. When adding wares, the amount of wares in the warehouse cannot grow larger than the capacity. If the amount to be added does not fit into the warehouse completely, then the warehouse is filled and the rest of the wares are "wasted."
- `int space()`, which returns the amount of empty space in the warehouse.
- `void empty()`, which empties the warehouse.
- `toString()`, which returns a text representation of the warehouse status, formulated as in the example below; observe the status when the warehouse is empty!

Next is an example that demonstrates the operations of a warehouse that has been implemented correctly:

```
public static void main(String[] args) {
    Warehouse warehouse = new Warehouse (24);
    warehouse.add(10);
    System.out.println(warehouse);
    System.out.println("space in warehouse " + warehouse.space());
    warehouse.add(-2);
    System.out.println(warehouse);
    warehouse.add(50);
    System.out.println(warehouse);
    warehouse.empty();
    System.out.println(warehouse);
}
```

if the class has been implemented correctly, then the output is

```
capacity: 24 items 10
space in warehouse 14
capacity: 24 items 10
capacity: 24 items 24
capacity: 24 empty
```

Question 5 (6 points)

This assignment is about making a program to manage the contents of a bookshelf. You have at your disposal the class Book:

```
public class Book{
    private String author;
    private String title;

    public Book(String author, String title) {
        this.author = author;
        this.name = name;
    }

    public String getAuthor() {
        return this.author;
    }
    @Override
    public String toString() {
        return this.author + ": " + this.name;
    }
}
```

Please program the class Bookshelf, which works like the example described below:

```
public static void main(String[] args) {
    Bookshelf shelf = new Bookshelf();
    shelf.addBook("Kent Beck", "Test Driven Development");
    shelf.addBook("Kent Beck", "Extreme Programming Embraced");
    shelf.addBook("Martin Fowler", "UML Distilled");
    shelf.addBook("Fedor Dostoyevski", "Crime and Punishment");

    shelf.print();
    System.out.println();
    shelf.get("Kent Beck");
}
```

if the class has been implemented correctly, then the output is

```
books total 4
books:
Kent Beck: Test Driven Development
Kent Beck: Extreme Programming Embraced
Martin Fowler: UML Distilled
Fedor Dostoyevski: Crime and Punishment

found:
Kent Beck: Test Driven Development
Kent Beck: Extreme Programming Embraced
```

The class must save books added to the shelf in an ArrayList containing Book objects.

As we can see in the example, the following methods have to be implemented for the class:

- `void addBook(String author, String title)`, which adds a book with the author and title given as parameters to the shelf.
- `void print()`, which prints the information of the bookshelf formulated as in the example above.
- `find(String author)`, which outputs the books in the shelf with the author given as method parameter, the output should be in the same form as in the example above.

B EXERCISES

Exercise 92: Difference of two dates

[This assignment has been formatted for printing. Students receive a ready-made class called MyDate, and their goal in this assignment is to add features to the class.]

First, add the method public int differenceInYears(MyDate comparedDate) to the class MyDate. The method should calculate the difference in years between the object for which the method is called and the object given as parameters.

Note that the first version of the method is not very precise as it only calculates the difference of the years and does not take the day and month of the dates. The method needs to work only in the case where the date given as parameter is before the date for which the method is called.

Try out your code with the following:

```
MyDate first = new MyDate(24, 12, 2009);
MyDate second = new MyDate(1, 1, 2011);
MyDate third = new MyDate(25, 12, 2010);

print(second + " and " + first + " difference in years: " +
    second.differenceInYears(first));
print(third + " and " + first + " difference in years: " +
    third.differenceInYears(first));
print(second + " and " + third + " difference in years: " +
    second.differenceInYears(third));
```

The output should be:

```
// as 2011-2009 = 2
1.1.2011 and 24.12.2009 difference in years: 2
// as 2010-2009 = 1
```

```
25.12.2010 and 24.12.2009 difference in years: 1
// as 2011-2010 = 1
1.1.2011 and 25.12.2010 difference in years: 1
```

Next, you need to add more accuracy to the method. Calculation of the previous version was not very exact, for example, the difference of dates 1.1.2011 and 25.12.2010 was one year, which is not true. Modify the method so it can calculate the difference properly: only full years in difference count. So, if the difference of two dates would be 1 year and 364 days, only the full years are counted and the result should be one.

The method still needs to work only in the case where the date given as parameter is before the date for which the method is called.

After improving the code, the output for the previous example should be:

```
1.1.2011 and 24.12.2009 difference in years: 1
25.12.2010 and 24.12.2009 difference in years: 1
1.1.2011 and 25.12.2010 difference in years: 0
```

Finally, modify the method so it works no matter which date is the latter one, the one for which the method is called or the parameter.

```
MyDate first = new MyDate(24, 12, 2009);
MyDate second = new MyDate(1, 1, 2011);

print(first + " and " + second + " difference in years: " +
    second.differenceInYears(first));
print(second + " and " + first + " difference in years: " +
    first.differenceInYears(second));
```

```
24.12.2009 and 1.1.2011 difference in years: 1
1.1.2011 and 24.12.2009 difference in years: 1
```

REFERENCES

[1] Alireza Ahadi, Vahid Behbood, Julia Prior, Arto Vihavainen, and Raymond Lister. 2016. Students' syntactic mistakes in writing seven different types of SQL queries and its application to predicting students' success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16).* 401–406.

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER'15).* ACM, New York, NY, 121–130. DOI: https://doi.org/10.1145/2787622.2787717

[3] D. G. Altman and J. M. Bland. 1994. Diagnostic tests. 1: Sensitivity and specificity. *BMJ* 308, 6943 (1994), 1552.

[4] Sadaf Fatima Salim Attar and Y. C. Kulkarni. 2015. Precognition of students academic failure using data mining techniques. *Int. J. Adv. Res. Comput. Commun. Eng.* (2015).

[5] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus A. F. Andersen, and Henrik Nielsen. 2000. Assessing the accuracy of prediction algorithms for classification: An overview. *Bioinformatics* 16, 5 (2000), 412–424.

[6] Ricky J. Barker and E. A. Unger. 1983. A predictor for success in an introductory programming class based upon abstract reasoning development. In *ACM SIGCSE Bull.* 15. ACM, 154–158.

[7] Jens Bennedsen and Michael E. Caspersen. 2006. Abstraction ability as an indicator of success for learning object-oriented programming? *ACM SIGCSE Bull.* 38, 2 (2006), 39–43.

[8] Susan Bergin and Ronan Reilly. 2005. Programming: Factors that influence success. *ACM SIGCSE Bull.* 37, 1 (2005), 411–415.

[9] Susan Bergin and Ronan Reilly. 2006. Predicting introductory programming performance: A multi-institutional multivariate study. *Comput. Sci. Edu.* 16, 4 (2006), 303–323.

[10] M. Berland and T. Martin. 2011. Clusters and patterns of novice programmers. In *Proceedings of the Meeting of the American Educational Research Association.* New Orleans, LA.

[11] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using learning analytics to understand the learning pathways of novice programmers. *J. Learn. Sci.* 22, 4 (2013), 564–599.

[12] IEC BiPM, ILAC IFCC, IUPAC ISO, and OIML IUPAP. 2008. International vocabulary of metrology—Basic and general concepts and associated terms, 2008. *JCGM* 200 (2008), 99–12.

[13] Paulo Blikstein. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge.* ACM, 110–116.

[14] Pat Byrne and Gerry Lyons. 2001. The effect of student attributes on success in programming. In *ACM SIGCSE Bull.* 33. ACM, 49–52.

[15] Brenda Cantwell Wilson and Sharon Shrock. 2001. Contributing to success in an introductory computer science course: A study of twelve factors. In *ACM SIGCSE Bull.* 33. ACM, 184–188.

[16] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER'15).* ACM, New York, NY, 141–150. Retrieved from DOI: https://doi.org/10.1145/2787622.2787710

[17] E. Chandra and K. Nandhini. 2010. Knowledge mining from student data. *Eur. J. Sci. Res.* 47, 1 (2010), 156–163.

[18] Eric Gaussier and Cyril Goutte. 2005. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *Lect. Notes Comput. Sci.* 3408. 345–359.

[19] Gerald E. Evans and Mark G. Simkin. 1989. What best predicts computer proficiency? *Commun. ACM* 32, 11 (1989), 1322–1327.

[20] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring problem solving paths in a java programming course. In *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group.*

[21] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR'15).* ACM, New York, NY, 41–63. DOI: https://doi.org/10.1145/2858796.2858798

[22] Matthew C. Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2nd International Workshop on Computing Education Research.* ACM, 73–84.

[23] Zlatko J. Kovačić and J. S. Green. 2010. Predictive working tool for early identification of "at risk" students (2010).

[24] S. Anupama Kumar and M. N. Vijayalakshmi. 2011. Implication of classification techniques in predicting students recital. *Int. J. Data Mining Knowl. Manage. Process (IJDKP)* 1, 5 (2011), 41–51.

[25] Essi Lahtinen. 2007. A categorization of novice programmers: A cluster analysis study. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group, Joensuu, Finnland.* Citeseer, 32–41.

[26] R. R. Leeper and J. L. Silver. 1982. Predicting success in a first programming course. In *ACM SIGCSE Bull.* 14. ACM, 147–150.

[27] Athanasios Papoulis. 1990. *Probability and Statistics.* Prentence-Hall International Editions.

[28] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12).* ACM, New York, NY, 153–160. DOI: https://doi.org/10.1145/2157136.2157182

[29] Leo Porter, Daniel Zingaro, and Raymond Lister. 2014. Predicting student success using fine grain clicker data. In *Proceedings of the 10th Annual Conference on International Computing Education Research.* ACM, 51–58.

[30] M. N. Quadri and N. V. Kalyankar. 2010. Drop out feature of student data for academic performance using decision tree techniques. *Global J. Comput. Sci. Technol.* 10, 2 (2010).

[31] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning curve analysis for programming: Which concepts do students struggle with? In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER'16).* ACM, New York, NY, 143–151. DOI: https://doi.org/10.1145/2960310.2960333

[32] Suzanne W. Fletcher and Robert H. Fletcher. 2005. *Clinical Epidemiology: The Essentials.* Vol. 1. Lippincott Williams and Wilkins.

[33] Maria Mercedes T. Rodrigo, Ryan S. Baker, Matthew C. Jadud, Anna Christine M. Amarra, Thomas Dy, Maria Beatriz V. Espejo-Lahoz, Sheryl Ann L. Lim, Sheila A. M. S. Pascua, Jessica O. Sugay, and Emily S. Tabanao. 2009. Affective and behavioral predictors of novice programmer achievement. *ACM SIGCSE Bull.* 41, 3 (2009), 156–160.

[34] Maria Mercedes T. Rodrigo, Emily Tabanao, Maria Beatriz E. Lahoz, and Matthew C. Jadud. 2009. Analyzing online protocols to characterize novice Java programmers. *Philippine J. Sci.* 138, 2 (2009), 177–190.

[35] Nathan Rountree, Janet Rountree, Anthony Robins, and Robert Hannah. 2004. Interacting factors that predict success and failure in a CS1 course. In *ACM SIGCSE Bull.* 36. ACM, 101–104.

[36] Michael V. Stein. 2002. Mathematical preparation as a basis for success in CS-II. *J. Comput. Sci. Colleges* 17, 4 (2002), 28–38.

[37] Markku Tukiainen and Eero Mönkkönen. 2002. Programming aptitude testing as a prediction of learning to program. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group.* 45–57.

[38] Philip R. Ventura Jr. 2005. Identifying predictors of success for an objects-first CS1 (2005).

[39] Arto Vihavainen. 2013. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Proceedings of the IEEE 13th International Conference on Advanced Learning Technologies (ICALT'13).* IEEE.

[40] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education.* ACM, 117–122.

[41] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the IEEE 13th International Conference on Advanced Learning Technologies (ICALT'13).* IEEE, 319–323.

[42] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2014. No tests required: Comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education.* ACM, 469–474.

[43] Laurie Honour Werth. 1986. *Predicting Student Performance in a Beginning Computer Science Class.* Vol. 18. ACM.

[44] Susan Wiedenbeck, Deborah Labelle, and Vennila N. R. Kain. 2004. Factors affecting course outcomes in introductory programming. In *Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group.* 97–109.

[45] F. Yates. 1934. Contingency tables involving small numbers and the 2 test. *Suppl. J. Roy. Stat. Soc.* 1, 2 (1934), 217–235.

[46] Michael Yudelson, Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Investigating automated student modeling in a Java MOOC. In *Proceedings of the 7th International Conference on Educational Data Mining 2014.*

[47] Daniel Zingaro. 2014. Peer instruction contributes to self-efficacy in CS1. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE'14).* 373–378. DOI: https://doi.org/10.1145/2538862.2538878

Received September 2016; revised April 2017; accepted May 2017

Chapter 13

Discussion and Conclusion

13.1 Introduction

In this Chapter, I will review the research carried out throughout my PhD candidature. I will be giving an overview on what I have done, what I have learned, and discuss what could be done as future work to further improve the automated identification of the difficulties that a novice programmer experiences in learning to code. More specifically, I will focus on the challenges with the methods and the source code snapshot data collected from novice programmers.

13.2 Overview of Research

The primary research question that I have been trying to answer could be paraphrased as *using programming source code snapshot data, how can we identify the students who are in need of assistance in time?* In this chapter I review my answer to that question by looking at the programming source code snapshot data collected from two different contexts: Java programming, and database queries (SQL). The collected data from students have been analyzed quantitatively where different static (traditional) and dynamic success factors have been analyzed to identify correlation between the students coding attempts and their results when assessed by different means including final exam mark, the programming assignment marks, and the success ratio in answering other coding tasks.

13.3 Research Questions

The primary focus of this dissertation has been to answer the following research questions.

RQ1 Is it possible to identify struggling students by analyzing the source code snapshot data?

RQ2 Can we address the problem of the sensitivity of the prediction (of the struggling novice programmers) in a more context-independent manner?

The work done in this dissertation has successfully answered both proposed research questions: the combination of machine learning techniques, statistical data mining, and the source code snapshot data have lead to design a predictive model which is able to identify students in need of assistance at early stages of the semester. Considering context independent factors and using dynamic source code snapshot data and static success factor together have helped to design a tool (ArAl) which does not suffer from the effects which are due to local changes.

13.4 Research Outcome

The primary outcome of my research is a programming source code snapshot analysis tool (ArAl) which is implemented as an online web-server (see http://online-analysis-demo.herokuapp.com/). Aral is mainly designed to assist computer science researchers to perform data analysis on the source code snapshot data collected from their institutes. The tool relies on information on the quantity of students' attempts on specific problems, whether the student was able to solve the problem, and on students' course outcomes. The tool produces a set of metrics for the researcher; these metrics include information on assignments and their predictive power over course exam.

13.5 Research highlights

During my candidature, I was mainly focused on using different **analysis methods** on **data with different nature** collected from **different contexts**. A combination of rich data and machine learning / data mining tools helped me to get deep insights on the quality of how novice programmers handle different coding tasks. The main focus of my research is on the notion of an **attempt**, with a general definition that could be applied to different contexts: *how many steps does it take a novice programming to tackle a coding task successfully?* I also observed that the notion of **attempt** can be quantified using different parameters such as time, number of steps, number of mistakes, etc. Appropriate selection of the data analysis tool, awareness of the details of the context from which the data is collected, and fine grained rich dataset all play an important role in the interpretation of the results.

13.6 Research Significance

Multiple studies of novice programmers have used dynamically accumulated online data, such as source code snapshots (Jadud, 2006; Watson, Li, and Godwin, 2013; Ahadi et al., 2015; Carter, Hundhausen, and Adesope, 2015). The methods used to analyze such data range from statistical analysis of a single variable constructed from the programming process – such as the *error quotient* (Jadud, 2006) or the *watwin-score* (Watson, Li, and Godwin, 2013) – to the use of a multiple variables combined by machine learning methods (Ahadi et al., 2015). The findings of the research carried out by the author of this PhD thesis and the above mentioned authors will redound to the benefit of the computer science education research society considering that generating strong performing programmers plays an important role in teaching IT courses today. The greater demand for graduates with strong coding background justifies the need for more effective, life-changing teaching approaches. Thus, scholars that apply results and the recommended data analysis approach derived from the results of this study will be able to train students better. Administrators will be guided on what should be emphasized by instructors in the school curriculum to improve students' performance in programming/database courses. My findings will help the computer science education researchers to uncover critical areas in the educational processes that many researchers were not able to explore.

13.7 Research Findings

In this section, I'll review what I have found throughout my research. I will review my finding based on the results of the publications which together form the backbone of this thesis.

13.7.1 Data

Nowadays, the majority of the programming IDEs have the capability of collecting data from the programming environments. The data collected by these tools usually have the form of 1) metadata which reports features of different events such as compilation, project open, line edit, etc, and 2) the source code snapshot data which reports the state of the code when the above mentioned events are triggered. Error messages are one of the most important means of communication between the programming environment and the programmers.

Error messages facilitate the progress of the user and, as a troubleshooting technique, error messages help the user understand the problem underlying the error. An insufficient or incomplete error message can be misleading to the programmer who could spend an enormous amount of time spotting the source of the problem. Multiple studies have attempted to analyze the errors encountered by the novice programmers. The study of the error messages generated by the novice programmer has roots in the interest of recognition of difficult topics and misconceptions of the novices of programming. This stream of research gives insight in the challenges that novice programmers face when learning to code. The analysis of such data also generates insights into how to improve such error messages in a way that leads to better learning outcomes.

Error messages are not the only dynamic factor which can quantify the degree to which a student is struggling. As this thesis demonstrates, the *number of attempts* required to complete a programming task is also a strong determinant of success in programming. Related parameters such as the time spent on the programming task, the number of edits given to a specific line of code, the amount of time spent on different methods with different difficulty levels, and the number of compiles were also found to contribute to model/s for predicting students' success with another programming task, or course grade.

13.7.2 Method

Machine learning, and more specifically, data mining techniques were shown to be useful tools in finding the answer to the research questions. Supervised machine learning techniques were used in two publications presented in this thesis. These techniques helped to 1) train models that classify students into multiple groups based on the data collected from their programming environment, 2) find associations between the class that a student belongs to and their performance in the course, and 3) early identification of those students who might experience difficulty during the programming course.

13.7.3 Context

I investigated the data collected from two contexts: the Java programming IDE, and the SQL server log files collected from an online database assessment tool. The nature of the data collected from these two different environments is different hence the data mining techniques which are applicable to one context was not necessarily applicable to the other context. A change of parameters was needed to fit the model which was successfully used in the other context.

13.8 Discussion

In this section, I'll open discussions on different aspects of the research carried out. I will focus on answering the question *what else I could possibly understand from my findings?*

13.8.1 Data

As I discussed, the notion of attempt can be quantified using different parameters such as number of steps taken to complete a programming task, the number of compiles, the number of errors encountered, the time required to complete a programing task, etc. But what concepts exactly are quantified by the number of attempts? Are we quantifying how much the novice struggles with the code, or is it an indicator of how much thinking the student has put in the code? Even if equipped with the best explanation of what the number of attempts means, can we conclude that students with a lower number of attempts will eventually be better programmers? Perhaps it is context dependent to draw a conclusion on how the number of attempts can explain the learning aptitude. On the other side, it is not certain how the notion of number of attempts can be operational in terms of explaining the intention behind the code. How this factor can quantify why a student chose to get the answer to a question right by taking less steps?

The trained classifiers investigated in this thesis were based on data collected from students early in the semester. A student's learning experience however has a dynamic nature throughout the semester. Would I get the same sort of results if I trained my classifiers based on the data collected at different stages of the semester? For example, wouldn't the accuracy level of the classifier be higher if the training model was based on the data collected from one week before the final exam?

A strong classifier would be able to predict the same course outcome if it was given the information collected from the same student on another test. However, due to lack of data, it was not possible to replicate the experiments in multiple tests. Would the final exam mark of a student who performed moderately in a programming exercise be classified correctly if the training model was based on the data collected on the very same week but from another programming exercise?

13.8.2 Method

The basis of the majority of supervised machine learning techniques is to group the input based on the similarity between the data points. That is, for a given set of students, if two students fall into the same cluster, then they share similar values for their static and dynamic success factors. A fundamental question in this case would be if the data itself is capturing all the necessary information to form such clusters. This affects the design of the training model. Since the majority of data mining techniques are sensitive to the quality of the data, then perhaps the data analysis technique selected to analyze the data should be less sensitive to the data points. On the other side, there is always a small portion of the data points which are not properly captured by the training model. In designing different classifiers, I noticed that decision trees and neural networks seem to be less sensitive to such outliers; hence decision trees and neural networks are recommended by the author of this thesis analyzing source code snapshot data. However, if the goal of the data mining technique is to group students based a very small number of dynamic and static success factors, then the contingency table approach introduced in this thesis is recommended.

Sequential data mining has received attention from the educational data mining community, but not to the same degree in the computer science education community, especially those who study the novice programmer. Sequential data mining looks at the data (in this case, the data could be students' submission to the automated grading system, or the sequential data collected from the programming environment) and tries to find patterns in students coding. These patterns can give the instructors clues on how students go about coding. Would a combination of a sequential data mining and the demographic data collected from students reveal common pattern in their coding and if so, what conceptual framework would explain that best?

13.8.3 Context

As an instructor, I have seen many cases where two students with completely different learning styles arrive at the same result. Thus two students might have different learning styles but be categorised as members of the same cluster. It is important to know what course specific parameters should be captured by the data and used by the classifiers to model students groups based on their activity similarity. These factors change from subject to subject, course to course, and institute to institute.

Replication of a research study is a good strategy to check if the results from one context can be applied in many other contexts: would I get the same sort of results if I used my trained model in other contexts?

13.9 Limitations

Here, I'll be reviewing my findings with a focus on answering the question *what aspects of the data, context and the method limited and affected my research?*

13.9.1 Data

The source code snapshot data is limited in nature. The primary limitation of the source code snapshot data is its inability to capture students intention. It is extremely difficult to say what exactly the novice has been thinking about when writing the code. It is hard to tell if pressing the compile button was for the purpose of checking the correctness of syntax of a line of code, or simply a habit? Such data doesn't provide enough information on a student's *plan* in tackling the programming exercise. This becomes even more complicated in the case for database SQL *SELECT* statements as there are no compilation buttons in that environment and hence pressing enter after the semicolon cannot be interpreted as an act of checking the syntax but instead as a check of the semantics.

There are also limitations to the data availability as well. At the moment of writing this PhD thesis, only a limited number of programming source code snapshot databases are freely available to the public.

13.9.2 Context

Changes in context is one of the main reasons why replication studies are not able to generate the same results as an original study. It is important to know the data and the environment the data is collected from. One of the issues experienced throughout this research was the lack of demographic data on the students. Also, the programming source code snapshot data collected at Helsinki university was generated by students in a non-invigilated environment. Thus, it was not clear if the Helsinki students did their programming assessments together or individually.

13.10 Recommendations

In this section, I'll recommend what else can be done in future research to get more insights from the data collected from the novices. I will break down my recommendations based on the data, method, and the context from which the data was collected.

13.10.1 Data

Static data

Prior to the beginning of the semester, the instructors should start collecting information on students static success factors such as gender, age, programming experience, maths scores, and etc. Some of this information is available in university's information systems but others need to be collected through surveys. It is never a bad practice to collect as much data from the student as possible prior to the beginning of the semester.

Dynamic data

At the moment, the majority of the programming IDEs collect both the metadata and the data from the programming environment. This data needs to become available to other researchers. At the moment, only two programming source code snapshot tools are publicly available. This is the primary issue regarding the data and the research done to study the novice programmer. The limitations in the publicly available datasets forces the researcher to collect their own data which usually takes at least a few months.

13.10.2 Method

Using different data analytics techniques, I realized that there is no unique technique capable of classifying every single student accurately. The accuracy level of the class prediction varies but is usually less than 85%.

13.10.3 Features of a strong algorithm to capture students learning from the source code snapshot data

In this section, I will propose a set of features which needs to be considered in the designing process of future data analysis algorithms of the novice programmer.

General Attributes

The strength of a metric in quantifying learning aptitude is dependent on a variety of parameters. I argue that the operationalization of the metric can be best achieved by comparing an individual's performance on one programming task with another programming task. Also, a strong metric should demonstrate *replicability*. That is, a strong metric should report a similar error profile/progress for the same individual on two near-identical programming tasks done in quick succession.

Other attributes to be considered when designing a metric are ease of implementation, language independence, applicability, context independence, no use of free parameters, minimal sensitivity to the bias, and population independence.

The last item is concerned with the fact that the amount of progress that a student makes should not be compared to a cohort of students but with the previous learning state of the same individual. We believe that comparing an individual to a whole student cohort is related to the applicability of a particular teaching strategy and the difficulty level of a particular exercise, rather than the student's learning ability.

Language Independence

The majority of universities use Java as the primary programming language for their CS1 course, while some universities use Python, C or even Perl. Peterson et al Petersen, Spacco, and Vihavainen, 2015 reported the impact of the language taught on the outcome of the EQ metric and its derivations. In general, programming languages can be divided into two groups: compilation based, or interpreted languages. Compilation based errors are either compile-time errors or run-time errors. In an interpreted language, both of these error types are detectable through interpretation. In general, a semantic mistake which could manifest itself in a run-time error is not as easily detected as it is in a compilation based language: there is a need for the piece of code to be actually *run* so that the run-time error is revealed. This is not necessarily the case for interpreted languages. This fundamentally is a matter of differences in the data generated in different environments, however a strong metric for quantifying learning to code should not be affected crucially by such changes.

The definition of what is called an error is also in many cases arbitrary: many syntax errors in one language are not regarded as errors in other languages, but rather presented as warnings. Even if one has fixed all compilation errors, the compiler of some languages (such as C and C++) may still give you "warnings". These warnings won't keep the code from compiling (unless the compiler is asked to treat warnings as errors). These warnings could be treated as errors in another programming language.

I believe that a new metric profiling students' learning ability to code should not be trained based on the features of a specific language, but rather be based on *universal features* that are all programming languages have in common.

Distribution Independence

In the metric proposed by Watson et al, the time required to fix the error encountered by each novice is compared with the rest of the population. Based on the median and the standard deviation of the time spent on fixing the given code, a penalty value of either 1 or 26 is considered in the calculation of the final Watwin score. The fundamental issues with comparing one's quantified attribute of coding (in this case, the time spent to fix the code) to the rest of the population is that the given parameter is **a)** forming a big part of the calculation in the given algorithm, **b)** without including other contributing features to the element of time, this comparison overwrites the hidden effect of those features, and **3)** there are other features which are not present in the algorithm which can distinguish one particular student from others. In this case a student who has spent a lot more time on the code compared to others has not necessarily failed to complete the exercise successfully. Including time and comparing it with the whole population is highly dependent on context and could be very misleading. For example, the majority of students in the institute where the data is collected and analyzed to evaluate EQ and its deviations in this thesis, finish the exercises very quickly and leave the test room. This is not due to the fact that they are good programmers, but mainly because their primary way of handling the code is to memorize it. Our analysis shows that these students in fact get lower marks in the final exam compared to those who demonstrate some engagement with understanding the code.

Cross-Context Parameter Variation in the Construction of a Metric

Some features used in the construction of a metric are highly affected by the changes in the context. For example, the element of time is directly affected by the condition under which the data is generated. Under exam conditions, a novice wouldn't necessarily have enough time to sit and think about how to fix the code in a particular exercise. However, if the data is collected from students working on their assignments in an unsupervised condition, then it is almost impossible to say what exactly the novice has been doing: the novice could have left the machine after facing an error, gone to have some lunch, while another could have been talking about it on the phone to a friend. Thus, such attributes are not good contributors in the construction of a strong metric as they are greatly impacted by educational settings.

Validity of the Operationalization

Operationalization is the process through which an abstract concept is translated into measurable variable/s. Operationalized concepts are related to theoretical concepts but are not coincident with each other. The major problem with operationalization is the problem of validity. How can one be sure that the operational measurement still measures the theoretical concept? There are no ways in which validity can be rigorously "tested" because of the break between theory and practice (empirical data) that is integral to the quantitative research tradition. Also, to what extent can one quantify the concept? For example, what is the difference between a novice with an EQ score of 0.4 and a novice with a score of 0.6? A robust validated metric should consider representing a measurement which is truly reflective of how much the novice has learned to code.

13.10.4 Context

There is a need for a data collection doctrine which explains in detail what sort of information should be collected from the novices, what sort of information should be collected from the data generation environment, and what sort of information should be collected from the programming environment At the moment, the present data collection tools do not follow a specific common set of steps in collecting the data from the programmers.

13.11 Conclusion

The purpose of this research was to study the data from novice computer programmer in a way that has not been done before: a machine learning based framework which can best predict who it going to struggle to learn programming and who is not. The results of this research have valuable pedagogical implications.

Appendix A

Definition of Authorship and Contribution to Publication

The *Australian Code for the Responsible Conduct of Research* defines authorship as having substantial contributions in a combination of:

a. conception and design of a project;

b. analysis and interpretation of research data;

c. (i) drafting significant parts of a work;

c. (ii) critically revising it so as to contribute to the interpretation.

Appendix B

Specific Contributions of Co-Authors for Thesis by Published Papers

This appendix provides specific information about the contribution of each of the authors of the papers presented in this thesis.

Chapter 5: Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant? [ICER 2013]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor. Planned the research methodology with principal supervisor. Planned the data collection with the principal supervisor. Conducted the data collection and collation. Analyzed the data under the principal supervisor's guidance. Co-wrote the outline of the paper with the principal supervisor. Took the lead in writing the paper with the principal supervisor. Revised the paper for publication with input from reviewers and principal supervisor.

Lister, Raymond: (Principal Supervisor)
Provided editorial feedback on drafts and revisions. Co-wrote the outline of the paper with the candidate. Co-wrote the paper with the candidate. Presented the paper at the conference.

Chapter 6: Exploring machine learning methods to automatically identify students in need of assistance. [ICER 2015]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor and the research colleague. Planned the research methodology with the research colleague. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper with the principal supervisor. Took the lead in writing the paper with the research colleague. Revised the paper for publication with input from reviewers, principal supervisor and the research colleague.

Lister, Raymond: (Principal Supervisor)
Provided guidance in designing the research methodology. Provided feedback in writing the paper.

Haapala, Heikki: (Research Colleague)
Provided feedback on the manuscript and the research methodology.

Vihavainen, Arto: (Research Colleague)
Provided the data, co-wrote the paper and presented the paper.

Chapter 7: A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries. [ITiCSE 2015]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor and the research colleagues. Planned the research methodology with the research colleagues. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper with the principal supervisor and the research colleagues. Took the lead in writing the paper with the research colleagues. Revised the paper for publication with input from reviewers, principal supervisor and the research colleagues. Presented the paper at the conference.

Prior, Julia: (Research Colleague)
Provided the data, provided feedback in planning the research methodology and co-wrote the paper.

Behbood, Vahid: (Research Colleague)
Contributed in the data pre-processing, provided feedback in planning the research methodology and co-wrote the paper.

Lister, Raymond: (Principal Supervisor)
Provided feedback in planning the research methodology and co-wrote the paper.

Chapter 8: Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. [ITiCSE 2016]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor. Planned the research methodology. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper. Took the lead in writing the paper. Revised the paper for publication with input from reviewers and the principal supervisor. Presented the paper at the conference.

Prior, Julia: (Research Colleague)
Provided the data and proofread the paper.

Behbood, Vahid: (Research Colleague)
Had a previous role in the data preparation step hence was included as an author in the paper.

Lister, Raymond: (Principal Supervisor)
Provided feedback on the manuscript.

Chapter 9: Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success. [SIGCSE 2015]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor. Planned the research methodology. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper. Took the lead in writing the paper. Revised the paper for publication with input from reviewers and the principal supervisor. Presented the paper at the conference.

Behbood, Vahid: (Research Colleague)
Had a previous role in the data preparation step hence was included as an author in the paper.

Vihavainen, Arto: (Research Colleague)
Presented the paper at the conference and proofread the paper.

Prior, Julia: (Research Colleague)
Provided the data and proofread for the paper.

Lister, Raymond: (Principal Supervisor)
Provided feedback on the manuscript.

Chapter 10: Performance and Consistency in Learning to Program. [ACE 2017]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor and the research colleague. Planned the research methodology with the principal supervisor and the research colleague. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper with the principal supervisor. Took the lead in writing the paper with the research colleague. Revised the paper for publication with input from reviewers, principal supervisor and the research colleague.

Lal, Shahil: (Research Colleague)
Presented the paper at the conference and proofread the paper.

Leinonen, Juho: (Research Colleague)
Provided feedback on the manuscript.

Lister, Raymond: (Principal Supervisor)
Provided feedback on the manuscript.

Hellas, Arto: (Research Colleague)
Provided the data and co-wrote the paper. Also provided feedback on the manuscript.

Chapter 11: On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions. [ITiCSE 2016]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor. Planned the research methodology with the principal supervisor. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper with the principal supervisor. Took the lead in writing the paper with the principal supervisor. Revised the paper for publication with input from reviewers and principal supervisor. Presented the paper at the conference.

Vihavainen, Arto: (Research Colleague)
Provided the data and proofread the manuscript.

Lister, Raymond: (Principal Supervisor)
Provided feedback on the manuscript.

Chapter 12: A Contingency Table Derived Methodology for Analyzing Course Data [TOCE 2017]

STATEMENT OF CONTRIBUTIONS OF JOINT AUTHORSHIP

Ahadi, Alireza: (Candidate)
Discussed the research question with the principal supervisor and the research colleague. Planned the research methodology with the principal supervisor and the research colleague. Conducted the data collation. Analyzed the data under the principal supervisor's guidance. Wrote the outline of the paper with the principal supervisor and the research colleague. Took the lead in writing the paper with the research colleague. Revised the paper for publication with input from reviewers and principal supervisor.

Lister, Raymond: (Principal Supervisor)
Provided feedback on the manuscript.

Hellas, Arto: (Research Colleague)
Provided the data and co-wrote the manuscript.

Appendix C

Complete list of Publications by Candidate

1. Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A. and Lister, R., 2012. Using neo-Piagetian theory, formative in-Class tests and think alouds to better understand student thinking: a preliminary report on computer programming. In Profession of Engineering Education: Advancing Teaching, Research and Careers: 23rd Annual Conference of the Australasian Association for Engineering Education 2012, The (p. 772). Engineers Australia.

2. Corney, M., Teague, D., Ahadi, A. and Lister, R., 2012, January. Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123 (pp. 77-86). Australian Computer Society, Inc..

3. Teague, D., Corney, M., Ahadi, A. and Lister, R., 2012, January. Swapping as the Hello World of relational reasoning: replications, reflections and extensions. In Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123 (pp. 87-94). Australian Computer Society, Inc..

4. Teague, D., Corney, M., Ahadi, A. and Lister, R., 2013, January. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136 (pp. 87-95). Australian Computer Society, Inc..

5. *Ahadi, A. and Lister, R., 2013, August. Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?. In Proceedings of the ninth annual international ACM conference on International computing education research (pp. 123-128). ACM.

6. Ahadi, A., Lister, R. and Teague, D., 2014. Falling behind early and staying behind when learning to program. In Proceedings of the 25th Psychology of Programming Conference, PPIG (Vol. 14).

7. Teague, D., Lister, R. and Ahadi, A., 2015, January. Mired in the Web: Vignettes from Charlotte and Other Novice Programmers. In Proceedings of the 17th Australasian Computing Education Conference (ACE 2015) (Vol. 27, p. 30).

8. *Ahadi, A., Prior, J., Behbood, V. and Lister, R., 2015, June. A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (pp. 201-206). ACM.

9. Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S.H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K. and Rubio, M.Á., 2015, July. Educational data mining and learning analytics in programming: Literature review and case studies. In Proceedings of the 2015 ITiCSE on Working Group Reports (pp. 41-63). ACM.

10. *Ahadi, A., Lister, R., Haapala, H. and Vihavainen, A., 2015, July. Exploring machine learning methods to automatically identify students in need of assistance. In Proceedings of the eleventh annual International Conference on International Computing Education Research (pp. 121-130). ACM.

11. Ahadi, A., Applying Educational Data Mining to the Study of the Novice Programmer, within a Neo-Piagetian Theoretical Perspective.

12. *Ahadi, A., Behbood, V., Vihavainen, A., Prior, J. and Lister, R., 2016, February. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (pp. 401-406). ACM.

13. *Ahadi, A., Prior, J., Behbood, V. and Lister, R., 2016, July. Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 272-277). ACM.

14. Leinonen, J., Longi, K., Klami, A., Ahadi, A. and Vihavainen, A., 2016, July. Typing Patterns and Authentication in Practical Programming Exams. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 160-165). ACM.

15. *Ahadi, A., Lister, R. and Vihavainen, A., 2016, July. On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 218-223). ACM.

16. Ahadi, A., 2016, August. Early Identification of Novice Programmers' Challenges in Coding Using Machine Learning Techniques. In Proceedings of the 2016 ACM Conference on International Computing Education Research (pp. 263-264). ACM.

17. *Ahadi, A., Hellas, A., Lister, R.A.Y.M.O.N.D. and Leinonen, J., 2017, January. Performance and Consistency in Learning to Program. In Australasian Computing Education Conference, pp. 11 – 16. *Won the award for Best Student Paper*.

18. Castro-Wunsch, K., Ahadi, A. and Peterson, A., 2017, March. Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. In SIGCSE technical symposium on computer science education.

19. *Ahadi, A., Lister, R. and Hellas, A., 2017. A Contingency Table Derived Methodology for Analyzing Course Data. ACM Transactions on Computing Education.

* Publications forming part of this PhD thesis.

Appendix D

UTS Human Ethics Approval Certificate - UTS HREC - ETH16-0340

ResearchMaster

Human Ethics Application

Application ID :	ETH16-0340
Application Title :	Analysing students' programming code in a data mining framework to understand their learning challenges
Date of Submission :	N/A
Primary Investigator :	Dr Julia Ruth Prior
Other Investigators :	A/Prof Raymond Francis Lister
	Mr Alireza Ahadi

Section 1: Ethics Portal

Select your application type

What type of application are you looking for?
Please **do not** change your application type without first consulting with the Ethics Secretariat (9514 9772).

- [✓ New application (including scope-checking for nil/negligible risk research)]
- Ratification of existing approval
- Transfer of existing approval
- Evaluation of teaching and learning activities
- Amendment to existing approval
- Program approval

You have selected 'new application (including scope checking for nil/negligible risk research)'.
This option allows you to create a new form and will check if you
application can be approved by the Faculty or whether it requires full ethics approval by the HREC. Please click save before continuing.

What should I know before I start?

You have been redirected to the Nil/Negligible Risk Declaration Form.

Would you like more information on:

- This system
- The ethics process
- Purpose of the ethics review process

This question is not answered.

Section 1A: Risk evalution

Determining the level of risk

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
Further examples and information to help you successfully complete your application can be found here

Please refer to the UTS HREC criteria for determining level of risk
for assistance in determining the level of risk.

Please answer each question carefully and thoughtfully.
If you need to contact the Research Ethics Officer you can call (02) 9514 9772 or you can email
the Research Ethics Officer

Does your research involve:

Collecting identifying information from participants*

Yes [✓ No]

Direct interaction between researcher/s and participants*

Yes [✓ No]

Any significant alternation to the routine care or service provided to participants*

Yes [✓ No]

Any risks for participants beyond that experienced in their everyday activities*

Yes [✓ No]

Participation by a member of any vulnerable group, other than incidental REF NS Chapter 4 *

Yes [✓ No]

Randomisation or the use of a control group or a placebo*

Yes [✓ No]

Infringing the rights, privacy or professional reputation of participants*

Yes [✓ No]

Access or establishing a register or database which will be maintained after the completion of the research*

Yes [✓ No]

Do you consider your research to be nil/negligible risk?*

[✓ Yes] No

Please explain why your consider your research to be negligible risk (4000 character limit)*

For the following reasons, we consider our application to be of type nil/negligible risk.
Students' generated code does NOT include any identifying information about them. That is, given the programming code of a student, one would not be able to identify that student. This data will NOT be used for nothing but research purposes, and only and only by me and my supervisor Assoc. Prof. Raymond Lister. Neither data nor any statistical report will be given to any other researcher, university or organisation for whatsoever the reason (Data stays with us at UTS). Deidentifying students IDs and data is performed using data encryption techniques so no one can possibly even read the data as it won't be stored in human language.

The system has assessed your research as being nil/negligible risk research. The nil/negligible risk process is split into two parts; checking that your research is negligible risk by the system, and completing the Nil/Negligible risk form for endorsement by your Faculty. You will now need to complete some basic project information (such as project title and personnel) and follow the "Next Steps" instructions at the end of this form to continue to the nil/negligible risk declaration form.

Please continue to the next page

Section 2: Project information

Project title

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

Declaration of Nil/Negligible Risk Research Form

Project Title*

Analysing students' programming code in a data mining framework to understand their learning challenges

Please note that the HREC is now granting a standard approval period for the research proposals.
The approval period for your project will be specified in your approval letter.
Please also note that research should not commence until ethics approval has been granted. The Committee cannot grant retrospective approval for data that has already been collected.

Ethics category code*

Human

Is this a pilot study?*

[✓ Yes] No

If this is a pilot study or 'scoping' study, you may need to apply for full HREC approval at a later date if your methodology is no longer negligible risk

Please save and continue to the next page

Project personnel

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

Are there external or non-UTS investigators or personnel listed on this application?

○ Yes
[✓ No]

Is this a student project for a current degree course?*

[✓ Yes]
○ No

NOTE: Under 'Position', additional Chief Investigators should be listed as '1Chief Investigator', Co-Investigators should be listed as '3Assoc. Investigator' and students should be listed as '5Research Student. Further options are available for Research/Project Managers and Administrators. The main contact should be marked as 'primary'

If any details are incorrect or missing please contact the Ethics Secretariat on (02) 9514 9772 or by email.

How to add a person to the personnel table:
1. Click on 'More criteria' which is located on the top right hand corner of the table below
2. Enter the surname (and given name if the surname is common) in the fields marked 'Surname' and 'Given name' and click 'Search'
3. Click on the name of the person you wish to add
4. If they are the primary contact (e.g. Chief Investigator/Supervisor), tick "Yes" under 'Primary contact'
5. Select the position from the drop-down list (e.g. Chief Investigator/Research Student)
6. Click on the green tick

Internal personnel listed on this ethics protocol:
*

1	Primary	No
	ID	010292
	Surname	Lister
	Given Name	Raymond
	Name	A/Prof Raymond Francis Lister
	Position	Chief Investigator
	Type	Internal
	AOU	FEIT.School of Software
	Managing Unit	Faculty of Engineering & Information Technology
	Email Address	Raymond.Lister@uts.edu.au
	Contact Phone	1850
2	Primary	Yes
	ID	020304
	Surname	Prior
	Given Name	Julia
	Name	Dr Julia Ruth Prior
	Position	3Assoc. Investigator
	Type	Internal
	AOU	FEIT.Associate Dean (Teaching & Learning)
	Managing Unit	Faculty of Engineering & Information Technology
	Email Address	Julia.Prior@uts.edu.au
	Contact Phone	4480
3	Primary	No
	ID	10973752
	Surname	Ahadi
	Given Name	Alireza
	Name	Mr Alireza Ahadi
	Position	5Research Student
	Type	Internal
	AOU	FEIT.Faculty of Engineering & Information Technology
	Managing Unit	Faculty of Engineering & Information Technology
	Email Address	Alireza.Ahadi@student.uts.edu.au
	Contact Phone	

If you cannot find a person through the personnel table(s) above, please enter their details here (title, name, organisation, department, phone number, address, email address and their position on this protocol). If the person you cannot find is EXTENRAL to UTS please contact the Ethics Secretariat
(4000 character limit)

Associate Professor Raymond Lister is a full time academic staff at school of software, faculty of engineering and IT. Allures Ahadi is his PhD student with student number 10973752.

Please provide additional (or preferred) contact details of any of the people listed on the project if necessary.

This question is not answered.

Please provide details of any formal qualifications (REF NS 1.1(e)) of each person listed on the project.*

Julia Prior is the Director of Engagement (Teaching and Learning), Associate Dean (Teaching & Learning).
Raymond Lister is an Associate Professor at School of Software

Please outline the experience of each person listed on this project relevant to this application.*

From 2004, Raymond has been one of three conveners of the BRACElet project, which uses an action research model to study the performance of students on programming exams, with the aim of better understanding the problems students have with understanding computer programs. In 2007, He was joint winner (with Professor Jenny Edwards) of an $80K Fellowship from the Australian Learning and Teaching Council, which funded work on this project until 2010. In that funding period, 26 project participants, from 14 different educational institutions across seven different countries co-authored 16 research papers. He has pursued this research on the mental development of the novice programmers since 2004, when he led an international ITiCSE working group, comprised of eleven other members from six countries: USA, UK, Sweden, Finland, Denmark, and New Zealand. That research was published in Lister et al. (2004) A Multi-National Study of Reading and Tracing Skills in Novice Programmers. Alireza Ahadi is his research student and has co-authored 14 publications with him in the past 5 years. Julia Prior worked as an analyst/programmer for large corporations in both the gold mining and car manufacturing industries. She also wrote, customised and maintained a number of scientific applications interfacing with electronic bio-mechanical measuring equipment in a research organisation. She has co-authored three research papers with Raymond Lister and Alireza Ahadi in the field of education research.

Primary AOU*

FEIT.Faculty of Engineering & Information Technology

Please save and continue to the next page

Student details

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

Degree being undertaken*

PhD

Have you been successful in your doctoral/masters assessment? *

[✓ Yes]
No

Please save and continue to the next page

Funding details

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
Further examples and information to help you successfully complete your application can be found here

Have you received funding in relation to this research?*

Yes
[✓ No]

Do you intend to apply for funding in the future?*

Yes
[✓ No]

Please continue to the next page

Section 3: Research summary

Description

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

Please provide a brief description of the research design including research questions and proposed methods for conducting the research (approximately 250 words)*

Research Design:
Data will be automatically collected via programming interfaces in two programming subjects at UTS:
Programming Fundamentals (48023)
Database Fundamentals (31271)
There will be no interaction with students at all.
Collected data will be de-identified through encryption techniques and will be ready for analysis.
A quantitative research methodology will be used for data analysis, depending on the volume, type, data repository type and available resources.
Data analysis will be performed using SPSS software, and statistical packages in R programming language.

What are the hypotheses/goals/aims/objectives of your research?
Please include a brief description using plain English explaining your research aims (approximately 100 words)
*

Hypothesis: "It is possible to identify at risk of failing students at early stages of the semester."
Aim of this project is to investigate students' coding snapshots to see if their learning challenges could be identified through data analysis.
Objectives of the project:
Finding learning pathway/patterns when students code in Java programming language.
Finding learning pathway/patterns when students code in SQL database language.
Comparing the result of the two to possibly draw widespread conclusions.

What do you hope the outcome(s) of this research will be?*

More insights into students psychology of learning.

Who do you think will benefit from this research?*

Students, education researchers, and programming teacher.

Please save and continue to the next page

Additional details

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

Will you be involving an external institution, organisation or community group?*

Yes [✓ No]

Will this research gather information from participants? (E.g. surveys, questionnaires, interviews, etc)*

Yes [✓ No]

Will you be obtaining consent from participants?*

Yes [✓ No]

Please continue to the next page

Section 4: Checklist

Attachments

You can save your application at any time by clicking on the save button on the right hand side in the toolbar.
For further information and help in completing your application go to Ask Research

An attachment table will appear once you have answered yes to any of the questions below.

I have attached the following supporting documents

Doctoral or Masters assessment*

[✓ Yes] N/A

Participant Information Sheet(s)*

Yes [✓ No]

Informed Consent Form *

Yes [✓ No]

Please explain why any of the above have not been attached (either softcopy/hardcopy) and if applicable, when they will be provided.
*

We don't deal with individuals as there will not be any direct interaction or a need for personal information from students.

Documents attached to this application
NOTE: If you are only attaching a hardcopy of any attachments relating to this application, you must still click on 'Additional Attachments' on the right hand side of the table. Click on the help button for more details.
*

Description	Reference	Soft copy	Hard copy
DSP.	DSP2.doc	✓	

I understand that I need to submit this application form to my Faculty Research Office*

[✓ Yes] No

Please save and continue to the next page

Declaration

Declaration Signoff

I declare that the information I have given above is true and that this research does not contravene the National Statement on Ethical Conduct in Human Research, the Australian Code for the Responsible Conduct of Research, and relevant UTS policy and guidelines relating to the safe and ethical conduct of research.

I also declare that I will respect the personality, rights, wishes, beliefs, consent and freedom of the individual participant in the conduct of my research and that I will notify the UTS Human Research Ethics Committee of any ethically relevant variation in this research.

In signing this declaration, I confirm that this form has been distributed to each member of the research team, and they have agreed to abide by the principles and processes of the research as outlined in this form.

I also declare that I believe this research to be nile/negligible risk for the reasons outlined in this form, and that my research does not require approval from the UTS Human Research Ethics Committee

Declaration Signoff*

1	Full Name	A/Prof Raymond Francis Lister
	Position	Chief Investigator
	Declaration signed?	No
	Signoff Date	
2	Full Name	Mr Alireza Ahadi
	Position	5Research Student
	Declaration signed?	~~No~~ Yes
	Signoff Date	

[illegible] A.A.

Please note that the Faculty cannot review your Declaration of nil/negligible risk research without first having received a signed hardcopy
I have obtained and attached signatures to this declaration form:

[✓ Yes] N/A

Faculty review

Please note that this section should be signed off by the Faculty after being printed.
Please continue to the next page to electronically submit your application.
You can submit the hardcopy of this form after submitting it electronically

Faculty review - Associate Dean (Research) or nominee

I am aware that this research is being conducted within this faculty and am satisfied that the researchers have met faculty requirements in relation to this research, and that this research is low or negligible risk.

Signature of Associate Dean (Research) or nominee (Please sign once printed):

Jie Lu [illegible]

This question is not answered.

Date:

31/3/16

This question is not answered.

I do/do not* wish to add comments in relation to this application (*please circle which one when signing the hard copy):

This question is not answered.

How do I submit this form?

Submission instructions

Your application has been assessed by the system as being nil/negligible risk and can be submitted to your Faculty for review.

Please ensure that you print out and sign the form and submit.

All questions marked with a red asterisk (*) must be answered before submitting this form. Please check that all pages in the form menu (located on the left of this page) have a green tick.
Pages marked with a (!) indicate that one or more mandatory questions have not been answered.

To electronically submit your application
Please click on the `save' button in the toolbar before clicking on the Action tab which is located on the left of this page next to the form tab.
Click on 'Submitunder the action tab to submit your application.
If you are a student, this form will automatically go to your supervisor for review. Your supervisor will have two options:

1. Submit to the Faculty or
2. Request further information or changes from you prior to submitting to the Faculty.

You will receive an email notification if your supervisor has requested further information or changes, or whether your application has been submitted to the Ethics Secretariat.

To physically submit your application
The Faculty cannot review this form without first having received a one signed hardcopy. Please submit the hardcopy to your Faculty for sign off by the Associate Dean Research (ADR) or nominee.
You can contact the Ethics Secretariat if you are unsure of who to submit this form to.

Please ensure you save a copy of this form before submitting to the Faculty as
access to this form will be restricted until after the Faculty has reviewed your declaration.

To save and print your application as a PDF
To save and print your applicatio

What happens next?

What happen's next?

Have you printed, signed and submitted a copy of this form for your supervisor and Faculty (ADR) to review?

[✓ Yes]

No

This question is not answered.

Once you click on submit, this form will be electronically submitted to your supervisor for review before going to the Faculty for review.
If your supervisor has any comments, they will be able to either make comments on your application or discuss their comments with you outside of the system.
Your supervisor will need to electronically send the application back to you for editing through the options listed in the action tab.
Once your supervisor electronically endorses your application it will available for the Faculty to review.
You must provide a signed hardcopy for the Faculty before they can electronically endorse your application.

If you would like any further information please see our instructions or contact the Ethics Secretariat by emailor on (02) 9514 9645.
You can also watch videos on how to submit this form or download detailed instructions about the form and its features.

Faculty Research Office only

Faculty Research Office check

Faculty Research Office only
Has this application been printed, signed and a hardcopy received by the Faculty Research Office?

○ Yes

○ No

This question is not answered.

Comments/Notes:

This question is not answered.

Appendix E

Extract from UTS Subject Outline – 31271 "Database Fundamentals" Sem. 2 2016

SUBJECT OUTLINE

31271 Database Fundamentals

Course area	UTS: Information Technology
Delivery	Spring 2016; City
Credit points	6cp
Requisite(s)	31267 Programming Fundamentals OR 48023 Programming Fundamentals These requisites may not apply to students in certain courses. There are also course requisites for this subject. See access conditions.
Result type	Grade and marks

Recommended studies: it is assumed that students are familiar with basic system analysis concepts and have basic software skills

Subject coordinator

Dr Vahid Behbood
Office: CB11.07.205
Phone: 9514 2263
Email: vahid.behbood@uts.edu.au

Email is the communication medium of last resort, as it is a particularly inefficient way for students to communicate with a subject co-ordinator who teaches hundreds of students often across more than one subject. Therefore, please do not communicate with the subject co-ordinator and teaching staff via email unless it is a personal matter.

Please always include the SUBJECT NUMBER and your STUDENT NUMBER and FULL NAME in any email sent to the teaching staff. Students' emails that do not adhere to this protocol will be ignored. Further, please note that emails sent from a student to the teaching staff must be sent from the student's UTS email address - it is University protocol that staff will not respond to email from, or send emails to, any other email addresses for currently enrolled students.

Except for urgent personal matters, voice mail will not normally be returned.

Teaching staff

Laurie Benkovich is the lecturer and head tutor for this subject.
Emailed queries about your studies in this subject should be sent to < vahid.behbood@uts.edu.au>

Asking questions is one of the most productive ways to learn; they are welcome, under appropriate circumstances. Questions may be asked in lectures, but class size may preclude a full answer. Questions are particularly welcome in tutorials, lab sessions and in the discussion forums of UTSOnline, where the answer can be shared by the entire class. Whilst online postings will not necessarily be responded to immediately, the teaching staff aims for all postings to be satisfactorily responded to (by other students or the teaching staff) within three to four working days.

Face-to-face communication is the preferred way of communication with the academic staff. Most questions are not so urgent that they cannot wait until the next consultation time or class session. Students are strongly encouraged to take advantage of the staff's consultation times to discuss subject issues.
Tutors' consulting times, preferred contact details etc will be available on the UTSonline Contacts page.

Email should be the communication medium of last resort, unless it is a personal matter or to organise a face-to-face appointment.

Please always include the SUBJECT NUMBER and your STUDENT NUMBER and FULL NAME in any email sent to the teaching staff. Students' emails that do not adhere to this protocol will be ignored. Further, please note that emails sent from a student to the teaching staff must be sent from the student's UTS email address - it is University protocol that staff will not respond to email from, or send emails to, any other email addresses for currently enrolled students.

Please note that the email protocol specified above apply to email communications to all teaching staff in the subject.

Subject description

This subject introduces students to the fundamentals of effective database systems. Students are taught how data is structured and managed in an organisation in a way that can be used effectively by applications and users. They also learn to use the language SQL for effective data retrieval and modification. This subject teaches students to appreciate the significance and challenges of good database design and management, which underpin the development of functional software applications.

Subject learning objectives (SLOs)

Upon successful completion of this subject students should be able to:

1. Explain the main issues related to the design and use of structured data.
2. Construct conceptual and logical data models applying database design principles.
3. Evaluate data redundancy levels and their impact on database integrity and maintainability.
4. Construct conceptual data models applying data modeling principles.
5. Construct logical data models adhering to data normalisation principles.
6. Distinguish between good and bad database design.
7. Construct efficient SQL queries to retrieve and manipulate data as required.

Graduate attributes

This subject also contributes specifically to the development of the following Course Intended Learning Outcomes (CILOs):

- B1. Identify and apply relevant problem solving methodologies (B.1)
- B2. Design components, systems and/or processes to meet required specifications (B.2)
- B3. Synthesise alternative/innovative solutions, concepts and procedures (B.3)
- B5. Implement and test solutions (B.5)
- C1. Apply abstraction, mathematics and/or discipline fundamentals to analysis, design and operation (C.1)
- C2. Develop models using appropriate tools such as computer software, laboratory equipment and other devices (C.2)
- C3. Evaluate model applicability, accuracy and limitations (C.3)
- E1. Communicate effectively in ways appropriate to the discipline, audience and purpose. (E.1)
- F1. Be able to conduct critical self-review and performance evaluation against appropriate criteria as a primary means of tracking personal development needs and achievements (F.1)

Teaching and learning strategies

Lectures and tutorial sessions totaling 3 hours per week, plus some optional, drop-in laboratory sessions during the semester.

Content (topics)

1. The Role of Databases in Information Systems
2. The Relational Data Model
3. SQL: simple queries, aggregate functions, data modification statements, simple joins, complex joins, subqueries and set operators
4. Conceptual database design (ER modeling)
5. Logical design: ER conversion to a relational model
6. Normalisation

Program

Week/Session	Dates	Description

1	1 Aug	Lecture 1: Introduction to Database Management Systems (DBMS) and Data Modeling I Tutorial 1 – Introduction to DBMS and Data Modeling I
2	8 Aug	Lecture 2: Data modeling II Tutorial 2: Data Modeling II
3	15 Aug	Lecture 3: Data Modeling III Tutorial 3: Data Modeling III **Notes:** *Assessment task due: **Assignment** Part A Case Study*
4	22 Aug	Lecture 4: Normalisation I Tutorial 4: Normalisation I
5	29 Aug	Lecture 5: Normalisation II Tutorial 5: Normalisation II **Notes:** *Assessment task due: **Assignment** Part B ERD*
6	5 Sept	Lecture 6: SQL I Tutorial 6: SQL I **Notes:** *Drop-in SQL labs*
	12 Sept	Review Week: No Lecture
7	19 Sept	Lecture 7: SQL II Tutorial 7: SQL II **Notes:** *Assessment task due: **Assignment** Part C Relational Model and Normalisation* *Drop-in SQL labs*

8	26 Sept	Lecture 8: SQL III Tutorial 8: SQL III **Notes:** *Drop-in SQL labs*
9	3 Oct	Lecture 9: SQL IV Tutorial 9: SQL IV **Notes:** *Drop-in SQL labs*
10	10 Oct	Lecture 10: Subject Review
11	17 Oct	***Online SQL Test*** in faculty's computer labs during normal tutorial session times.

Student Attendance: The Faculty of Engineering and Information Technology expects that students will attend all scheduled sessions for a subject in which they are enrolled.

Additional information

U:PASS Program

This subject participates the UTS Peer Assisted Study Success (U:PASS) this semester. You are encouraged to sign in this program.

U:PASS is a voluntary “study session” where you will be studying the subject with other students in a group. It is led by a student who has previously achieved a distinction or high distinction in that subject, and who has a good WAM. The leader will typically prepare questions for you to work on, or if you have specific questions or things you’re not clear on, you can bring them along, and the leader will get the group to work on that. It’s really relaxed, friendly, and informal. Because the leader is a student just like you, they understand what it’s like to study the subject and how to do well, and they can pass those tips along to you. Students also say it’s a great way to meet new people and a “guaranteed study hour”.

You can sign up for U:PASS sessions in My Student Admin https://onestopadmin.uts.edu.au/. You’ll find it listed in the area where you sign up for lectures, tutorials, etc. Note that sign up is not open until week 1, as it’s voluntary and only students who want to go should sign up.

Note that you don’t have to be struggling in the subject to attend U:PASS – frequently students who are already doing well will do even better after attending U:PASS.

If you have any questions or concerns about U:PASS, please contact Georgina at upass@uts.edu.au, or check out the website: http://www.ssu.uts.edu.au/peerlearning/index.html

Additional subject costs

Students are strongly advised to purchase the customised, prescribed textbook for the subject, see details under 'Required texts'.

Assessment

Minimum Requirements

You should submit the online SQL test and the assignment for assessment during the semester, and you must do the final examination.

It is compulsory to pass the final examination in order to pass the subject, i.e. you must gain a minimum of 40% of the total possible exam mark. The Faculty’s policy is to award an X grade as the student's final result for the subject, or the student’s actual mark, whichever is the lower, for failure of compulsory assessment items.
Thus, you are not required to pass the assignment, the SQL test, or the tutorial quizzes, in order to pass the subject, but you do have to pass the final examination in order to pass the subject. (N.B. it will be difficult for you to pass the final exam if you have not taken the SQL test or attempted the assignment and tutorial quizzes, however).
If you pass the final exam, i.e. score 40% or more for it, your final subject mark will be the highest of:

- the sum of your weighted tutorial quiz mark, SQL test result, assignment result and exam result;
- your exam result alone i.e. the exam result weighted 100%

If you fail the final exam, i.e. score less than 40%, your final subject mark will be the lowest of:

- the sum of your weighted tutorial quizzes result, SQL test result, assignment result and exam result;
- an X grade

It is very important to note that, even if you pass the final examination, you must still gain a final Subject mark of at least 50% in order to pass the subject -- either your final exam mark as a percentage must be 50% or more, or your weighted mark marks of exam, assignment, tutorial quizzes and SQL test together must be 50 or more. Simply scoring a mark between 40% and 49% in the exam will not be enough to gain a pass in the subject; in this case, the student’s weighted mark must be 50 or more in order to pass.

Assessment task 1: SQL Online Test

Intent: The online test assesses the student's practical ability to construct appropriate SQL statements to retrieve particular information from the database.

Objective(s): This assessment task addresses the following subject learning objectives (SLOs):

1 and 7

This assessment task contributes to the development of the following course intended learning outcomes (CILOs):

B.1, B.2, B.3 and B.5

Type: Quiz/test

Groupwork: Individual

Weight: 20%

Task: The test will be taken online under supervised conditions in the faculty's computer labs. In a limited time period, students will be required to design and execute SQL queries that correctly return specific information from a database.

Due: Friday 21 October 2016
See also Further information.

Criteria linkages:

Criteria	Weight (%)	SLOs	CILOs
Functionality of design	20	7	B.5
Correction of ‘interpretation’ of the problem	20	7	B.1, B.2
Validity of solution	20	1, 7	B.5
Application of analysis and construct	20	7	B.3
Appropriate choice of construct	20	7	B.2

SLOs: subject learning objectives
CILOs: course intended learning outcomes

Further information: Students will sit for the SQL test in **Week 11,Friday 21th October 2016,** during special laboratory sessions as timetabled for each tutorial class group in the faculty's computer laboratories.

The test will be taken online. You will be given ample opportunity to practice doing tests with the online software in the weeks prior to the formal, marked test. The practice online test is accessible via the Internet and a web browser and can be done any time at the student's convenience, as often as they wish.

More details about the test will be released separately.

NOTE: it is very important that you practice using SQL regularly as much as possible before the test date. You will not pass the SQL test if you do not invest adequate time and effort of your own into doing sufficient practical SQL exercises beforehand**.**

Assessment task 2: Data Modeling and Database Design Assignment

Intent: The assignment assesses the student's ability to analyse and interpret data requirements, and to create conceptual and logical designs for a suitable database by applying the principles of data modeling and data normalisation.

Objective(s): This assessment task addresses the following subject learning objectives (SLOs):

1, 2, 3, 4, 5, 6 and 7

This assessment task contributes to the development of the following course intended learning outcomes (CILOs):

B.1, B.2, B.3, B.5, C.1, C.2, C.3, E.1 and F.1

Type: Design/drawing/plan/sketch

Groupwork: Group, group assessed

Weight: 20%

Task: The assignment involves the conceptual and logical design of a database to support a small- to medium-size enterprise or organisational unit. The task can be done individually, in pairs or as part of a group.

Due: Week 3 to Week 7
There are 3 due dates for the assignment deliverables: Part A is due Friday 19th August 2016, Part B is due Friday 2nd September 2016, and Part C is due Friday 23th September 2016.
See also Further information.

Criteria linkages:

Criteria	Weight (%)	SLOs	CILOs
Correctness of notation and design drawing	7	1, 2, 3, 4, 5	B.2, E.1
Correctness of application	7	1, 2, 3, 4, 5	B.2, E.1
Appropriateness of design solution in relation to the context	7	1, 2, 3, 4, 5, 6	B.5
Functionality of design	7	7	B.5
Correction of 'interpretation' of the problem	7	7	B.1, B.2

Validity of solution	7	1, 7	B.5
Application of analysis and construct	7	7	B.3
Appropriate choice of construct	7	7	B.2
Active participation in self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Accuracy of self-review in self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Justification of self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Identification of appropriate data elements and their relationship between them	7	2, 3, 4, 5, 6	C.1
Application of analysis and modelling construct	7	2, 3, 4, 5	B.3, C.2
Validity of solution	7	1, 2, 3, 4, 5, 6, 7	B.5, C.3
Validity of justification for the design	2	1, 6	B.5, C.3

SLOs: subject learning objectives
CILOs: course intended learning outcomes

Further information: An electronic copy of each of the assignment parts must also be submitted to the subject's UTSonline Turnitin Assignment module by the due dates and times. Please note that only an assignment deliverable that has both a hard copy and an electronic copy submitted to the correct places by the due date will be marked.

Assignments may be done individually, in pairs or in groups of a maximum of 3 students. If you decide to do it in pairs or groups, you are advised to find assignment partner(s) as soon as possible in the semester—do not wait until the first assignment part is almost due. You are also advised to find partner(s) who are aiming for the same grade, e.g. if you are aiming for a D, you should find a partner who is also aiming for a D, not someone who realistically is only aiming for a P.

The assignment will be marked according to the same criteria regardless of whether work is done by an individual or a group. The marking criteria will be included with the assignment. Students will be asked to assess the contribution of their peers to the group project. This peer assessment will be used to determine whether all group members receive the same mark for the assignment. If discrepancies in contribution are noted, individual marks will be scaled in accordance with the peer feedback.

Late assignment submissions will incur a penalty of 10% of the student's maximum possible mark for the required deliverable per day or part thereof that they are overdue. Assignments submitted more than 5 working days late will not be marked.

Submission and return details will be included in the assignment specification.

Special consideration, for late submission, must be arranged well before the due dates with the subject co-ordinator.

Assessment task 3: Final Examination

Intent: The final exam assesses the student's level of attainment of all of the subject objectives. The student's appreciation of good database design and management principles and the effects of poorly designed database on database integrity and maintainability will be assessed, as well as their understanding and practical application of the relational database model, conceptual and logical database design principles and SQL for effective data retrieval, management and modification.

Objective(s): This assessment task addresses the following subject learning objectives (SLOs):

1, 2, 3, 4, 5, 6 and 7

This assessment task contributes to the development of the following course intended learning outcomes (CILOs):

B.1, B.2, B.3, B.5, C.1, C.2, C.3, E.1 and F.1

Type: Examination

Groupwork: Individual

Weight: 50%

Task: The final examination will be held in the usual, university formal examination time.
It is comprised entirely of multiple-choice questions, which the student must answer on a special OCR answer sheet.
It is a 2-hour closed-book examination.

Criteria linkages:

Criteria	Weight (%)	SLOs	CILOs
Correctness of notation and design drawing	7	1, 2, 3, 4, 5	B.2, E.1
Correctness of application	7	1, 2, 3, 4, 5	B.2, E.1
Appropriateness of design solution in relation to the context	7	1, 2, 3, 4, 5, 6	B.5
Functionality of design	7	7	B.5
Correction of ‘interpretation’ of the problem	7	7	B.1, B.2
Validity of solution	7	1, 7	B.5
Application of analysis and construct	7	7	B.3
Appropriate choice of construct	7	7	B.2
Active participation in self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Accuracy of self-review in self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Justification of self-assessment	7	1, 2, 3, 4, 5, 6	F.1
Identification of appropriate data elements and their relationship between them	7	2, 3, 4, 5, 6	C.1
Application of analysis and modelling construct	7	2, 3, 4, 5	B.3, C.2
Validity of solution	7	1, 2, 3, 4, 5, 6, 7	B.5, C.3
Validity of justification for the design	2	1, 6	B.5, C.3

SLOs: subject learning objectives
CILOs: course intended learning outcomes

Further information: Students should not think that the final exam will be relatively easy because it is all multiple-choice questions. Answering the questions correctly will require students to have a solid understanding of the concepts, including being able to apply them in a very practical way. Mastering these skills and knowledge requires students to work on and practice them consistently throughout the semester. Students who have done this will generally be able to tackle the exam with confidence.
Although it is in a different format, the exam assessment is strongly aligned with the lecture and tutorial material, the textbook content and exercises, the online SQL test and the assignment.

It is compulsory to pass the examination in order to pass the subject, i.e. you must gain a minimum of 40% of the total possible exam mark. The faculty policy is to award a X grade as the final result or the student's actual mark, whichever is the lower, for failure of compulsory assessment items. However, please note that a 40% mark in the exam is not enough to pass the subject, as a student's final mark must also be 50 or greater in order to gain a Pass grade.

NB: Under the University's Assessment Policy, no supplementary examination is required in this subject, and none is offered.

Assessment task 4: Weekly Tutorial Quiz

Intent: To assess the individual student's understanding of one to two specified concepts covered in the previous week's lecture topic.

Objective(s): This assessment task addresses the following subject learning objectives (SLOs):

1, 2, 3, 4, 5, 6 and 7

Type: Quiz/test

Groupwork: Individual

Weight: 10%

Task: Every tutorial session will include a short written quiz that will test the individual student's knowledge of two concepts covered in the Review Questions section of that week's tutorial (previous week's lecture). Before the class, students are expected to complete the Review Questions section, which relate to the previous week's lecture topic.

Due: In class in
See also Further information.

Criteria linkages:

Criteria	Weight (%)	SLOs	CILOs
Correct recall, identification or brief definition of one piece of specific information in the current topic	50	1, 2, 3, 4, 5, 6, 7	
Correct explanation of a specific concept in the current topic	50	1, 2, 3, 4, 5, 6, 7	

SLOs: subject learning objectives
CILOs: course intended learning outcomes

Further information: Each quiz will be marked during the tutorial session.

Students may score 0, 1 or 2 for each quiz. These weekly scores will be added together to give the student's total quiz mark, out of a possible total of 20 (as there are ten tutorial sessions).
The total quiz mark is weighted 10% of the student's final weighted mark.

No Special Consideration requests will be granted for these quizzes

Use of plagiarism detection software

An electronic copy of the assignment must be submitted to the subject's UTSonline Turnitin Assignment module.

Minimum requirements

It is compulsory to pass the examination in order to pass the subject, i.e. you must gain a minimum of 40% of the total possible exam mark

The Faculty's policy is to award an X grade as the final result, or the student's actual mark, whichever is the lower, for failure of compulsory assessment items.

NB: Under the University's Assessment Policy no supplementary examination is required in this subject, and none is offered.

Required texts

To reduce the cost of the prescribed textbook for students, a **custom book** for this subject has been compiled, which includes only the six chapters from the original Hoffer book Modern Database Management (details below) that are used in the subject (chapters 1-4, 6 and the first part of 7). The custom book has a customised cover stating that it is a custom book based on this Hoffer title and 'compiled by Julia Prior'. It is available for purchase in the Co-Op Bookshop.

Students who prefer to purchase the full original Hoffer title (new or second-hand) will of course be able to use this as their prescribed textbook. The tenth edition is also acceptable, although the relevant page numbers may be different.

It is expected that every student has their own copy of the prescribed textbook, which is not simply a reference book for the subject. The content of the subject is based heavily on the contents of this textbook, which is the primary resource for the subject.

Modern Database Management, 11th Edition, by Hoffer, J.A, Ramesh, V., and Topi, H.. (2011), ISBN-10: 0273779281 or ISBN-13: 9780273779285, published by Pearson Education.

Recommended texts

Mannino, Michael V (University of Colorado, Denver).: *Database Design, Application, Development & Administration*, 2nd, 3rd (McGraw-Hill publishers) and 4th editions are all acceptable; the relevant material in all of these editions is essentially the same in content, it is mostly chapter and page numbers that may be different. There are several copies of the 4th edition available in 7-day loans section, open reserve and on the open shelves, as well as a number of copies of the 2nd edition in 7-day loans, closed reserve and on the open shelves, in the university's City campus library in Haymarket.
Date, C.J.: *An introduction to Database Systems*, Eighth Edition, Pearson Addison Wesley, 2003, ISBN: 0321197844 (there are several copies of this in the City campus library, and other books by the same author). Whilst this book is quite technical, this is the seminal text for relational database management systems.
Simsion, Graeme C. : *Data Modeling Essentials,* Morgan Kaufmann Publishers, Amsterdam, 2005, ISBN: 0126445516 (there is also a later edition, but this is not in the uni library).
Pratt, **Philip J.** : *A Guide to SQL,* Seventh Edition, Thomson Course Technology, 2005, ISBN 0619216743 (earlier editions also fine).
Connolly, Thomas M.: *Database solutions : a step-by-step guide to building databases*, Second Edition, Pearson Addison Wesley, 2003, ISBN 0321173503.
Connolly, Thomas and Begg, Carolyn: *Database Systems – a practical approach to design, implementation, and management,* Fifth Edition, AddisonWesley, 2010, ISBN 0321523067 (third and fourth editions also fine).

These texts, and several others on relational database design, use and management, are all available in the university library.

References

Useful web references include:
http://www.sqlcourse.com/
http://www.w3schools.com/sql/default.asp
http://sqlzoo.net/

http://wiki.postgresql.org/

Additional references and reading material will be handed out, recommended during lectures or posted to UTSOnline when necessary during the semester.

Other resources

UTSOnline:

UTSOnline is the web-based online learning and teaching environment used at UTS in a variety of ways to support, complement and extend student learning activities. Subject announcements, links to subject learning materials and other information will be posted on UTSOnline. You may use it for communicating with other course participants and staff, and you should also participate in the online discussion forums related to the subject.

If you are having problems logging on to UTSOnline or forget your password, contact the ITD helpdesk on x2222, email itsc@uts.edu.au, or go to the ITD support counter in Building 2, Level 4. (Please do not contact any of the subject's teaching staff for help with access to UTSOnline).

You should check the announcements on UTSOnline for this subject at least once a week, as all student notices for this subject will be given via this site. The subject coordinator will assume that every student is checking UTSOnline regularly for subject announcements, as well as the discussion forums and subject material.

The URL for UTSOnline is: http://online.uts.edu.au

The UTSonline course id for this subject is 31271 Database Fundamentals (same as the subject id and name).

Lecture Slides: Please note that although electronic copies of all lecture slides are available for students, these are not to be regarded as adequate lecture notes, nor as the complete subject content. They are merely a **guide** to what students need to master in the subject, and for students to use as a basis for making their own notes during lectures.

FEIT Student Guide

For further information regarding your studies please see
my.feit.uts.edu.au/modules/myfeit/downloads/StudentGuide_Online.pdf

Graduate attribute development

For a full list of the faculty's graduate attributes, refer to the Student Guide.

Assessment: faculty procedures and advice

Special Consideration

If you believe your performance in an assessment item or exam has been adversely affected by circumstances beyond your control, such as a serious illness, loss or bereavement, hardship, trauma, or exceptional employment demands, you may be eligible to apply for Special Consideration.

Information about eligibility for special consideration and instructions on how to apply can be found at:

http://www.uts.edu.au/current-students/managing-your-course/classes-and-assessment/special-circumstances/special

Academic liaison officer

Academic Liaison Officers (ALOs) are academic staff in each faculty who assist three groups of students: students with disabilities and ongoing illnesses; students who have difficulties in their studies because of their family commitments (e.g. being a primary carer for small children or a family member with a disability); and students who gained entry through the UTS Educational Access Scheme or Special Admissions.

ALOs are responsible for determining alternative assessment arrangements for students with disabilities. Students who are requesting adjustments to assessment arrangements because of their disability or illness are requested to see a Disability Services Officer in the Special Needs Service before they see their ALO.

The ALO for undergraduate students is:

Chris Wong
telephone +61 2 9514 4501

The ALO for postgraduate students is:

TBA - please refer to the University's ALO contact list for updates

Disclaimer

This outline serves as a supplement to the Faculty of Engineering and Information Technology Student Guide. On all matters not specifically covered in this outline, the requirements specified in the Student Guide apply.

Appendix F

Extract from Helsinki Subject Outline – 581325 "Introduction to Programming" Sem. 2 2016

This chapter reviews the topics and the course structure for the subject Introduction to Programming at University of Helsinki.

Material

Note: each week unlocks after you have been rewarded 85% of the points from the previous week.

- Week 1
  - User input
  - Printing on screen
  - Conditional statements
- Week 2
  - Loops
  - Basics of methods
- Week 3
  - Methods
  - The ArrayList data structure

- Week 4
    - Basics of objects
- Week 5
    - More on objects
- Week 6
    - Tables
    - Sorting
    - Searching

CONTINUE TO PART II

Information Twitter & Facebook

Guidance #mooc.fi @ IRCNet

Email mooc@cs.helsinki.fi

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI
TIETOJENKÄSITTELYTIETEEN LAITOS
INSTITUTIONEN FÖR DATAVETENSKAP
DEPARTMENT OF COMPUTER SCIENCE

581325 Introduction to programming

Principal theme	Prerequisite knowledge	Approaches the learning objectives	Reaches the learning objectives	Deepens the learning objectives
Algorithms and control structures	• No pre-requisites (comprehensive-school mathematics)	• Know and can explain the concept of programming languages, compilation, and interpretation. • Can explain the significance of assignment operations and the sequential execution of algorithms. • Can simulate simple algorithms.	• Can formulate simple algorithms. • Can explain the concept "algorithm state." • Understand how logical expressions are statements on algorithm's state. • Know how to use basic control structures. • Understand the concept of a program that asks for input data and writes output data, and can implement one. • Know the concept of arrays and can program sequential search, binary search, and some way to sort the elements of an array.	• Can understand why a sequential search is a linear operation, a binary search logarithmic, and sorting squared. • Can create programs that are elegant both in their logic and their appearance.
Variables and types	• The concept of algorithms	• Grasp the concept of the type and value of variables.	• Can use variables and write expressions of the types int, double, boolean and String. • Know the difference between simple types and reference types. • Know the significance of assignment compatibility in programming.	• Know some of the history of type categorisation and can assess the consequences of different options.

			• Understand the behaviour of formal parameters and local variables. • Know how to use classes as types. • Know how to index the array.	
Sub-programs	• The concept of algorithms	• Grasp the principles of naming and calling algorithms.	• Can define and call sub-programs, Java methods. • Can describe and use formal and actual parameters. • Know how a method can change the value of a parameter, if the parameter's class allows it. • Know the technique for overloading methods and also know how to program overloaded methods and constructors in practice.	• Know that the Java value parameters is just one alternative for parameter technique: there are languages with reference parameters, for example. •
Classes, objects and encapsulation	• Variables, algorithms, methods, parameters	• Can outline the objects for a class specification as "drawings" for creating objects.	• Can program instance variables and accessors. • Know the technique for encapsulation and can apply it in programming. • Know the concept of 'object state.' • Know the lifespan of an object and how it differs from the lifespan of the local variables of methods. • Can give objects as parameters. • Know the significance of automatic garbage	• Understand the significance of encapsulation in software design and the need for program validation. • Understand the consequences of automatic garbage collection in the applicability of the Java language and what Java is suitable for for this reason, and what it is completely unsuitable for.

			collection.	

Last updated: 22.09.2011 - 09:28 Arto Wikla

Post date: 05.09.2011 - 13:02 Marina Kurtén

Permanent link: https://www.cs.helsinki.fi/en/node/65659

Printer-friendly version

Appendix G

Extract from UTS Database Fundamentals – 31271, Practice Questions and Answers

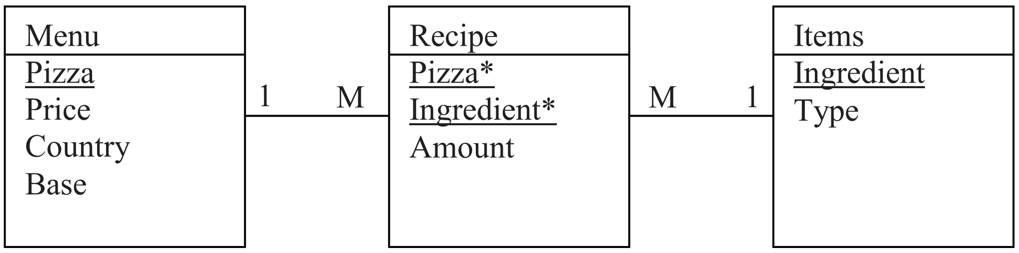
G.1 Introduction

This appendix reviews the material corresponding to the practice version of online sQL test used at UTS. The practice database schema, the *CREATE* statements corresponding to each table, the sample data stored in each table as well as different questions designed based on this information and their answered are reviewed.

G.2 Pizza Database

Pizza Database Description and Data

Relational Model:



```
create table menu
(
pizza char(20),
price real,
country char(20),
base char(20),
PRIMARY KEY (pizza)
);

create table items
(
ingredient char(12),
type char(8),
PRIMARY KEY (ingredient)
);

create table recipe
(
pizza  char(20),
ingredient  char(12),
amount  int,
PRIMARY KEY (pizza, ingredient),
FOREIGN KEY (pizza) REFERENCES
menu,
FOREIGN KEY (ingredient)
REFERENCES items
);
```

P.T.O. for table data.

recipe

pizza	ingredient	amount
margarita	spice	5
margarita	cheese	120
ham	ham	150
ham	spice	5
napolitana	anchovies	100
napolitana	olives	75
napolitana	spice	10
hawaiian	ham	100
hawaiian	pineapple	100
hawaiian	spice	5
cabanossi	cabanossi	150
cabanossi	spice	10
siciliano	onion	50
siciliano	capsicum	75
siciliano	olives	50
siciliano	anchovies	50
siciliano	spice	15
americano	salami	120
americano	pepperoni	75
americano	spice	10
mexicano	onion	75
mexicano	capsicum	75
mexicano	mushroom	50
mexicano	chilli	25
mexicano	spice	20
seafood	seafood	200
seafood	spice	5

recipe

pizza	ingredient	amount
garlic	garlic	25
garlic	spice	10
vegetarian	onion	50
vegetarian	capsicum	50
vegetarian	mushroom	50
vegetarian	peas	50
vegetarian	tomato	50
vegetarian	spice	5
mushroom	mushroom	100
mushroom	spice	5
special	cheese	25
special	tomato	25
special	ham	25
special	anchovies	25
special	olives	25
special	mushroom	25
special	bacon	25
special	egg	25
special	pineapple	25
special	cabanossi	25
special	salami	25
special	capsicum	25
special	onion	25
special	peas	25
special	seafood	25
special	spice	10
stagiony	ham	75
stagiony	mushroom	50
stagiony	olives	50
stagiony	anchovies	25
stagiony	spice	10

Pizza Database Description and Data

menu

pizza	price	country	base
margarita	6.2	italy	Wf
ham	7.3		Wf
napolitana	7.4	italy	Wf
stagiony	7.8	italy	Wm
hawaiian	7.4	hawaii	Wm
cabanossi	7.4	italy	Wf
special	9.9		Wf
siciliano	7.4	italy	Wm
americano	7.4	usa	Wm
mexicano	7.4	mexico	Wf
vegetarian	7.4		Wm
mushroom	7.3		Wm
seafood	9.2		Wm
garlic	3.5		Wm

items

ingredient	type
cheese	dairy
tomato	veg
ham	meat
anchovies	fish
olives	veg
mushroom	veg
prawn	fish
garlic	spice
egg	dairy
pineapple	fruit
cabanossi	meat
salami	meat
pepperoni	meat
capsicum	veg
onion	veg
bacon	meat
chilli	spice
peas	veg
seafood	fish
spice	spice

G.2.1 Questions Used in The Practice Online SQL Test

Table G.1 reviews the questions designed for different *difficulty levels* based on the Pizza database.

G.2.2 Proposed Answers for The Questions Used in The Practice Online SQL Test

Table G.2 reviews the proposed answers for different questions presented in Table G.1.

TABLE G.1: List of questions and their corresponding covered topic.

Question No.	Question	Topic
1.1	List pizzas with the substring 'i' anywhere within the pizza name.	Simple
1.2	List all pizzas, giving pizza name, price and country of origin where the country of origin has NOT been recorded (i.e. is missing).	Simple
1.3	List all price categories recorded in the MENU table, eliminating duplicates.	Simple
2.1	Give the average price of pizzas from each country of origin.	Group by
2.2	Give the most expensive pizzas from each country of origin.	Group by
2.3	Give the cheapest pizzas from each country of origin.	Group by
3.1	Give the average price of pizzas from each country of origin, do not list countries with only one pizza.	Group by with having
3.2	Give the average price of pizzas from each country of origin, only list countries with 'i' in the country's name	Group by with having
3.3	Give cheapest price of pizzas from each country of origin, only list countries with cheapest price of less than $7.00	Group by with having
4.1	List all ingredients and their types for the 'margarita' pizza. Do not use a subquery.	Natural join
4.2	List all 'fish' ingredients used in pizzas, also list the pizza names. Do not use a subquery.	Natural join
4.3	List all 'meat' ingredients used in pizzas, also list the pizza names. Do not use a subquery.	Natural join
5.1	Give pizzas and prices for pizzas that are more expensive than all Italian pizzas. You must use a subquery.	Simple sub-query
5.2	List all ingredients for the Mexican pizza (i.e. country = 'mexico'). You must use a subquery.	Simple sub-query
5.3	List pizzas with at least one 'meat' ingredient.You must use a subquery.	Simple sub-query
6.1	Give all pizzas that originate from the same country as the 'siciliano' pizza.	Self-join
6.2	List all pizzas that cost more than 'stagiony' pizza, also give their prices.	Self-join
6.3	List all pizzas that cost less than 'siciliano' pizza, also give their prices.	Self-join
7.1	List each ingredient and the pizza that contains the largest amount of this ingredient.	Correlated sub-query
7.2	List ingredients used in more than one pizza.	Correlated sub-query
7.3	List the ingredients, and for each ingredient, also list the pizza that contains the largest amount of this ingredient.	Correlated sub-query

TABLE G.2: Proposed Answers for the questions presented in G.1

Question No.	**Proposed answer**
1.1	select pizza from menu where pizza like '%i%'
1.2	select pizza, price, country from menu where country is null
1.3	select distinct price from menu
2.1	select country, AVG(price) AS Average from menu where country is not null group by country
2.2	Select country, MAX(price) AS Most from menu where country is not null group by country
2.3	Select country, MIN(price) AS Least from menu where country is not null group by country
3.1	Select country, AVG(price) from menu where country is not null group by country having COUNT(*) > 1
3.2	Select country, AVG(price) from menu where country is not null group by country having country LIKE '%i%'
3.3	Select country, MIN(price) from menu where country is not null group by country having MIN(price) < 7.00
4.1	Select i.ingredient, type from recipe r, items i where pizza = 'margarita' and i.ingredient = r.ingredient
4.2	Select i.ingredient, pizza from items i, recipe r where i.ingredient=r.ingredient and type = 'fish'
4.3	Select i.ingredient, pizza from items i, recipe r where i.ingredient=r.ingredient and type = 'meat'
5.1	select pizza, price from menu where price > all (select price from menu where country = 'italy')
5.2	Select distinct ingredient from recipe where pizza in (select pizza from menu where country like '%mexico%')
5.3	Select distinct pizza from recipe where ingredient = any (select ingredient from items where type like '%meat%')
6.1	Select m1.pizza from menu m1, menu m2 where m1.country = m2.country and m2.pizza = 'siciliano' and m1.pizza <> 'siciliano'
6.2	Select m1.pizza, m1.price from menu m1, menu m2 where m1.price > m2.price and m2.pizza = 'stagiony'
6.3	Select m1.pizza, m1.price from menu m1, menu m2 where m1.price < m2.price and m2.pizza = 'siciliano'
7.1	select ingredient, pizza, amount from recipe r where amount = (select max(amount) from recipe where ingredient = r.ingredient)
7.2	Select distinct ingredient from recipe r where ingredient in (select ingredient from recipe where pizza <> r.pizza)
7.3	Select ingredient, pizza, amount from recipe r where amount = (select MAX(amount) from recipe where ingredient = r.ingredient)

Appendix H

The Final Exam Questions Used at Helsinki University

This appendix review four questions of the final exam. Due to local administrative regulations, the complete final exam is not to be released.

Question 2, part a (3p)
Create a program that outputs (using a loop statement such as while or for) all integers divisible with 2, starting with 1000 and ending in 2. The output must occur so that 5 integers are printed on each row, and that each column must be aligned. The program output should look like this:

```
1000 998 996 994 992
 990 988 986 984 982
 980 978 976 974 972
     (lots of rows)
  10   8   6   4   2
```

Question 2, part b (4p)
Create a program where the input is integers representing the exam points gained by students. The program starts by reading the numbers of points from the user. The reading of the points stops when the user enters the integer -1.

The number of points must be an integer between 0 and 30. If some other integer is input (besides -1 that ends the program), the program ignores it.

After reading the numbers of points, the program states which number of points (between 0 and 30) is the greatest. Out of the number of points, the integers under 15 are equivalent to the grade *failed*, and the rest are *passed*. The program also announces the number of passed and failed grades.

Example:

```
Enter numbers of exam points, -1 ends the program:
20
12
29
15
-1

best number of points: 29
passed: 3
failed: 1
```

In the above example, 12 points failed and the points 20, 29 and 15 passed exams. Thus, the program announces that 3 students passed and 1 student failed.
Please note that the program must ignore all integers outside 0-30. An example of a case where there are integers that have to be ignored among the input numbers:

```
Enter numbers of exam points, -1 ends the program:
10
100
20
-4
30
-1

best number of points: 30
passed: 2
failed: 1
```

As shown, the points -4 and 100 are ignored.

Question 3, part a (3p)
Create the method `public static void printInterval(int edge1, edge2)` that prints, in ascending order, each integer in the interval defined by its parameters.

If we call `printInterval(3, 7)`, it prints

```
3 4 5 6 7
```

The methods also works if the first parameter is greater than the second one, i.e. if we call `printInterval(10, 8)`, it prints

```
8 9 10
```

Thus, the integers are always printed in ascending order, regardless of which method parameter is greater, the first one or the second one.

Question 3, part b (3p)
Create the method `public static boolean bothFound(int[] integers, int integer1, integer2)`, which is given an integer array and two integers as parameters. The method returns true if both integers given as parameters (`integer1` and `integer2`) are in the array given as method parameter. In other cases the method returns false.

If the method receives as parameters for example the array [1,5,3,7,5,4], and the integers 5 and 7, it returns true. If the method received the array [1,5,3,2] and the integers 7 and 3 as parameters, it would return false.

Create a main program, as well, which demonstrates how to use the method.

Note! If you don't know how to use arrays, you can create `public static boolean bothFound(ArrayList<Integer> integers, int integer1, int integer2)`, where the method is given as parameters an ArrayList containing the integers and the integers to be found.

Question 4. (6 points)
Create the class Warehouse. The warehouse has a capacity, which is an integer, and the amount of wares stored in the warehouse is also stored as an integer. The warehouse capacity if specified with the constructor parameter (you can assume that the value of the parameter is positive). The class has the following methods:

- `void add(int amount)`, that adds the amount of wares given in the parameter to the warehouse. If the amount is negative, the status of the warehouse does not change. When adding wares, the amount of wares in the warehouse cannot grow larger than the capacity. If the amount to be added does not fit into the warehouse completely, the warehouse is filled and the rest of the wares are 'wasted."
- `int space()`, that returns the amount of empty space in the warehouse.
- `void empty()`, that empties the warehouse.
- `toString()`, which returns a text representation of the warehouse status, formulated as in the example below; observe the status when the warehouse is empty!

Next is an example that demonstrates the operations of a warehouse that has been implemented correctly:

```
public static void main(String[] args) {
  Warehouse warehouse = new Warehouse(24);
  warehouse.add(10);
  System.out.println(warehouse);
  System.out.println("space in warehouse " + warehouse.space());
  warehouse.add(-2);
  System.out.println(warehouse);
  warehouse.add(50);
  System.out.println(warehouse);
  warehouse.empty();
  System.out.println(warehouse);\\
```

if the class has been implemented correctly, the output is

```
capacity: 24 items 10
space in warehouse 14
capacity: 24 items 10
capacity: 24 items 24
capacity: 24 empty
```

Question 5. (6 points)

This assignment is about making a program to manage the contents of a bookshelf. You have at your disposal the class Book:

```
public class Book {
 private String author;
 private String title;

 public Book(String author, String title) {
   this.author = author;
   this.name = name;
 }

 public String getAuthor() {
   return this.author;
 }

 @Override
 public String toString() {
    return this.author + ": " + this.name;
  }
}
```

Please program the class Bookshelf, that works like the example described below:

```
public static void main(String[] args) {
    Bookshelf shelf = new Bookshelf();
    shelf.addBook("Kent Beck", "Test Driven Development");
    shelf.addBook("Kent Beck", "Extreme Programming Embraced");
    shelf.addBook("Martin Fowler", "UML Distilled");
    shelf.addBook("Fedor Dostoyevski", "Crime and Punishment");

    shelf.print();
    System.out.println("---");
    shelf.get("Kent Beck");\\
}
```

if the class has been implemented correctly, the output is
books total 4
books:
Kent Beck: Test Driven Development
Kent Beck: Extreme Programming Embraced
Martin Fowler: UML Distilled
Fedor Dostoyevski: Crime and Punishment

found:
Kent Beck: Test Driven Development Kent Beck: Extreme Programming Embraced

The class must save books added to the shelf in an ArrayList containing Book objects.

As we can see in the example, the following methods have to be implemented for the class:

- `void addBook(String author, String title)`, which adds a book with the author and title given as parameters to the shelf.

- `void print()`, which prints the information of the bookshelf formulated as in the example above.

- `find(String author)`, which outputs the books in the shelf with the author given as method parameter, the output should be in the same form as in the example above.

Appendix I

Extract from Helsinki Subject Content – 581325 "Introduction to Programming" Sem. 2 2016, Week 1.

This chapter reviews detailed information on the content of the first week material used in Introduction to Programming at University of Helsinki.

Object-Oriented Programming with Java, part I ››

Authors: Arto Vihavainen, Matti Luukkainen
Translators to English: Emilia Hjelm, Alex H. Virtanen, Matti Luukkainen, Virpi Sumu, Birunthan Mohanathas

1. 1. THE PROGRAM AND THE SOURCE CODE

1.1 1.1 SOURCE CODE

A computer program is composed of commands written in the *source code*. A computer generally runs *commands* in the source code from *top to bottom* and *from left to right*. Source code is saved in a textual format and will be *executed* somehow.

1.2 1.2 COMMANDS

Computers execute different *operations*, or actions, based on the commands. For example, when printing the text "Hello world!" on the screen, it is done by the command `System.out.println`.

```
System.out.println("Hello world!");
```

The `System.out.println` command prints the string given inside the brackets on the screen. The suffix `ln` is short for the word *line*. Therefore, this command prints out a line. This means that after the given string has been printed, the command will also print a line break.

1.3 1.3 COMPILER AND INTERPRETER

Computers do not directly understand the programming language we are using. We need a *compiler* between the source code and the computer. When we are programming using the command line interface, the command `javac Hello.java` will compile the `Hello.java` file into *bytecode*, which can be executed using the Java interpreter. To run the compiled program, you can use the command `java Hello` where Hello is the name of the original source code file.

When using a modern development environment (more on this later), it will take care of compiling the source code. When we choose to run the program, the development environment will compile and execute the program. All development environments compile source code while it is being written by the programmer, which means that simple errors will be noticed before executing the program.

1.4 1.4 COMPONENTS OF COMMANDS

1.4.1 1.4.1 SEMICOLON

A semicolon `;` is used to separate different commands. The compiler and the interpreter both ignore line breaks in the source code, so we could write the entire program on a single line.

In the example below we will use the `System.out.print` command, which is similar to the `System.out.println` command except that it will not print a line break after printing the text.

Example of how the semicolons are used

```
System.out.print("Hello "); System.out.print("world");
System.out.print("!");
```

```
Hello world!
```

Even though neither the compiler nor the interpreter need line breaks in the source code, they are very important when considering human readers of the source code. Line breaks are required to divide source code in a clear manner. Readability of source code will be emphasized throughout this course.

1.4.2 1.4.2 PARAMETERS (INFORMATION PASSED TO COMMANDS)

The information processed by a command are the *parameters of a command*. They are passed to the command by placing them between `()` brackets that follow the command name. For example, the `System.out.print` command is given the text *hello* as a parameter as follows: `System.out.print("hello")`.

1.4.3 1.4.3 COMMENTS

Comments are a useful way to make notes in the source code for yourself and others. Everything on a line after two forward slashes `//` is treated as a comment.

1.4.4 1.4.4 EXAMPLE OF USING COMMENTS

```
// We will print the text "Hello world"
System.out.print("Hello world");

System.out.print(" and all the people of the world."); // We print more text to the sa

// System.out.print("this line will not be executed, because it is commented out"
```

The last line of the example introduces a particularly handy use for comments: you can comment out code instead of completely deleting it if you want to temporarily try out something.

1.5 1.5 MORE ABOUT PRINTING

As we can see from the examples above, there are two commands for printing.

- `System.out.print` prints the text without the line break at the end
- `System.out.println` prints the text and the line break

The printed text can contain both traditional characters and special characters. The most important special character is `\n`, which stands for a line break. There are also other special characters.

```
System.out.println("First\nSecond\nThird");
```

When executed, the example above prints:

```
First
Second
Third
```

2. 2. MAIN PROGRAM BODY

The body for a program named "Example" is as follows:

```java
public class Example {
    public static void main(String[] args) {
        // program code
    }
}
```

The program is stored in a text file named after the program with the *.java* extension. For a program named *Example*, the file should be named `Example.java`.

The execution of the program begins at the part marked with the *// program code* comment above. During our first week of programming, we will limit ourselves to this part. When we are talking about commands such as printing, we need to write the commands into the program body. For example: `System.out.print("Text to be printed");`

```java
public class Example {
    public static void main(String[] args) {
        System.out.print("Text to be printed");
    }
}
```

From this point on, the main program body will be omitted from the examples.

3. 3. GETTING TO KNOW YOUR DEVELOPMENT ENVIRONMENT

Programming these days takes place in development environments almost without exceptions. The development environment provides several tools and features to assist the programmer. Although the development environment does not write the program on behalf of the programmer, it contains several handy features such as hinting about mistakes in code and assisting the programmer to visualize the structure of the program.

In this course, we will use the NetBeans development environment. A guide for using NetBeans is available here.

Until you become familiar with NetBeans, follow the guides and steps precisely. Most of the following exercises show what needs to be printed to the screen for the program to function correctly.

Note: Do not do the exercises by writing code and then clicking the test button. You should also execute the code manually (green arrow) and observe the result on the screen. This is especially useful if an exercise fails to pass the tests.

In the following exercises, we will practice the use of NetBeans and printing of text on the screen.

Remember to read the guide on using NetBeans before you continue!

Please answer to our survey: here. It will take less than five minutes.

Exercise 1: Name

Exercise 1: Name

Exercise 2: Hello world! (And all the people of the world)

Exercise 2: Hello world! (And all the people of the world)

Exercise 3: Spruce

Exercise 3: Spruce

Note: You probably wrote `System.out.println("...")` quite a few times. Try typing only *sout* on an empty line in NetBeans and then press the tab key. What happened? This tip will save a lot of your time in the future!

4. 4. VARIABLES AND ASSIGNMENT

4.1 4.1 VARIABLES AND DATA TYPES

A *variable* is one of the most important concepts in computer programming. A variable should be imagined as a box in which you can store information. The information stored in a variable always has a type. These types include text (*String*), whole numbers (*int*), decimal numbers (*double*), and truth values (*boolean*). A *value* can be assigned to a variable using the equals sign (=).

```
int months = 12;
```

In the statement above, we assign the value 12 to the variable named `months` whose data type is integer (`int`). The statement is read as "the variable `months` is assigned the *value* 12".

The value of the variable can be appended to a string with the plus + sign as shown in the following example.

```
String text = "includes text";
int wholeNumber = 123;
double decimalNumber = 3.141592653;
boolean isTrue = true;

System.out.println("The variable's type is text. Its value is " + text);
System.out.println("The variable's type is integer. Its value is  " + wholeNumber);
System.out.println("The variable's type is decimal number. Its value is " + decimalNumber);
System.out.println("The variable's type is truth value. Its value is " + isTrue);
```

Printing:

```
The variable's type is text. Its value is includes text
The variable's type is integer. Its value is 123
The variable's type is decimal number. Its value is 3.141592653
The variable's type is truth value. Its value is true
```

A variable holds its value until it is assigned a new one. Note that the variable type is written only when the variable is first declared in the program. After that we can use the variable by its name.

```
int wholeNumber = 123;
System.out.println("The variable's type is integer. Its value is  " + wholeNumber);

wholeNumber = 42;
System.out.println("The variable's type is integer. Its value is  " + wholeNumber);
```

The output is:

```
The variable's type is integer. Its value is 123
The variable's type is integer. Its value is 42
```

4.2 4.2 VARIABLE DATA TYPES ARE IMMUTABLE

When a variable is declared with a data type, it cannot be changed later. For example, a text variable cannot be changed into an integer variable and it cannot be assigned integer values.

```
String text = "yabbadabbadoo!";
text = 42; // Does not work! :(
```

Integer values can be assigned to decimal number variables, because whole numbers are also decimal numbers.

```
double decimalNumber = 0.42;
decimalNumber = 1; // Works! :)
```

Exercise 4: Varying variables

Exercise 4: Varying variables

4.3 4.3 ALLOWED AND DESCRIPTIVE VARIABLE NAMES

There are certain limitations on the naming of our variables. Even though umlauts, for example, can be used, it is better to avoid them, because problems might arise with character encoding. For example, it is recommended to use A instead of Ä.

Variable names must not contain certain special characters like exclamation marks (!). Space characters cannot be used, either, as it is used to separate commands into multiple parts. It is a good idea to replace the space character using a *camelCase* notation. **Note:** The first character is always written in lower case when using the camel case notation.

```
int camelCaseVariable = 7;
```

Variable names can contain numbers as long it does not start with one. Variable names cannot be composed solely of numbers, either.

```
int 7variable = 4; // Not allowed!
int variable7 = 4; // A valid, but not descriptive variable name
```

Variable names that have been defined before cannot be used. Command names such as `System.out.print` cannot be used, either.

```
int camelCase = 2;
int camelCase = 5; // Not allowed, the variable camelCase is already defined!
```

It is strongly recommended to name variables so that their purpose can be understood without comments and without thinking. Variable names used in this course **must** be descriptive.

4.3.1 4.3.1 VALID VARIABLE NAMES

- lastDay = 20
- firstYear = 1952
- name = "Matti"

4.3.2 4.3.2 INVALID VARIABLE NAMES

- last day of the month = 20
- 1day = 1952
- watchout! = 1910
- 1920 = 1

5. 5. CALCULATION

The calculation operations are pretty straightforward: `+`, `-`, `*` and `/`. A more peculiar operation is the modulo operation `%`, which calculates the remainder of a division. The order of operations is also pretty straightforward: the operations are calculated from left to right taking the parentheses into account.

```
int first = 2;   // variable of whole number type is assigned the value 2
int second = 4;  // variable of whole number type is assigned the value 4
int sum = first + second;  // variable of whole number type is assigned the value of
                           //      (which means 2 + 4)

System.out.println(sum); // the value of the sum of variables is printed
```

```
int calcWithParens = (1 + 1) + 3 * (2 + 5);  // 23
int calcWithoutParens = 1 + 1 + 3 * 2 + 5;   // 13
```

The parentheses example above can also be done step by step.

```
int calcWithParens = (1 + 1);
calcWithParens = calcWithParens + 3 * (2 + 5);  // 23

int calcWithoutParens = 1 + 1;
calcWithoutParens = calcWithoutParens + 3 * 2;
calcWithoutParens = calcWithoutParens + 5;      // 13
```

Calculation operations can be used almost anywhere in the program code.

```
int first = 2;
int second = 4;

System.out.println(first + second);
System.out.println(2 + second - first - second);
```

5.1 5.1 FLOATING POINT NUMBERS (DECIMAL NUMBERS)

Calculating the division and remainder of whole numbers is a little trickier. A floating point number (decimal number) and integer (whole number) often get mixed up. If all the variables in a calculation operation are integers, the end result will also be an integer.

```
int result = 3 / 2;  // result is 1 (integer) because 3 and 2 are integers as well
```

```
int first = 3:
int second = 2;
double result = first / second;  // the result is again 1 because first and second ar
```

The remainder can be calculated using the remainder operation (`%`). For example, the calculation `7 % 2` yields 1.

```
int remainder = 7 % 2;  // remainder is 1 (integer)
```

If either the dividend or the divisor (or both!) is a floating point number (decimal number) the end result will also be a floating point number.

```
double whenDividendIsFloat = 3.0 / 2;  // result is: 1.5
double whenDivisorIsFloat = 3 / 2.0;   // result is: 1.5
```

If needed, integers can be converted to floating point using the type cast operation `(double)` as follows:

```
int first = 3;
int second = 2;
double result1 = (double)first / second;  // result is: 1.5

double result2 = first / (double)second;  // result is: 1.5

double result3 = (double)(first / second);  // result is: 1
```

In the last example calculation, the result is rounded incorrectly because the calculation between the integers is done before the type cast to a floating point number.

If the quotient is assigned to a variable of integer type, the result will be an integer as well.

```
int integerResultBecauseTypeIsInteger = 3.0 / 2;  // quotient is automatically intege
```

The next example will print "1.5" because the dividend is transformed into a floating point number by multiplying it with a floating point number (1.0 * 3 = 3.0) before the division.

```
int dividend = 3;
int divisor = 2;

double quotient = 1.0 * dividend / divisor;
System.out.println(quotient);
```

What does the following code print?

```
int dividend = 3;
int divisor = 2;

double quotient = dividend / divisor * 1.0;
System.out.println(quotient);
```

From now on, make sure that you name your variables that follow good conventions like the variables in the examples above.

Exercise 5: Seconds in a year

Exercise 5: Seconds in a year

6. 6. CONCATENATION OR COMBINING STRINGS

Let us take a closer look on combining strings with the + operator.

If the + operator is used between two strings, a new string is created with the two strings combined. Note the clever use of space characters in the values of the variables below!

```
String greeting = "Hi ";
String name = "John";
String goodbye = ", and goodbye!";

String sentence = greeting + name + goodbye;

System.out.println(sentence);
```

```
Hi John, and goodbye!
```

If a string is on either side of the + operator, the other side is converted to a string and a new string is created. For example, the integer 2 will be converted into the string "2" and then combined with the other string.

```
System.out.println("there is an integer --> " + 2);
System.out.println(2 + " <-- there is an integer");
```

What we learned earlier about the order of operations is still valid:

```
System.out.println("Four: " + (2 + 2));
System.out.println("But! Twenty-two: " + 2 + 2);
```

```
Four: 4
But! Twenty-two: 22
```

Using this information, we can print a mix of strings and values of variables:

```java
int x = 10;

System.out.println("variable x has the following value: " + x);

int y = 5;
int z = 6;

System.out.println("y has the value  " + y + " and z has the value " + z);
```

This program obviously prints:

```
variable x has the following value: 10
y has the value 5 and z has the value 6
```

Exercise 6: Addition

Exercise 6: Addition

Exercise 7: Multiplication

Exercise 7: Multiplication

7. 7. READING USER INPUT

So far our programs have been rather one-sided. Next we will learn how to read *input* from the user. We will use a special *Scanner* tool to read the user input.

Let us add the *Scanner* to our existing main program body. Do not worry if the main program body seems obscure as we will continue to write our code in the part marked *// program code*.

```
import java.util.Scanner;

public class ProgramBody {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        // program code
    }
}
```

7.1 7.1 READING A STRING

The following code reads the user's name and prints a greeting:

```
System.out.print("What is your name? ");
String name = reader.nextLine(); // Reads a line of input from the user and assigns i
                                 //      to the variable called name

System.out.println("Hi, " + name);
```

```
What is your name? John
Hi, John
```

The program above combined along with the main program body is shown below. The name of the program is *Greeting*, which means that it must be located in a file named `Greeting.java`.

```
import java.util.Scanner;

public class Greeting {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        System.out.print("Who is greeted: ");
        String name = reader.nextLine(); // Reads a line of input from the user and a
                                         //      to the variable called name
        System.out.print("Hi " + name);
    }
}
```

When the program above is executed, you can type the input. The output tab in NetBeans (at the bottom) looks as follows when the program has finished (the user inputs the name "John").

```
run:
Who is greeted: John
Hi John
BUILD SUCCESSFUL (total time: 6 seconds)
```

7.2 7.2 READING INTEGERS

Our Scanner tool is not good for reading integers, so we will use another special tool to read an integer. The command `Integer.parseInt` converts the string given to it into an integer. The command's parameter is given between brackets and it returns an integer that can be assigned to an integer variable.

Basically, we are joining two commands together. First we read the input as a string from the user and immediately give it to the command `Integer.parseInt`.

```
System.out.print("Type an integer: ");
int number = Integer.parseInt(reader.nextLine());

System.out.println("You typed " + number);
```

Next we will ask the user to give us his name and age. The program body is included this time.

```
import java.util.Scanner;

public class NameAndAgeGreeting {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        System.out.print("Your name: ");
        String name = reader.nextLine();    // Reads a line from the users keyboard

        System.out.print("How old are you: ");
        int age = Integer.parseInt(reader.nextLine()); // Reads a string variable fro

        System.out.println("Your name is: " + name + ", and you are " + age + " years old, 
    }
}
```

7.3 7.3 SUMMARY

The program body for interaction with the user is as follows:

```
import java.util.Scanner;
public class ProgramName {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        // code here
    }
}
```

Reading a string:

```
String text = reader.nextLine();
```

Reading an integer:

```
int number = Integer.parseInt(reader.nextLine());
```

Exercise 8: Adder

Exercise 8: Adder

Exercise 9: Divider

Exercise 9: Divider

Exercise 10: Calculating the circumference

Exercise 10: Calculating the circumference

Exercise 11: Bigger number

Exercise 11: Bigger number

Exercise 12: Sum of the ages

Exercise 12: Sum of the ages

Exercise 13: NHL statistics, part 1

Exercise 13: NHL statistics, part 1

8. 8. CONDITIONAL STATEMENTS AND TRUTH VALUES

So far, our programs have progressed from one command to another in a straightforward manner. In order for the program to *branch* to different execution paths based on e.g. user input, we need conditional statements.

```
int number = 11;

if (number > 10) {
    System.out.println("The number was greater than 10");
}
```

The condition `(number > 10)` evaluates into a truth value; either `true` or `false`. The `if` command only handles truth values. The conditional statement above is read as "if the number is greater than 10".

Note that the `if` statement is not followed by semicolon as the condition path continues after the statement.

After the condition, the opening curly brace `{` starts a new *block*, which is executed if the condition is true. The *block* ends with a closing curly brace `}`. Blocks can be as long as desired.

The comparison operators are:

- `>` Greater than
- `>=` Greater than or equal to
- `<` Less than
- `<=` Less than or equal to
- `==` Equals
- `!=` Not equal

```
int number = 55;

if (number != 0) {
    System.out.println("The number was not equal to 0");
}

if (number >= 1000) {
    System.out.println("The number was greater than or equal to 1000");
}
```

A block can contain any code including other `if` statements.

```
int x = 45;
int number = 55;

if (number > 0) {
    System.out.println("The number is positive!");
    if (number > x) {
        System.out.println(" and greater than the value of variable x");
        System.out.println("after all, the value of variable x is " + x);
    }
}
```

The comparison operators can also be used outside the `if` statements. In such case the truth value will be stored in a truth value variable.

```
int first = 1;
int second = 3;

boolean isGreater = first > second;
```

In the example above the boolean (i.e. a truth value) variable `isGreater` now includes the truth value *false*.

A boolean variable can be used as a condition in a conditional sentence.

```java
int first = 1;
int second = 3;

boolean isLesser = first < second;

if (isLesser) {
    System.out.println(first + " is less than " + second + "!");
}
```

```
1 is less than 3!
```

8.1 8.1 CODE INDENTATION

Note that the commands in the block following the `if` statement (i.e. the lines after the curly brace, `{`) are not written at the same level as the `if` statement itself. They should be **indented** slightly to the right. Indentation happens when you press the tab key, which is located to the left of q key. When the block ends with the closing curly brace, indentation ends as well. The closing curly brace `}` should be on the same level as the original `if` statement.

The use of indentation is crucial for the readability of program code. During this course and generally everywhere, you are expected to indent the code properly. NetBeans helps with the correct indentation. You can easily indent your program by pressing shift, alt, and f simultaneously.

8.2 8.2 ELSE

If the truth value of the comparison is false, another optional block can be executed using the `else` command.

```java
int number = 4;

if (number > 5) {
    System.out.println("Your number is greater than five!");
} else {
    System.out.println("Your number is equal to or less than five!");
}
```

```
Your number is equal to or less than five!
```

Exercise 14: A positive number

Exercise 14: A positive number

Exercise 15: Age of majority

Exercise 15: Age of majority

Exercise 16: Even or odd?

Exercise 16: Even or odd?

8.3 8.3 ELSE IF

If there are more than two conditions for the program to check, it is recommended to use the `else if` command. It works like the `else` command, but with an additional condition. `else if` comes after the `if` command. There can be multiple `else if` commands.

```
int number = 3;

if (number == 1) {
    System.out.println("The number is one.");
} else if (number == 2) {
    System.out.println("The number is two.");
} else if (number == 3) {
    System.out.println("The number is three!");
} else {
    System.out.println("Quite a lot!");
}
```

```
The number is three!
```

Let us read out loud the example above: If number is one, print out "The number is one.". Otherwise if the number is two, print out "The number is two.". Otherwise if the number is three, print out "The number is three!". Otherwise print out "Quite a lot!".

8.4 8.4 COMPARING STRINGS

Strings cannot be compared using the equality operator (==). For string comparison, we use the `equals`. command, which is always associated with the string to compare.

```
String text = "course";

if (text.equals("marzipan")) {
    System.out.println("The variable text contains the text marzipan");
} else {
    System.out.println("The variable text does not contain the text marzipan");
}
```

The `equals` command is always attached to the string variable with a dot in between. A string variable can also be compared to another string variable.

```
String text = "course";
String anotherText = "horse";

if (text.equals(anotherText)) {
    System.out.println("The texts are the same!");
} else {
    System.out.println("The texts are not the same!");
}
```

When comparing strings, it is crucial to make sure that both string variables have been assigned some value. If a value has not been assigned, the program execution terminates with a *NullPointerException* error, which means that variable has no value assigned to it (*null*).

Exercise 17: Greater number

Exercise 17: Greater number

Exercise 18: Grades and points

Exercise 18: Grades and points

8.5 8.5 LOGICAL OPERATIONS

The condition statements can be made more complicated using logical operations. The logical operations are:

- `condition1 && condition2` is true if both conditions are true.
- `condition1 || condition2` is true if either of the conditions are true.
- `!condition` is true if the condition is false.

Below we will use the AND operation `&&` to combine two individual conditions in order to check if the value of the variable is greater than 4 **and** less than 11 (i.e. in the range 5 - 10).

```
System.out.println("Is the number between 5-10?");
int number = 7;

if (number > 4 && number < 11) {
    System.out.println("Yes! :)");
} else {
    System.out.println("Nope :(")
}
```

```
Is the number between 5-10?
Yes! :)
```

Next up is the OR operation `||`, which will be used to check if the value is less than 0 **or** greater than 100. The condition evaluates to true if the value fulfills either condition.

```
System.out.println("Is the number less than 0 or greater than 100?");
int number = 145;

if (number < 0 || number > 100) {
    System.out.println("Yes! :)");
} else {
    System.out.println("Nope :(")
}
```

```
Is the number less than 0 or greater than 100?
Yes! :)
```

Now we will use the negation operation `!` to negate the condition:

```
System.out.println("Is the string equal to 'milk'?");
String text = "water";

if (!(text.equals("milk"))) {  // true if the condition text.equals("milk") is false
    System.out.println("No!");
} else {
    System.out.println("Yes")
}
```

```
Is the text equal to 'milk'?
No!
```

For complicated conditions, we often need parentheses:

```
int number = 99;

if ((number > 0 && number < 10) || number > 100 ) {
    System.out.println("The number was in the range 1-9 or it was over 100");
} else {
    System.out.println("The number was equal to or less than 0 or it was in the range 10-99");
}
```

```
The number was equal to or less than 0 or it was in the range 10-99
```

Exercise 19: Age check

Exercise 19: Age check

Exercise 20: Usernames

Exercise 20: Usernames

Exercise 21: Leap year

Exercise 21: Leap year

9. 9. INTRODUCTION TO LOOPS

Conditional statements allow us to execute different commands based on the conditions. For example, we can let the user login only if the username and password are correct.

In addition to conditions we also need repetitions. We may, for example, need to keep asking the user to input a username and password until a valid pair is entered.

The most simple repetition is an infinite loop. The following code will print out the string *I can program!* forever or "an infinite number of times":

```java
while (true) {
    System.out.println("I can program!");
}
```

In the example above, the `while (true)` command causes the associated block (i.e. the code between the curly braces `{}`) to be *looped* (or repeated) infinitely.

We generally do not want an infinite loop. The loop can be interrupted using e.g. the `break` command.

```java
while (true) {
    System.out.println("I can program!");

    System.out.print("Continue? ('no' to quit)? ");
    String command = reader.nextLine();
    if (command.equals("no")) {
        break;
    }
}

System.out.println("Thank you and see you later!");
```

Now the loop progresses like this: First, the program prints *I can program!*. Then, the program will ask the user if it should continue. If the user types *no*, the `break` command is executed and the loop is interrupted and *Thank you and see you again!* is printed.

```
I can program!
Continue? ('no' to quit)?yeah
I can program!
Continue? ('no' to quit)? jawohl
I can program!
Continue? ('no' to quit)? no
Thank you and see you again!
```

Many different things can be done inside a loop. Next we create a simple calculator, which performs calculations based on commands that the user enters. If the command is *quit*, the `break` command will be executed to end the loop. Otherwise two numbers are asked. Then, if the initial command was *sum*, the program calculates and prints the sum of the two numbers. If the command was *difference*, the program calculates and prints the difference of the two numbers. If the command was something else, the program reports that the command was unknown.

```
System.out.println("welcome to the calculator");

while (true) {
    System.out.print("Enter a command (sum, difference, quit): ");
    String command = reader.nextLine();
    if (command.equals("quit")) {
        break;
    }

    System.out.print("enter the numbers");
    int first = Integer.parseInt(reader.nextLine());
    int second = Integer.parseInt(reader.nextLine());

    if (command.equals("sum") ) {
        int sum = first + second;
        System.out.println( "The sum of the numbers is " + sum );
    } else if (command.equals("difference")) {
        int difference = first - second;
        System.out.println("The difference of the numbers is " + difference);
    } else {
        System.out.println("Unknown command");
    }

}

System.out.println("Thanks, bye!");
```

Exercise 22: Password

Exercise 22: Password

Exercise 23: Temperatures

Exercise 23: Temperatures

Exercise 24: NHL statistics, part 2

Exercise 24: NHL statistics, part 2

Bibliography

Abdullah, AL, Areej Malibari, and Mona Alkhozae (2014). "STUDENTS'PERFORMANCE PREDICTION SYSTEM USING MULTI AGENT DATA MINING TECHNIQUE". In: *International Journal of Data Mining & Knowledge Management Process* 4.5, p. 1.

Ahadi, Alireza et al. (2015). "Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance". In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. Omaha, Nebraska, USA: ACM, pp. 121–130. ISBN: 978-1-4503-3630-7. DOI: `10.1145/2787622.2787717`. URL: `http://doi.acm.org/10.1145/2787622.2787717`.

Ahmadzadeh, Marzieh, Dave Elliman, and Colin Higgins (2005). "An analysis of patterns of debugging among novice computer science students". In: *ACM SIGCSE Bulletin* 37.3, pp. 84–88.

Ahmed, Abeer Badr El Din and Ibrahim Sayed Elaraby (2014). "Data Mining: A prediction for Student's Performance Using Classification Method". In: *World Journal of Computer Application and Technology* 2.2, pp. 43–47.

Allevato, Anthony et al. (2008). "Mining data from an automated grading and testing system by adding rich reporting capabilities". In: *Educational Data Mining*, pp. 167–176.

Anderson, John R and Robin Jeffries (1985). "Novice LISP errors: Undetected losses of information from working memory". In: *Human–Computer Interaction* 1.2, pp. 107–131.

Arockiam, L et al. (2010). "Deriving Association between Urban and Rural Students Programming Skills". In: *International Journal on Computer Science and Engineering* 2.3.

Attar, Sadaf Fatima Salim and YC Kulkarni (2015). "Precognition of Students Academic Failure Using Data Mining Techniques". In: *International Journal of Advanced Research in Computer and Communication Engineering*.

Barker, Ricky J and EA Unger (1983). "A predictor for success in an introductory programming class based upon abstract reasoning development". In: *ACM SIGCSE Bulletin*. Vol. 15. 1. ACM, pp. 154–158.

Basawapatna, Ashok et al. (2011). "Recognizing computational thinking patterns". In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, pp. 245–250.

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

Becker, Brett A and Catherine Mooney (2016). "Categorizing compiler error messages with principal component analysis". In: *12th China-Europe International Symposium on Software Engineering Education (CEISEE 2016), Shenyang, China, 28-29 May 2016*.

Bennedsen, Jens and Michael E Caspersen (2005). "An investigation of potential success factors for an introductory model-driven programming course". In: *Proceedings of the first international workshop on Computing education research*. ACM, pp. 155–163.

— (2006). "Abstraction ability as an indicator of success for learning object-oriented programming?" In: *ACM SIGCSE Bulletin* 38.2, pp. 39–43.

Bergin, Susan and Ronan Reilly (2005). "Programming: factors that influence success". In: *ACM SIGCSE Bulletin* 37.1, pp. 411–415.

— (2006). "Predicting introductory programming performance: A multi-institutional multivariate study". In: *Computer Science Education* 16.4, pp. 303–323.

Berland, M and T Martin (2011). "Clusters and patterns of novice programmers". In: *The meeting of the American Educational Research Association. New Orleans, LA*.

Berland, Matthew et al. (2013). "Using learning analytics to understand the learning pathways of novice programmers". In: *Journal of the Learning Sciences* 22.4, pp. 564–599.

Biamonte, AJ (1964). "Predicting success in programmer training". In: *Proceedings of the second SIGCPR conference on Computer personnel research*. ACM, pp. 9–12.

Biggs, John B (1978). "Individual and group differences in study processes". In: *British Journal of Educational Psychology* 48.3, pp. 266–279.

Blikstein, Paulo (2011). "Using learning analytics to assess students' behavior in open-ended programming tasks". In: *Proceedings of the 1st international conference on learning analytics and knowledge*. ACM, pp. 110–116.

Blikstein, Paulo et al. (2014). "Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming". In: *Journal of the Learning Sciences* 23.4, pp. 561–599. DOI: `10.1080/10508406.2014.954750`. URL: `http://dx.doi.org/10.1080/10508406.2014.954750`.

Bonar, Jeffrey and Elliot Soloway (1985). "Preprogramming knowledge: A major source of misconceptions in novice programmers". In: *Human–Computer Interaction* 1.2, pp. 133–161.

Booth, Shirley (1992). *Learning to program: A phenomenographic perspective*.

Brooks, Ruven (1977). “Towards a theory of the cognitive processes in computer programming”. In: *International Journal of Man-Machine Studies* 9.6, pp. 737–751.

Brown, Neil C. C. and Amjad Altadmri (2014). “Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data”. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER ’14. Glasgow, Scotland, United Kingdom: ACM, pp. 43–50. ISBN: 978-1-4503-2755-8. DOI: `10.1145/2632320.2632343`. URL: `http://dx.doi.org/10.1145/2632320.2632343`.

Brown, Neil Christopher Charles et al. (2014). “Blackbox: a large scale repository of novice programmers’ activity”. In: *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, pp. 223–228.

Bruce, Christine et al. (2006). “Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university”. In: *Transforming IT education: Promoting a culture of excellence*, pp. 301–325.

Buffardi, Kevin and Stephen H Edwards. “Adaptive and social mechanisms for automated improvement of eLearning materials”. In:

— (2014). “Introducing CodeWorkout: an adaptive and social learning environment”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. ACM, pp. 724–724.

Byrne, Pat and Gerry Lyons (2001). “The effect of student attributes on success in programming”. In: *ACM SIGCSE Bulletin*. Vol. 33. 3. ACM, pp. 49–52.

Cantwell Wilson, Brenda and Sharon Shrock (2001). “Contributing to success in an introductory computer science course: a study of twelve factors”. In: *ACM SIGCSE Bulletin*. Vol. 33. 1. ACM, pp. 184–188.

Carter, Adam S., Christopher D. Hundhausen, and Olusola Adesope (2015). “The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER ’15. Omaha, Nebraska, USA: ACM, pp. 141–150. ISBN: 978-1-4503-3630-7. DOI: `10.1145/2787622.2787710`. URL: `http://doi.acm.org/10.1145/2787622.2787710`.

Chandra, E and K Nandhini (2010). “Knowledge mining from student data”. In: *European journal of scientific research* 47.1, pp. 156–163.

Cooper, Stephen et al. (2005). *Outcomes-based computer science education*. Vol. 37. 1. ACM.

Détienne, Françoise (1990). “Expert programming knowledge: a schema-based approach”. In: *Psychology of programming*, pp. 205–222.

Du Boulay, Benedict (1986). “Some difficulties of learning to program”. In: *Journal of Educational Computing Research* 2.1, pp. 57–73.

Edwards, Stephen H (2003). “Improving student performance by evaluating how well students test their own programs”. In: *Journal on Educational Resources in Computing (JERIC)* 3.3, p. 1.

Edwards, Stephen H. and Manuel A. Perez-Quinones (2008). “Web-CAT: Automatically Grading Programming Assignments”. In: *SIGCSE Bull.* 40.3, p. 328. DOI: 10.1145/1597849.1384371. URL: http://doi.acm.org/10.1145/1597849.1384371.

Edwards, Stephen H. et al. (2009). “Comparing Effective and Ineffective Behaviors of Student Programmers”. In: *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*. ICER ’09. Berkeley, CA, USA: ACM, pp. 3–14. ISBN: 978-1-60558-615-1. DOI: 10.1145/1584322.1584325. URL: http://dx.doi.org/10.1145/1584322.1584325.

Evans, Gerald E and Mark G Simkin (1989). “What best predicts computer proficiency?” In: *Communications of the ACM* 32.11, pp. 1322–1327.

Fenwick, James B. et al. (2009). “Another Look at the Behaviors of Novice Programmers”. In: *SIGCSE Bull.* 41.1, pp. 296–300. ISSN: 0097-8418. DOI: 10.1145/1539024.1508973. URL: http://dx.doi.org/10.1145/1539024.1508973.

Francisco, Rodrigo Elias and Ana Paula Ambrosio (2015). “Mining an Online Judge System to Support Introductory Computer Programming Teaching.” In: *EDM (Workshops)*.

Gilmore, David J (1990). “Expert programming knowledge: a strategic approach”. In: *Psychology of programming*, pp. 223–234.

Gomes, Anabela Jesus, Alvaro Nuno Santos, and António José Mendes (2012). “A study on students’ behaviours and attitudes towards learning to program”. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, pp. 132–137.

Hagan, Dianne and Selby Markham (2000). “Does it help to have some programming experience before beginning a computing degree program?” In: *ACM SIGCSE Bulletin* 32.3, pp. 25–28.

Hosseini, Roya, Arto Vihavainen, and Peter Brusilovsky (2014). “Exploring Problem Solving Paths in a Java Programming Course”. In: *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group*.

Hovemeyer, David and Jaime Spacco (2013). “CloudCoder: A Web-based Programming Exercise System”. In: *J. Comput. Sci. Coll.* 28.3, p. 30. ISSN: 1937-4771. URL: http://portal.acm.org/citation.cfm?id=2400161.2400167.

Ihantola, Petri and Ville Karavirta (2011). “Two-dimensional parson’s puzzles: The concept, tools, and first observations”. In: *Journal of Information Technology Education* 10, p. 2011.

Ihantola, Petri et al. (2015). "Educational data mining and learning analytics in programming: Literature review and case studies". In: *Proceedings of the 2015 ITiCSE on Working Group Reports*. ACM, pp. 41–63.

Jackson, James, MJ Cobb, and Curtis Carver (2005). "Identifying top Java errors for novice programmers". In: *Frontiers in Education Conference*. Vol. 35. 1. STIPES, T4C.

Jadud, Matthew C (2006). "Methods and tools for exploring novice compilation behaviour". In: *Proceedings of the second international workshop on Computing education research*. ACM, pp. 73–84.

Johnson, Philip M. et al. (2004). "Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from Hackystat-UH". In: *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, pp. 136–144. ISBN: 0-7695-2165-7. DOI: `10.1109/isese.2004.1334901`. URL: `http://dx.doi.org/10.1109/isese.2004.1334901`.

Johnson, W Lewis and Elliot Soloway (1985). "PROUST: Knowledge-based program understanding". In: *IEEE Transactions on Software Engineering* 3, pp. 267–275.

Joni, Saj-Nicole et al. (1983). "Just so stories: how the program got that bug". In: *ACM SIGCUE Outlook* 17.4, pp. 13–26.

Kalelioğlu, Filiz (2015). "A new way of teaching programming skills to K-12 students: Code. org". In: *Computers in Human Behavior* 52, pp. 200–210.

Katz, Sandra et al. (2003). "A study to identify predictors of achievement in an introductory computer science course". In: *Proceedings of the 2003 SIGMIS conference on Computer personnel research: Freedom in Philadelphia–leveraging differences and diversity in the IT workforce*. ACM, pp. 157–161.

Kölling, Michael et al. (2003). "The BlueJ system and its pedagogy". In: *Computer Science Education* 13.4, pp. 249–268.

Kovačić, Zlatko J and JS Green (2010). *Predictive working tool for early identification of 'at risk' students*.

Kumar, Amruth N (2003). "Learning programming by solving problems". In: *Informatics curricula and teaching methods*. Springer, pp. 29–39.

— (2016). "Providing the Option to Skip Feedback in a Worked Example Tutor". In: *International Conference on Intelligent Tutoring Systems*. Springer, pp. 101–110.

Kumar, S Anupama and MN Vijayalakshmi (2011). "Implication of classification techniques in predicting student's recital". In: *Int. J. Data Mining Knowl. Manage. Process (IJDKP)* 1.5, pp. 41–51.

Kurland, D Midian et al. (1986). "A study of the development of programming ability and thinking skills in high school students". In: *Journal of Educational Computing Research* 2.4, pp. 429–458.

Lahtinen, Essi (2007). “A categorization of Novice Programmers: a cluster analysis study”. In: *Proceedings of the 19th annual workshop of the psychology of programming interest group, Joensuu, Finnland*. Citeseer, pp. 32–41.

Leeper, RR and JL Silver (1982). “Predicting success in a first programming course”. In: *ACM SIGCSE Bulletin*. Vol. 14. 1. ACM, pp. 147–150.

Lewis, Tracy L et al. (2005). “The effects of individual differences on CS2 course performance across universities”. In: *ACM SIGCSE Bulletin*. Vol. 37. 1. ACM, pp. 426–430.

Linn, Marcia C and John Dalbey (1989). “Cognitive consequences of programming instruction”. In: *Studying the novice programmer*, pp. 57–81.

Marceau, Guillaume, Kathi Fisler, and Shriram Krishnamurthi (2011). “Mind your language: on novices’ interactions with error messages”. In: *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, pp. 3–18.

Marion, Bill et al. (2007). “Assessing computer science programs: what have we learned”. In: *ACM SIGCSE Bulletin*. Vol. 39. 1. ACM, pp. 131–132.

Minaei-Bidgoli, Behrouz et al. (2003). “Predicting student performance: an application of data mining methods with an educational web-based system”. In: *Frontiers in education, 2003. FIE 2003 33rd annual*. Vol. 1. IEEE, T2A–13.

Orr, Dominic, Christoph Gwosć, and Nicolai Netz (2011). *Social and economic conditions of student life in Europe: synopsis of indicators; final report; Eurostudent IV 2008-2011*. W. Bertelsmann Verlag.

Papancea, Andrei, Jaime Spacco, and David Hovemeyer (2013). “An open platform for managing short programming exercises”. In: *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, pp. 47–52.

Parsons, Dale and Patricia Haden (2006). “Parson’s programming puzzles: a fun and effective learning tool for first programming courses”. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., pp. 157–163.

Pea, Roy D (1986). “Language-independent conceptual “bugs” in novice programming”. In: *Journal of Educational Computing Research* 2.1, pp. 25–36.

Perkins, David N et al. (1986). “Conditions of learning in novice programmers”. In: *Journal of Educational Computing Research* 2.1, pp. 37–55.

Petersen, Andrew, Jaime Spacco, and Arto Vihavainen (2015). “An exploration of error quotient in multiple contexts”. In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, pp. 77–86.

Piech, Chris et al. (2012). “Modeling How Students Learn to Program”. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE ’12. Raleigh, North Carolina, USA: ACM, pp. 153–

160. ISBN: 978-1-4503-1098-7. DOI: `10.1145/2157136.2157182`. URL: `http://doi.acm.org/10.1145/2157136.2157182`.

Pintrich, Paul R et al. (1993). "Reliability and predictive validity of the Motivated Strategies for Learning Questionnaire (MSLQ)". In: *Educational and psychological measurement* 53.3, pp. 801–813.

Porter, Leo, Daniel Zingaro, and Raymond Lister (2014). "Predicting student success using fine grain clicker data". In: *Proceedings of the tenth annual conference on International computing education research*. ACM, pp. 51–58.

Prior, Julia Coleman and Raymond Lister (2004). "The backwash effect on SQL skills grading". In: *ACM SIGCSE Bulletin* 36.3, pp. 32–36.

Prior, Julia R (2014). "AsseSQL: an online, browser-based SQL skills assessment tool". In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, pp. 327–327.

Quadri, Mr MN and NV Kalyankar (2010). "Drop out feature of student data for academic performance using decision tree techniques". In: *Global Journal of Computer Science and Technology* 10.2.

Raadt, Michael de et al. (2005). "Approaches to learning in computer programming students and their effect on success". In: *Proceedings of the 28th HERDSA Annual Conference: Higher Eduation in a Changing World (HERDSA 2005)*. Higher Education Research and Development Society of Australasia (HERDSA), pp. 407–414.

Ramaswami, M and R Bhaskaran (2010). "A CHAID based performance prediction model in educational data mining". In: *arXiv preprint arXiv:1002.1144*.

Ramesh, V, P Parkavi, and K Ramar (2013). "Predicting student performance: a statistical and data mining approach". In: *International journal of computer applications* 63.8.

Rist, Robert S (1995). "Program structure and design". In: *Cognitive Science* 19.4, pp. 507–562.

Rodrigo, Ma Mercedes T et al. (2009a). "Affective and behavioral predictors of novice programmer achievement". In: *ACM SIGCSE Bulletin* 41.3, pp. 156–160.

Rodrigo, Maria Mercedes T et al. (2009b). "Analyzing online protocols to characterize novice Java programmers". In: *Philippine Journal of Science* 138.2, pp. 177–190.

Rogalski, Janine and Renan Samurçay (1990). "Acquisition of programming knowledge and skills". In: *Psychology of programming* 18.1990, pp. 157–174.

Rountree, Nathan, Janet Rountree, and Anthony Robins (2002). "Predictors of success and failure in a CS1 course". In: *ACM SIGCSE Bulletin* 34.4, pp. 121–124.

Rountree, Nathan et al. (2004). “Interacting factors that predict success and failure in a CS1 course”. In: *ACM SIGCSE Bulletin*. Vol. 36. 4. ACM, pp. 101–104.

Ruthmann, Alex et al. (2010). “Teaching computational thinking through musical live coding in scratch”. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, pp. 351–355.

Shah, Anuj Ramesh (2003). “Web-cat: A web-based center for automated testing”. PhD thesis. Virginia Tech.

Sheard, Judy et al. (2008). “Performance and progression of first year ICT students”. In: *Proceedings of the tenth conference on Australasian computing education-Volume 78*. Australian Computer Society, Inc., pp. 119–127.

Simon et al. (2006). “Predictors of success in a first programming course”. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., pp. 189–196.

Simon, Beth et al. (2008). “Common sense computing (episode 4): Debugging”. In: *Computer Science Education* 18.2, pp. 117–133.

Sleeman, D et al. (1984). “Pascal and High-School Students: A Study of Misconceptions. Technology Panel Study of Stanford and the Schools. Occasional Report# 009.” In:

Soloway, Elliot and Kate Ehrlich (1984). “Empirical studies of programming knowledge”. In: *IEEE Transactions on Software Engineering* 5, pp. 595–609.

Sorva, Juha and Teemu Sirkiä (2011). “Context-sensitive guidance in the UUhistle program visualization system”. In: *Proceedings of the 6th Program Visualization Workshop (PVW’11)*, pp. 77–85.

Spacco, Jaime et al. (2006). “Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses”. In: *SIGCSE Bull.* 38.3, pp. 13–17. ISSN: 0097-8418. DOI: 10.1145/1140123.1140131. URL: http://dx.doi.org/10.1145/1140123.1140131.

Spacco, Jaime et al. (2015). “Analyzing Student Work Patterns Using Programming Exercise Data”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, pp. 18–23. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677297. URL: http://dx.doi.org/10.1145/2676723.2677297.

Spohrer, James C and Elliot Soloway (1986). “Novice mistakes: Are the folk wisdoms correct?” In: *Communications of the ACM* 29.7, pp. 624–632.

Spohrer, James C, Elliot Soloway, and Edgar Pope (1985). “A goal/plan analysis of buggy Pascal programs”. In: *Human–Computer Interaction* 1.2, pp. 163–207.

Stein, Michael V (2002). “Mathematical preparation as a basis for success in CS-II”. In: *Journal of Computing Sciences in Colleges* 17.4, pp. 28–38.

Thai-Nghe, Nguyen, Andre Busche, and Lars Schmidt-Thieme (2009). "Improving academic performance prediction by dealing with class imbalance". In: *2009 Ninth International Conference on Intelligent Systems Design and Applications*. IEEE, pp. 878–883.

Tillmann, Nikolai et al. (2014). "Code hunt: Searching for secret code for fun". In: *Proceedings of the 7th International Workshop on Search-Based Software Testing*. ACM, pp. 23–26.

Tonin, Neilor A, Fabio A Zanin, and Jean Luca Bez (2012). "Enhancing traditional algorithms classes using URI online judge". In: *e-Learning and e-Technologies in Education (ICEEE), 2012 International Conference on*. IEEE, pp. 110–113.

Tonin, Neilor Avelino and Jean Luca Bez (2013). "Uri online judge: A new interactive learning approach". In: *Computer Technology and Application* 4.1.

Tukiainen, Markku and Eero Mönkkönen (2002). "Programming aptitude testing as a prediction of learning to program". In: *Proc. 14th Workshop of the Psychology of Programming Interest Group*, pp. 45–57.

Turkle, Sherry and Seymour Papert (1992). "Epistemological pluralism and the revaluation of the concrete". In: *Journal of Mathematical Behavior* 11.1, pp. 3–33.

Ventura Jr, Philip R (2005). "Identifying predictors of success for an objects-first CS1". In:

Vihavainen, Arto (2013). "Predicting students' performance in an introductory programming course using data from students' own programming process". In: *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*. IEEE.

Vihavainen, Arto, Jonne Airaksinen, and Christopher Watson (2014a). "A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success". In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. Glasgow, Scotland, United Kingdom: ACM, pp. 19–26. ISBN: 978-1-4503-2755-8. DOI: `10.1145/2632320.2632349`. URL: `http://doi.acm.org/10.1145/2632320.2632349`.

— (2014b). "A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success". In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. Glasgow, Scotland, United Kingdom: ACM, pp. 19–26. ISBN: 978-1-4503-2755-8. DOI: `10.1145/2632320.2632349`. URL: `http://doi.acm.org/10.1145/2632320.2632349`.

Vihavainen, Arto et al. (2013). "Scaffolding students' learning using Test My Code". In: *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, pp. 117–122.

Watson, Christopher, Frederick WB Li, and Jamie L Godwin (2013). "Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior". In: *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*. IEEE, pp. 319–323.

— (2014). "No tests required: comparing traditional and dynamic predictors of programming success". In: *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, pp. 469–474.

Werth, Laurie Honour (1986). *Predicting student performance in a beginning computer science class*. Vol. 18. 1. ACM.

Wiedenbeck, Susan, Deborah Labelle, and Vennila NR Kain (2004). "Factors affecting course outcomes in introductory programming". In: *16th Annual Workshop of the Psychology of Programming Interest Group*, pp. 97–109.

Winslow, Leon E (1996). "Programming pedagogy—a psychological overview". In: *ACM SIGCSE Bulletin* 28.3, pp. 17–22.

Worsley, Marcelo and Paulo Blikstein (2013). "Programming Pathways: A Technique for Analyzing Novice Programmers' Learning Trajectories". In: *International Conference on Artificial Intelligence in Education*. Springer, pp. 844–847.

Yudelson, Michael et al. (2014). "Investigating Automated Student Modeling in a Java MOOC". In: *Proceedings of The Seventh International Conference on Educational Data Mining 2014*.

Zingaro, Daniel et al. (2013). "Facilitating Code-writing in PI Classes". In: *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: ACM, pp. 585–590. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445369. URL: http://dx.doi.org/10.1145/2445196.2445369.