THAYER SCHOOL OF
ENGINEERING
AT DARTMOUTH

HELMUT SCHMIDT
UNIVERSITÄT
Universität der Bundeswehr Hamburg

A DATA-DRIVEN APPROACH TO IMPROVE THE TEACHING OF PROGRAMMING

A Thesis
Submitted to the Faculty of the
Helmut Schmidt University
in partial fulfillment of the requirements for the
degree of

Master of Science

by

ALISAN ÖZTÜRK

July 24, 2017

Prof. Dr. Petra Bonfert-Taylor
Thayer School of Engineering
Dartmouth College
Hanover, NH, USA

Prof. Dr. Armin Fügenschuh
Helmut Schmidt University
University of the Federal Armed Forces
Hamburg, Germany

Declaration of Authorship

I, Alisan Öztürk (matriculation no. [redacted] hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I certify that the thesis in digital form is identical to the thesis in printed form.

Hanover, NH, USA, ...

Date Signature

Abstract

In today's IT-driven world, programming skills are indispensable for engineers. While there are different approaches to teaching students how to code, it is useful regardless to have meaningful data on student performance, for example in order to determine early on which students could benefit from extra help. At the Thayer School of Engineering, we have developed a platform- independent in-browser programming environment which allows students to focus on the basics of programming without the distractions of having to deal with complicated editors or servers. This environment is additionally linked to students' accounts and can be embedded in our Learning Management System, where students must complete daily assignments. All data of the students' interactions with the platform are captured, resulting in a unique data set from a full on-campus introductory programming course. Based on this data set, we evaluated various metrics that quantify a student's error compilation behavior and showed that they can be implemented in a different context and be of use as students' performance metrics. In addition, we found out that small programming assignments which cover the basic concepts of programming are essential for students' success. Linked to these assignments, we present two metrics which show that students who spend more time on average on these assignments, as well as solve fewer assignments, are more likely to perform worse on the course. In combination with the error compilation metrics, we built multiple regression models that performed superior to individual models. Furthermore, we implemented machine learning algorithms that could accurately predict at-risk students with up to seven days of intervention time left before the first midterm exam. Based on our results, we can show that a data-driven analysis in an introductory programming course can be utilized to assist both, students that are struggling to pass the course and educators in getting insights on student learning.

Acknowledgements

First and foremost, I want to thank my advisor Professor Petra Bonfert-Taylor for being patient, giving advice and spending countless hours on discussions. Thank you for giving me the freedom to explore many directions and assisting me in every aspect within this project. Working with you was a wonderful and exciting experience.

I want to acknowledge Professor Armin Fügenschuh who toke every chance to promote my academic career within the last four years. It was a challenging and exciting experience and I am thankful for every opportunity I received through your everlasting support.

Furthermore, I want to thank Ben Servoz who is the mastermind behind the coding environment and always spent an extra minute listening to my requests and issues and spending that extra time on trying to implement every feasible request. Our enriching discussions helped me exploring important areas and had a major influence on shaping this thesis.

I am very grateful for Ted and Mavra who spent hours on discussing this work as well as providing me with an incredible, differentiated and helpful view.

A special gratitude goes to Kelli, Meghan, and Monica for the proofreading of my thesis and for the honest and helpful advice.

I also want to thank the Thayer School of Engineering and everyone involved who made this exchange possible and unbelievable fascinating.

Finally, I want to give my gratitude to my family for their unbreakable support for more than two decades and to my fiancée for encouraging me to pursue this exchange and always supporting me on this path.

Thank you.

Contents

List of Tables

List of Figures

List of Abbreviations

AIC	Akaike's Information Criterion	26
API	Application Program Interface	44
AUC	Area Under Curve	21
EDM	Educational Data Mining	2
EQ	Error Quotient	11
ID	Identification	46
IDE	Integrated Development Environment	8
IFrame	Inline Frame	47
IT	Information Technology	4
ITS	Intelligent Tutoring System	11
KDD	Knowledge Discovery from Data	4
LMS	Learning Management System	1
MOOC	Massive Open Online Course	5
OLS	Ordinary Least Squares	24
RED	Repeated Error Density	12
RFE	Recursive Feature Elimination	16
ROC	Receiver Operating Characteristic	21
RR	Robust Relative	30
SMOTE	Synthetic Minority Over-sampling Technique	17
TA	Teaching Assistant	2

Chapter 1

Introduction and Purpose

In the age of information technology, a basic understanding of programming becomes increasingly important for a broader range of persons. This student demand of programming skills leads to an increase in the size of introductory programming classes, which are additionally filled with students having different backgrounds, including non-technical ones. At the same time, the difficult nature of programming is unchanged, resulting in more students having problems following and higher student dropout rates. Educators have yet to come up with better pedagogical strategies to prevent this attrition.

Novice programmers have to simultaneously master two quite disparate tasks: They need to master the syntax of the programming language while also acquiring the fundamental concepts of programming. There is thus a strong need to reduce any additional barriers to learning. Depending on the working environment, they may face additional challenges just to start coding, such as running compiler commands on their source code.

There are currently a number of different approaches for trying to understand how novice programmers learn to code. Through the use of an online Learning Management System (LMS) as well as through the use of an online coding envi-

ronment, a data-driven analysis is possible. This might allow for the detection of students prone to dropping out so that the instructor or Teaching Assistants (TAs) can intervene and help the students before it is too late. Being able to identify and assist struggling students also benefits students who are not struggling, because those can be assigned individual and more challenging problems. Therefore the course can be of increased value for all of the students.

In order to keep providing personalized feedback to students, especially in large introductory programming courses, researchers provided various data-driven methods to assist instructors. These methods include enhanced error messages and auto-grader systems and have been largely applied as well as evaluated in either distance learning courses or laboratory environments. In contrast, the data set used in this thesis resulted from a traditional on-campus course with lecture hall seminars backed by an online LMS and a server-sided online coding environment. Although the major online LMSs provide their own analytics features, such as page views or participation in discussions, those only touch the surface of possible analyses, because the data is already aggregated. The instructors are not provided with personalized and helpful dashboards which then can be used to assist in decision making. These general aggregations result from the fact that an LMS serves as a platform for a diverse range of courses. Instead there is a need to present student data in a meaningful way in order for the instructor to receive important signals without having to grind through all the available data. Applying techniques of Educational Data Mining (EDM) allows for the identification of relevant data followed by a practical evaluation of the course to aggregate the data customized to the domain of the course.

This thesis proposes a number of methods to analyze and improve an introductory programming course at the Thayer School of Engineering at Dartmouth, based on our course setting and the data collected therein. A system of enhanced error

messages and an new auto-grader approach are implemented as well as evaluated. The currently discussed metrics, that are promising in predicting at-risk students, are evaluated. Furthermore, time dependent attributes, within this context, are extracted and analyzed based on whether they can be used as metrics to predict a student's performance.

The thesis starts with a brief definition of educational data mining in Chapter 2 and a literature review in Chapter 3 on current approaches that are trying to achieve the outlined goals. The literature review is followed by a description in Chapter 4 of the methods used and a detailed discussion in Chapter 5 of the later implemented student performance metrics. After that, Chapter 6 introduces the course setting and the coding environment. The main part of the thesis is the analysis in Chapter 7, which includes the steps from data preparation to interpretation of the results. Finally, the limitations of the approach are discussed in Chapter 8 and the thesis ends with a conclusion and possible paths for future work.

Chapter 2

Educational Data Mining

This chapter introduces the terms data mining and Educational Data Mining (EDM) and highlights the areas on which this thesis builds.

2.1 Data Mining

As a result of larger and faster ways to store data and through the rapid growth of Information Technology (IT)-software in almost every possible area, from consumer wearables to production processes in industry, the collection of data is growing immensely. This leads to new challenges in how to handle this data, but also to great opportunities when the right tools are applied to extract useful and valuable information. One such tool is data mining, which is also known as Knowledge Discovery from Data (KDD). KDD is a more appropriate term, because it is not the individual data that are of value, but rather the information and knowledge that can be extracted through an analysis. Therefore data mining can be described as a process of discovering patterns and knowledge from large amounts of data. Data is not typically sitting somewhere and waiting to be analyzed, but rather needs to be collected, cleaned and transformed among other things, which involves several steps and preparation [1].

2.2 Data Mining in Education

EDM is a collection of a broad range of methods and applications that transitioned from data mining in other areas to an educational context.

EDM evolved through the rise of online LMSs as well as the fact that Massive Open Online Courses (MOOCs) generate huge amounts of different kinds of data. This creates an opportunity to process and make use of the data. Because the field of EDM is relatively young, current research spans many different questions and areas. The definition by the *International Educational Data Mining Society* provides the most recent explanation of the objectives:

> "Educational Data Mining is an emerging discipline, concerned with developing methods for exploring the unique and increasingly large-scale data that come from educational settings, and using those methods to better understand students, and the settings which they learn in." [2]

Additionally [3] categorizes research in EDM in *Prediction*, *Clustering*, *Relationship Mining*, *Distillation of data for human judgment* and *Discovery with models*. This thesis can be classified as EDM work in prediction as it applies methods in an effort to predict students' course achievements based on a single explanatory variable or a combination of those. At the same time, available attributes from the data set are studied as to whether they can serve as reliable predictors in the given context, too.

Chapter 3

Literature Review

This chapter illustrates the different directions and approaches in EDM research and outlines the immense number of possibilities to make use of the data generated. Starting with an overview of general approaches that are possible in an EDM-context, this literature review focuses on studies that explore data sets from introductory programming courses. The studies develop metrics to measure student performance and ultimately allow the recognition of students that are prone to drop out of a course.

Since EDM is a relatively young research area, most relevant publications are from within the last decade. [4] provides a comprehensive review of important work published through 2009. It also reports an exponential growth of relevant publications since 2000 based on number of citations. According to [3], most work between 1995 and 2005 involved relationship mining with a share of 43%, followed by prediction and exploratory data analysis and clustering. The fields focus shifted as stated by [3] to prediction with a share of 42%, based on the publications from 2008 to 2009. The shift was followed by a new method called discovery with models as well as exploratory data analysis and clustering.

Looking only at EDM literature that focuses on the domain teaching and learning of programming, [5] contains an extensive literature review analyzing relevant work between 2005 and 2015. In this special domain a rise in relevant EDM work was discovered as well, especially when looking at the number of papers published through the *International Conference on Educational Data Mining*. This number of publications rose from 17 papers in 2008 to 91 in 2015. The reviewed literature in [5] is categorized into work on:

- Students,
- programming and
- learning environments.

Student-focused research aims to predict student's performance, study student affective states - such as programming related confusion and frustration - and estimate student knowledge. Programming focused research aims to identify programming behavior, strategies and errors made. Lastly, studies on learning environments aim to find tools for instructors and different mechanisms for automated testing, grading, and feedback of programming assignments.

This thesis includes work on predicting student performance, an analysis of the errors encountered and the benefits of having that information, as well as work on automated testing and feedback for students when solving small programming assignments. The following sections focus on the literature describing compiler error messages and predicting student performance.

3.1 Improving Compiler Error Messages

A compiler translates source code from a high-level programming language, such as *C*, in a format that can be executed by a computer. If the syntax of the source

codes is incorrect then the compiler cannot translate it. To support programmers in finding and correcting the syntax, compilers provide error messages that try to pinpoint the location of the mistake. These are called *compiler error messages*.

There are multiple possible reasons of why introductory programming courses are struggling with high drop out rates. One problem that has gained attention in the past is that of improper compiler error messages. In addition to learning the fundamentals of programming, novice programmers also have to master the specific syntax of the programming language they are learning. If students get additionally confused by cryptic compiler error messages while being in the process of producing a syntactically correct program, it can lead to student frustration.

Besides raising frustration and potentially lowering retention, these compiler messages force novice students to waste a significant amount of time to correct simple syntax errors or require the help of instructors [6].

Compiler error messages aim to help the user correct the syntax, therefore it is indeed important for novice programmers to learn how to utilize them. However, the error messages are written by compiler developers and can mislead even advanced programmers, too. Moreover, error messages often fail to narrow down the root cause of the failed compile attempt, making the debugging process even more difficult. While Integrated Development Environments (IDEs) have gained a lot of attention in order to assist programming, compiler error messages have not [7].

However, the customizability of IDEs and online coding environments allows for the implementation of one's own tools to help address these issues. When trying to find explanations for compiler errors that guide a student through the debugging process, it is important to be general enough to capture all recurrent problems while also providing enough details to help a student in that moment. Incorrectly phrased explanations or those that are not general enough can lead to increased confusion.

Because it is not trivial to find a helpful explanation for an error message and not all errors are encountered frequently, using data-driven tools to identify the most frequent errors based on the application helps narrow the list of needed explanations down to a few. By implementing a system that aggregates error occurrences, [8] showed that the most frequent errors identified by experts were different from the actual errors that the students encountered. Even results of studies on error messages for the same language may differ: Although there are definitely errors that occur in every context frequently, the overall distribution depends heavily on the context - the assignments the students are working on and their programming experience. With currently available software and hardware, it is easy to aggregate error data from ones own environment and make use of it.

The work in [9] goes a step beyond a generalized explanation and includes prepossessing of the student's code to further improve the enhanced error message by customizing it using part of the student's source code. The system was furthermore evaluated using a control group showing a significant reduction in both number of errors and number of unique errors coupled with students that were less frustrated by compiler errors in the test group.

While enhanced compiler messages can directly assist students, [10] reported that it also enables instructors and TAs to spend more time on helping students with more difficult problems, instead of having to spend most of their time helping correct common syntax errors.

Going one step further, [11] implemented a system called *HelpMeOut* that stores errors and their fixes from other students in the course and presents those to students as a hint when they encounter an error that is in the database. Additionally that system also displays further information in the form of text to help correct that error. While [11] implemented that system in a Java environment, a separate study was mentioned which used the system in a C/C++ context. The study also showed

that, especially for errors encountered in the C programming language, the root cause of the compiler error is often hard to map to the displayed error message, making it impossible to provide appropriate assistance for some errors to the student, where the *HelpMeOut* system would be more effective, showing the parts of the source code that caused the error and the corresponding fix that corrected the error.

3.2 Predicting Student Performance

While there is work in predicting students' academic performance based on their admission data as in [12], this thesis work is focused on student performance in introductory programming classes. As a result of the growing popularity of LMSs, researchers shifted their focus towards LMS data and applying EDM-techniques to help course instructors in identifying at-risk students by providing relevant information mined from student platform usage data. [13]

Researchers aim to find hidden attributes in student data which can be used as predictors of their overall performance. This involves work that does not focus on LMS data, but rather uses features and attributes from demographic data - such as age, gender, educational background and many more. These data are not necessarily easily accessible but are useful for that task. [14] as well as [15] concluded that time dependent variables extracted from online LMSs are critical factors for online learning which allowed them to build models that could accurately predict students at-risk. Additionally [15] presented an early warning system that was integrated into the LMS and is executed at certain points to take further action after the identification.

In addition to demographic data, [16] used attributes of student's performance at specific check points to predict students that were at-risk before the halfway

point of the course. They compared the performance of different models using various attributes and the model using all available data performed better than that which used only demographic data.

Aside from using demographic data or LMS data, many studies use data which is directly related to the course material. There is a huge opportunity to gain insight into how students learn, if the platform they are interacting with to solve an assignment is generating fine-grained data of the interaction between the student and the problem.

This data enables the implementation of Intelligent Tutoring Systems (ITSs), that can guide a student through the learning process and give personalized feedback.

[17] describes the framework of such systems, specifying an inner loop and an outer loop: The inner loop assists a student on a given problem for instance with hints [18], whereas the outer loop provides the student with an appropriate assignment based on an estimator of their knowledge.

For programming courses, it is possible to build an environment, capable of capturing student interactions. Although there are differences in how such platforms have been integrated as outlined in Section 6.3, many studies followed the same goals by evaluating similar types of data. The resulting data that is gathered from online protocols is then analyzed to find which data or metric can be used as an indicator on whether a student is performing poorly.

Jadud analyzed the error compilation behavior [19] and proposed a metric called Error Quotient (EQ) that shows big differences in accuracy when applied across a wide range of different studies [20], [21], [22]. Despite the results being sensitive to specific context, the metric seems to have promising potential; therefore studies have evaluated it in their specific context. [23] introduced the *Watwin Score* to address the weaknesses of the EQ. The *Watwin Score* is also based on the error compilation behavior and shows an improvement of the predictive power. In [24],

the authors present a modified version of the *Watwin Score* by fine-tuning the parameters on a different data set. While the EQ has mostly been evaluated in Java programming courses, [25] compares it across different contexts, notably with data from *Python*- and *C*-courses. Lastly, [26] proposed a method called Repeated Error Density (RED) which evaluates a student's error compilation process with a more generic approach, in an attempt to reduce the context sensitivity. Although none of these metrics present a one-fits-all solution, [27] showed that data-driven metrics derived from a student's course participation are in general more accurate than traditional metrics, opening a promising new field. The error compilation metrics are discussed in more detail in Chapter 5.

Parallel to the efforts focused on creating student metrics that directly correlate with student performance, machine learning methods have become increasingly popular in building models that predict students at-risk using traditional variables such as demographic data [16], but also trying to build models on top of the generated metrics. [28] compared the performance of the EQ and the *Watwin Score* to a machine learning model that used features such as the average grade of the student based on their past courses, their major, but also the number of steps needed to solve specific programming assignments. The feature extraction process used in this study also revealed that information on age, gender, and past programming experience did not contribute to their model. Using the relevant features, the trained classifier was able to predict low-performing and high-performing students after the first week of programming with over 70% accuracy when evaluated on a separate data set. When evaluating the EQ and the *Watwin Score* on their data set, they found little to no correlation between both metrics and the final grade. It is also notable that the authors provided an in-depth description of their assignments, which had been flagged as important features. This information can be used to compose similar assignments and try to replicate some of the results in this paper.

One of the more recent works tried to replicate the above results in a different context, using data from different universities and different courses and showing at the same time that their neural network was superior to other algorithms [29]. The authors managed to largely replicate the results from [28] and showed again the high accuracy of the machine learning classifiers and their overall stability over several data sets compared to the performance metrics.

A different approach is to apply machine learning methods to a keystroke analysis. [30] tried to separate experienced programmers from novice programmers based on their typing patterns, while also trying to map the keystroke latency of relevant combinations - for example i and $+$ were extracted as relevant - to the student's exam performance.

Lastly, recent work using machine learning techniques to predict at-risk students compares different classifiers on two contrasting data sets [31]. One data set resulted from an online introductory programming course, while the other was from an on-campus course from the same university. Although the results seem very promising, the authors used features such as civil status and income alongside their performance on weekly activities and exams. Demographic data, besides being difficult to gather, also add unnecessary variance to the models because they vary not only between classes but also between universities. Moreover, using demographic features increases the difficulty to extract relevant features that could provide insights on how students learn to program.

It is apparent from this variety of contexts that the resulting data sets from almost every study in this domain are notably different. This leads to a need to replicate the studies and verify the results on external data sets. Hence, [5] tries to address these issues and proposes methods in how to collect data, build the study and present the results in order to perform the needed replication and reproduction studies. Details on this procedure are presented in the description of the context for the course setting of our study in Chapter 6.

Chapter 4

Methodology

Before starting with the analysis, the methods used are outlined in this chapter and the underlying basic concepts are explained. Since one major part of this thesis is to predict students' performances and classify at-risk students, the EDM methods for prediction and classification - mainly linear regression and logistic regression - are introduced. This chapter starts with a brief introduction to machine learning and the techniques applied in this thesis.

4.1 Machine Learning

Machine Learning is the process of using algorithms for recognizing and learning from patterns in data and applying these findings to new data to make predictions. Machine Learning is a form of artificial intelligence. Instead of explaining the structure of the patterns to a computer, the algorithms find the patterns on their own. Facial recognition, which is a trivial task for a human, was only made possible for computers through machine learning [32].

Machine Learning can mainly be separated into the two categories *supervised learning* and *unsupervised learning*. Supervised learning consists of machine learning algorithms that learn from a sample data set - training data - that consists of input

data - features - and the desired output - labels - in order to make predictions or classifications for new data. While the goal of a classification task is to predict a binary response, for example, the detection of fraudulent activities in credit card transactions, a prediction or regression task has a continuous outcome, such as the income of a student after graduation. Unsupervised learning on the other hand is a collection of clustering algorithms. The input data is not labeled with a class since the labels are not known in advance. An example of this is the segmentation of customers into groups with similar preferences.

This thesis applies supervised machine learning methods to predict students at-risk of dropping out of an introductory programming course. The analysis is done in *Python 3.6* and uses the *scikit-learn* library [33]. *scikit-learn* is an open source project and widely used in science and industry for implementing machine learning methods in *Python*.

4.1.1 Fitting a Machine Learning Model

For supervised learning, a data set consists of a matrix of predictors X and a vector of the corresponding response y. The different algorithms then need to be tuned to the specific data set by adjusting their parameters. A problem that can arise within this process is that of *overfitting*. The performance of an overfitted model on the training data is good, but when the model is tested on unseen data the performance is bad because the model is too closely adjusted to the training data. An example of an overfitted model is when there is a linear relationship between two variables, but a higher order polynomial function is used instead of a regression line to describe the relationship.

There are different steps to prevent overfitting a model. A first step is to split the data into a training set and a testing set. The classifier is trained on data from the training set and then evaluated on the testing set, where the testing set contains data

the classifier has not seen before. Another step to prevent overfitting is reducing the number of features in the model to a minimum - a process called *feature selection*. Once the features are selected and the model is trained, it has to be evaluated.

4.1.2 Feature Selection

The more features are used to build a model, the more complex the model becomes and higher the risk of overfitting gets. This is why selecting the relevant features is a necessary step for the training of a machine learning model. Features can be manually selected by ranking features via their individual correlation with the response variable, which selects features independently of the algorithm used to build the model. Apart from this, some algorithms rank features internally by assigning an importance measurement. This can be utilized in a first iteration as a feature selection method.

Another approach is to select features through multiple iterations. One method is called Recursive Feature Elimination (RFE) and was used in this thesis for feature selection. The implementation in *scikit-learn* is based on the algorithm proposed in [34]. The RFE builds a model based on a selected classifier in one iteration and eliminates the least important feature to build another, reduced model in the following iteration. This loop is executed continuously until the reduced model contains the desired number of features.

4.1.3 Imbalanced Data

When a model is trained on a data set where one class is underrepresented, the classifier can become less effective, although its prediction accuracy is very good. Given a data set of 90 samples of class 1 and 10 samples of class 0, a classifier can then choose to always predict class 1, resulting in an accuracy of 90%. This is an extreme

scenario, but the effects of imbalanced data still need to be addressed. Various methods can be applied, such as under-sampling and over-sampling. Under-sampling means removing samples of the over-represented class, which is only feasible when there is enough data. Over-sampling instead focuses on adding new data to the data set, either by replicating existing data or by synthesizing new data.

In data sets from introductory programming courses, students at-risk are fortunately underrepresented. Because the data set is limited, over-sampling is the method of choice. Two techniques of over-sampling have been applied in this thesis. The first technique is the so-called Synthetic Minority Over-sampling Technique (SMOTE) and synthesizes new data points that are close to k nearest neighbors of the minority class, with k being a starting parameter. The method prevents the algorithms from ignoring the minority class and allowing it to generalize better. [35] A *Python* package, supplementary to the *scikit-learn* package, called *imbalanced-learn* [36] is used to implement SMOTE in this work.

The second technique is an implementation in *scikit-learn* and allows the adjustment of a `class_weight` parameter that increases the importance of the desired class and penalizes the model when it misclassifies data belonging to that class.

4.1.4 Model Evaluation

Splitting a data set into a training and testing set is an initial step toward the prevention of overfitting and allows for a more accurate representation of the actual model performance. In unfavorable conditions, splitting the data can lead to states where only one class is represented in a subset. On the other hand, randomly splitting the data can also result in a scenario, where the model is overperforming on the testing set.

To prevent such issues, a better approach to evaluating the performance of a model is called *cross-validation*. The most popular version of cross-validation is

called *k-fold-cross-validation*, where k defines the number of similarly sized partitions - *folds* - of the data set. Common values for k are 5 or 10 folds. Once the data set is split into k folds, the model is trained on the union of $k-1$ folds and tested on the remaining fold. This process is repeated until every fold is used as the testing set, which results in k performance measurements that can be averaged. The main benefit of cross-validation is that the model is forced to be more generalize since it is tested on different slices of the data.

While there are various metrics to evaluate a classification model, the following metrics are used in the analysis of this thesis.

Confusion Matrix

A confusion matrix visualizes the performance of a model by displaying the predictions as well as the true values per class. An example of a confusion matrix for a binary classification problem is given in Fig. 4.1.

Figure 4.1: Confusion matrix.

		Predicted	
		Fail	Pass
Actual	Fail	4	10
	Pass	0	86

This is an example of an imbalanced data set that consists of 86 actual Positives (P) and 14 actual Negatives (N). The diagonal shows the correct predictions, 4 True Negatives (TN) and 86 True Positives (TP). Together with the incorrect predictions, 10 False Positives (FP) and 0 False Negatives (FN), different metrics can be used to extract the characteristics of the classifier.

Accuracy

A model's accuracy is the number of correct predictions divided by the total number of samples. It is calculated by:

$$\rho_0 = \frac{(TP + TN)}{P + N}. \tag{4.1}$$

In the above example, the classifier scores a 90% accuracy by favoring the dominant class. Given an imbalanced data set, the accuracy is not useful to evaluate a classifier.

Cohen's Kappa

Cohen's Kappa - or the kappa score - compares the observed accuracy ρ_0 with the expected accuracy ρ_e. It measures how well the predictions agree with the ground truth, which incorporates imbalanced data sets. A classifier that only predicts the dominant class and scores a high accuracy, gets therefore penalized by the kappa score. The kappa score is calculated by:

$$\kappa = \frac{\rho_0 - \rho_e}{1 - \rho_e}. \tag{4.2}$$

The expected accuracy is calculated by:

$$\rho_e = \frac{(TN + FN)}{P + N} \cdot \frac{(TN + FP)}{P + N} + \frac{(FP + TP)}{P + N} \cdot \frac{(FN + TP)}{P + N}. \tag{4.3}$$

The value range of κ is between -1 and +1, with $\kappa < 0$ indicating that the classifier less effective than by chance and $0.8 \leq \kappa \leq 1$ indicating almost perfect agreement. For the above example, the kappa score is $k = 0.41$, with the observed accuracy $\rho_0 = 0.9$ and an expected accuracy of $\rho_0 = 0.83$. The kappa score of $k = 0.41$ indicates only a moderate agreement.

Precision

Precision is the ability of the classifier not to label a negative class as positive and is calculated by:

$$\textit{precision} = \frac{TP}{(TP + FP)}. \tag{4.4}$$

For the above example, the precision score of the classifier is 0.896.

Recall

Recall, also referred to as *sensitivity* or *true positive rate*, is the ability to correctly classify all positive samples and is calculated by:

$$\textit{recall} = \frac{TP}{(TP + FN)}. \tag{4.5}$$

Because there are no false negatives in the above example, the classifier has a perfect recall score of 1.0.

Specificity

Specificity, also referred to as the *true negative rate*, is the ability to correctly classify all negative samples and is calculated by:

$$\textit{specificity} = \frac{TN}{(TN + FP)}. \tag{4.6}$$

In the above example, the classifier has a specificity score of 0.286. This indicates that the classifier has trouble to correctly classify negative samples.

F1-Score

The F1-score is the harmonic mean of precision and recall and is calculated by:

$$F1 = 2 \cdot \frac{(\textit{precision} \cdot \textit{recall})}{(\textit{precision} + \textit{recall})}. \tag{4.7}$$

In the above example, the F1-score of the classifier is 0.945, resulting from a high precision score and a perfect recall score.

All these metrics can also be calculated for each class label individually. In this thesis, students will either be classified as *Fail* or *Pass*, where *Pass* means that they passed the exam or the course. At-risk students are those who are classified as *Fail*. For a high-performing classifier, it is desirable to at least classify all at-risk students correctly. It would be acceptable to some degree to falsely flag some passing students as at-risk. It is then the instructor's choice to evaluate the flagged students. Therefore, a suitable classifier should have a high specificity.

Receiver Operating Characteristic

The Receiver Operating Characteristic (ROC) is a popular metric to evaluate a classifier. It is a curve that is created by plotting different values for the false positive rate against the true positive rate. The true positive rate is given in Eq. (4.5) and the false positive rate is equal to $1 - \textit{specificity}$. The different values are generated by using different thresholds for the classification. A threshold is the probability cut-off for the classification of a true prediction. The ROC visualizes how much better a model performs in comparison to a random classifier.

The ROC-characteristic can also be quantified by a metric called Area Under Curve (AUC) score. It has a value range between 0 and 1, where an $AUC > 0.5$ indicates a model that performs better than a random model. The AUC is insensitive to imbalanced data and therefore more useful as the accuracy score in this case.

Fig. 4.2 shows several ROC curves, ranging from a perfect classifier to a classifier that performs just as good as a random classifier:

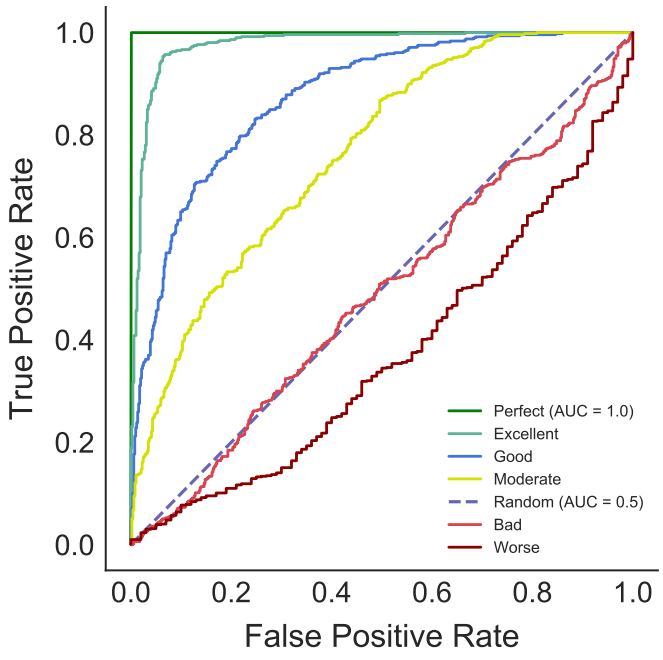


Figure 4.2: ROC curves.

4.2 Linear Regression

While there are many different regression models that can be used for prediction, linear regression was implemented in this thesis. Regression models are used to detect correlation between variables based on historic data, in order to obtain a function that can predict the value of a variable based on the input of one or multiple variables. The variable to be predicted is usually called *outcome* or *response* and is denoted as y, while the variables used to make the prediction are called *predictors* or *explanatory variables* and are denoted as x. The regression model is called *simple regression* if only one explanatory variable is used and is called *multiple regression* if there is more than one explanatory variable.

The simplest form of regression is *linear regression*, where the assumption is

made that the relationship between the explanatory variables and the response is linear. The model used in linear regression is a linear combination of the explanatory variables as seen in [37]:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k + \epsilon. \tag{4.8}$$

In the above equation β_0 is called the *intercept* and β_1 to β_k are the *slopes* of the linear relationship between the individual predictor variables and the response variable. Because the function of the regression hyperplane only estimates the data points and the real observations do not fall directly on that hyperplane, there is an error term ϵ which takes the random variance into account. For a single sample i out of n samples, the difference between the estimate $\hat{y}_i$ and the true value y_i is called *residual* ϵ_i, also known as an error term. For multiple regression models it is more convenient to use the matrix notation:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}. \tag{4.9}$$

For a simple linear regression model of the form $y = \beta_0 + \beta_1 x + \epsilon$, a regression model can be visualized as displayed in Fig. 4.3.

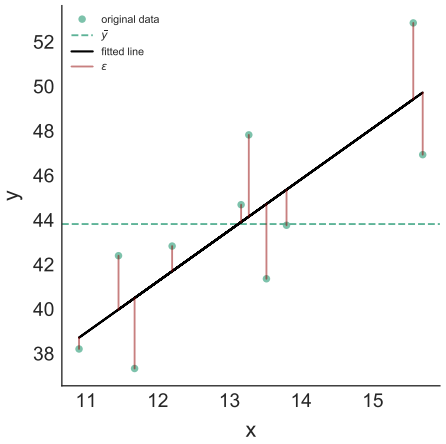


Figure 4.3: A simple linear regression model.

Here, a residual ϵ_i can be calculated by:

$$\epsilon_i = y_i - (\beta_1 x_i + \beta_0). \tag{4.10}$$

Since the parameters in $\boldsymbol{\beta}$ are unknown, they have to be estimated. The simplest and most common method for the estimation is called Ordinary Least Squares (OLS) regression. The OLS regression minimizes the sum of the squared residuals. The OLS criterion is, therefore:

$$S(\boldsymbol{\beta}) = \sum_{i=1}^{n} \epsilon_i^2 = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon} = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}). \tag{4.11}$$

Here, $\boldsymbol{\epsilon}^T$ is the transpose of $\boldsymbol{\epsilon}$. Finding the arg min of $S(\boldsymbol{\beta})$ gives the least-squares estimators in $\hat{\boldsymbol{\beta}}$ and the function $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}$ yields a point estimate of $\boldsymbol{y}$.

4.2.1 Model Evaluation

One way to evaluate a regression model is to analyze the variance of the model. The R^2 value is called the *coefficient of determination* and indicates the percentage of variance in $\boldsymbol{y}$ that can be explained by the explanatory variables $\boldsymbol{X}$. R^2 is calculated by:

$$R^2 = 1 - \frac{SS_E}{SS_T}. \tag{4.12}$$

Here, SS_E is the sum of the squared residuals and SS_T is the total sum of squares of the deviations from the mean. In Fig. 4.3 a deviation from the mean is the absolute distance of a point (x_i, y_i) to the horizontal line at $\bar{y}$. SS_T is calculated by:

$$SS_T = \boldsymbol{y^T y} - \frac{(\sum_{i=1}^{n} y_i)^2}{n}. \tag{4.13}$$

Since $0 \leq SS_E \leq SS_T$ we see that $0 \leq R^2 \leq 1$. When adding additional explanatory variables to the model, R^2 will never decrease. If there is a correlation between the new variables and the response, or by random chance, R^2 will increase and falsely indicate a better model fit. As a consequence, it is advised to use the *adjusted* R^2 - which is calculated by Eq. (4.14) - as a metric for multiple regression models instead, because R^2_{adj} will only increase with an additional variable when this variable reduces the residual mean squared. In other words, R^2_{adj} will only increase when more information is added to the model:

$$R^2_{adj} = 1 - \frac{SS_E/(n-p)}{SS_T/(n-1)}. \tag{4.14}$$

Here p is the number of the predictors and n is the sample size. It is evident that $SS_T/(n-1)$ is constant and not affected by how many explanatory variables are in the model, and so R^2_{adj} will decrease when added variables do not reduce the residual mean square. This is helpful against overfitting the model because R^2_{adj} will penalize unhelpful variables.

Equally important to deciding of whether or not the regression model is significant is hypothesis testing [37]. For a simple regression model, this means testing the null hypothesis H_0 that the slope β_1 is equal to zero against the alternative hypothesis H_1 that the slope is not equal to zero. Failing to reject H_0 means that there is no linear relationship between x and y. For multiple regression H_0 means that every β equals zero. Rejecting it implies that at least one explanatory variable contributes significantly to the model. Testing the individual regression coefficients shows which variables contribute significantly to the model and which variables do not and therefore should be excluded. The test statistic for a random estimator

β_j which is necessary for a t-test is:

$$t_0 = \frac{\hat{\beta}_j}{\sqrt{\hat{\sigma}^2 C_{jj}}} = \frac{\hat{\beta}_j}{se(\hat{\beta}_j)}. \tag{4.15}$$

Here, $\hat{\sigma}^2$ is an unbiased estimator of σ^2 and C_{jj} is the diagonal element of $(X'X)^{-1}$ corresponding to $\hat{\beta}_j$. These can be used to calculate $se(\hat{\beta}_j)$, which is the standard error of the estimator β_j.

The null hypothesis is rejected if $|t_0| > t_{a/2,n-k-1}$ indicating, that there is a significant linear relationship between the predictor and the response.

Besides R^2_{adj}, a popular metric which is mainly used to compare models to each other is called the Akaike's Information Criterion (AIC). It is calculated as:

$$AIC = n \cdot \log\left(\frac{SS_E}{n}\right) + 2k. \tag{4.16}$$

Here k is the number of parameters in the model - including the intercept. The lower the AIC the better. It is reduced by a smaller SS_E and penalizes additional parameters.

4.3 Logistic Regression

Whereas there are many different available classifiers for a classification task and other studies focused on comparing these to each other, this thesis argues that the choice of the best classifier is highly dependent on the data, even within the same domain. Hence, it is more important to focus on the features selected by the classifier, instead of comparing different classifier performances to each other. In this thesis, logistic regression proved to be the most useful for this data set, hence, this section is limited to a brief introduction to logistic regression.

Although the name *logistic regression* can lead to the assumption that it is an-

other regression model, it is instead used for the classification of a binary response variable. The reason it is called logistic regression is, that a linear model is fit in the feature space. The result is the probability of belonging to the positive class and the classification is made based on a probability threshold. The probability that a set of features $\boldsymbol{X_i}$ from a sample i belongs to the class y is:

$$P(y = 1|\boldsymbol{X_i}) = \sigma(\boldsymbol{X_i^T} \cdot \boldsymbol{w} + c). \tag{4.17}$$

The linear term is given by $\boldsymbol{X_i^T} \cdot \boldsymbol{w} + \boldsymbol{c}$, with $\boldsymbol{w}$ representing the weights of the parameters, equivalent to the β terms in linear regression, and $\boldsymbol{c}$ representing the bias, equivalent to the intercept. σ is the logistics function, which is a *sigmoid* (s-shaped) function. It transforms values between $-\infty$ and ∞ to values between 0 and 1, the value range of probabilities. The logistics function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4.18}$$

An example logistic regression model that was built with only one explanatory variable x_1 and the scalar parameters $c = -0.14$ and $w_1 = 1.57$ is displayed in Fig. 4.4.

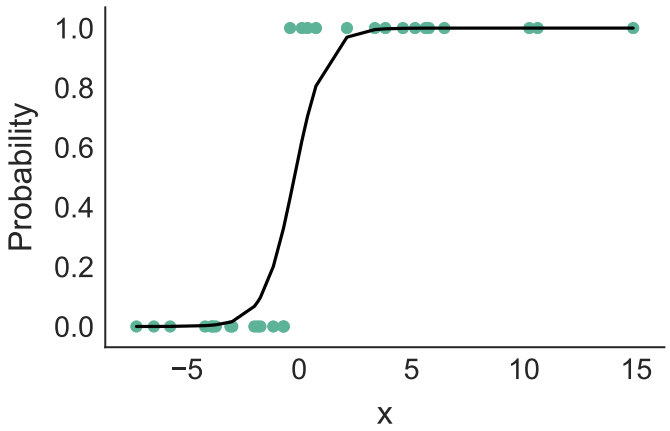


Figure 4.4: Example logistic regression model.

The probability that a sample with the value $x_1 = 0.12$ belongs to the class y can then be calculated by $\sigma(-0.14 + 1.57 \cdot 0.12) = 0.85$.

Because there is no closed solution to obtain the parameters $\boldsymbol{w}$ and $\boldsymbol{c}$, an approximation for the estimators can be obtained by maximizing the log-likelihood function or minimizing the negative log-likelihood function. The likelihood function for the logistics function is:

$$L(\boldsymbol{w}, \boldsymbol{c}) = \prod_{i=1}^{n} P(y = 1 | \boldsymbol{X_i}; \boldsymbol{w}, \boldsymbol{c}) = \prod_{i=1}^{n} \frac{1}{1 + e^{-y_i(\boldsymbol{X_i}^T \boldsymbol{w} + \boldsymbol{c})}}. \tag{4.19}$$

The objective function for the optimization is:

$$\min_{\boldsymbol{w},\boldsymbol{c}} \sum_{i=1}^{n} \log\left(1 + e^{-y_i(\boldsymbol{X_i}^T \boldsymbol{w} + \boldsymbol{c})}\right). \tag{4.20}$$

It is usually solved by Newton's method or by a gradient descent algorithm.

4.3.1 Regularization

In *scikit-learn* the logistic regression model is implemented by additionally including regularization parameters. It is a method to prevent overfitting and improve model generalization. The regularization strength is controlled by the parameter C, with lower values corresponding to more regularization and higher values corresponding to less regularization

The most common forms of regularization are *L1 regularization* and *L2 regularization*, both of which penalize the weight coefficients by adding a regularization term to the loss function, favouring smaller weights. L1 regularization adds the term $||\boldsymbol{w}||_1$ - the sum of the absolute values of the weights - to the loss function. It can be utilized as a form of feature selection because less useful features will be assigned a weight of zero, while others will be assigned relative large values.
L2 regularization adds the term $\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$ - adding the sum of the squared values - to

the loss function.

The main difference is that with L2 regularization, assigning weights of 0 does not give an advantage, but they can be close to zero. This means all features will be kept in the model. Furthermore, L1 regularization leads to a sparse output which is not differentiable. This increases the difficulty to solve the optimization problem.

Because feature selection is performed automatically, using the RFE algorithm, the logistic regression model built in this thesis is including L2 regularization. Thus, the final objective function is:

$$\min_{\boldsymbol{w},\boldsymbol{c}} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{n} \log\left(1 + e^{-y_i(\boldsymbol{X_i}^T\boldsymbol{w}+\boldsymbol{c})}\right). \tag{4.21}$$

Chapter 5

Student Performance Metrics

This chapter describes the three metrics of student performance that quantify the error compilation behavior of a student. The chapter begins with the Error Quotient (EQ) followed by the Robust Relative (RR) and the Repeated Error Density (RED). Finally, additional information about the implementation within the analysis is given.

5.1 Error Quotient

To date, the are no metrics that have been agreed upon, that are able to categorize, or at least describe, the performance of novice programmers. One metric that gained a lot of attention is the EQ, first published in [19]. It is an effort to try and map students' performance in an introductory programming course in Java to an error compilation behavior. The error compilation behavior is the way students edit and compile their source code to produce a syntactically correct program. The EQ assigns the student's error compilation behavior a value between 0 and 1. Having an EQ of 1 means every compilation event resulted in the same error, while having an EQ of 0 means the student has never encountered at least two sequential error events.

5.1.1 Algorithm

The algorithm to calculate the EQ needs a session of compilation events e_1 to e_n as input. In [19] a *session* included all compilations where neighbouring compilations were no more than ten minutes apart. Afterwards, steps one to four are executed for every session:

1. **Collate** - Create consecutive pairs from $(e_1, e_2), (e_2, e_3)$ to (e_{n-1}, e_n).
2. **Calculate** - Score each pair using a scoring algorithm.
3. **Normalize** - Divide the score of each pair by the maximum possible score.
4. **Average** - Take the average of the scores to get the EQ for that session.

The main step of the algorithm is to score each individual pair. To do so, Jadud [19] proposed the scoring algorithm shown in Fig. 5.1.

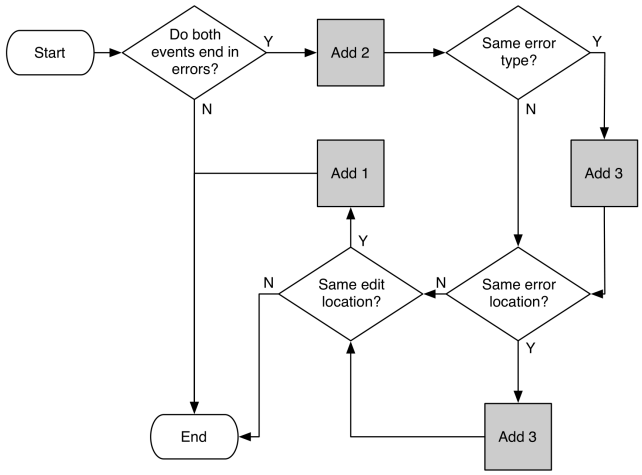


Figure 5.1: EQ scoring algorithm from [19, p. 116].

Jadud tuned the scoring algorithm based on a parameter analysis. If only one of the compilations results in an error, the score of that pair is 0. Otherwise the score increases when the two errors are the same, the errors occur in the same line (Jadud allowed this condition to be true for +/- 1 line) and when the edit location of the

two submissions is the same. In order to calculate the EQ, the coding environment needs to capture the student's snapshot data, which includes the source code as well as the compiler output that includes the detailed error message and location.

5.1.2 Verification

Having an EQ close to 1 can either mean that a student is having difficulty in writing syntactically correct programs or that the student has difficulty in understanding the error message in order to properly address the problem. Indeed, Jadud found a negative correlation between the EQ and the student's final exam grade. When correlating the EQ of a whole academic year - calculated from 24,852 compilation pairs - to the final exam grades he achieved an $R^2_{adj} = 0.267$ in his data set. When using only data from one term, consisting of 3,462 compilation pairs, the model only reached a $R^2_{adj} = 0.028$, indicating no correlation. One possible explanation in this context is that there is not sufficient data for that time span.

In an effort to verify the meaningfulness of the EQ, [38] conducted a study in a laboratory environment with 143 students. The students coded in a Java IDE called *BlueJ* which captured the necessary data. In total the students generated 28,386 compilation events over nine weeks. As there is a relatively strong correlation between a student's midterm exam performance and their final performance, [38] correlated the EQ to the midterm performance. The authors showed that calculating the EQ in a more specified context - the lab assignments - yields better results.

In [22], significant differences in the EQs between high-performing, average and at-risk students was revealed. A simple linear regression model was built to predict the midterm score with an $R^2_{adj} = 0.297$. However, the model was poorly suited for predicting at-risk students.

Additionally, Jadud ran a study in 2015 on the *BlueJ Blackbox* data set [21]. This data set grows out of the *BlueJ Blackbox Data Collection* project, which tries to col-

lect data in a large scale from anonymous users from all over the world to allow researchers to answer many different questions [39]. At the time of the study, there were almost 78,000,000 compilations recorded in the *blackbox* data set. From this data, [21] built a data by limiting the minimum number of compilations per user to 50 and specifying the time frame to 19 weeks. This resulted into a data set that contains 27,698 users and 10,193,432 compilations. Due to the nature of the data set, this study was not able to link the EQ to any kind of student performance, but instead tried to analyze the robustness of the EQ on a large data set. It showed that the EQ values for all users still resemble a normal distribution, but the distribution is slightly skewed to the left. Moreover, the study shows that there are differences between the means when looking at subsets based on different countries. Most importantly, it displays that the EQ is highly variable for the first compilation events. However, a steady-state or a slowly decreasing state is reached - when the student's error compilation behavior improves - for larger numbers of compilation events. This implies that a student needs to have a minimum number of compilation events in order for the EQ to become reliable.

Interestingly, Jadud used a different scoring algorithm in that study, which he also used previously in another study [40]. Fig. 5.2 shows the modified algorithm, which does not take the error and edit location into account.

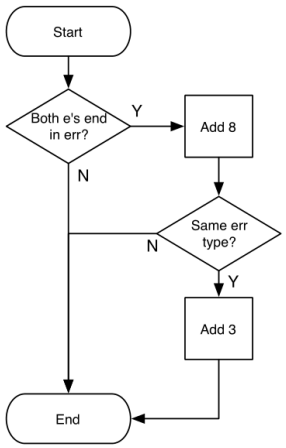


Figure 5.2: Modified EQ scoring algorithm from [40, p. 78].

In thesis we implemented the version that was presented first (see Fig. 5.1), in part because it was discussed in the literature more often.

5.2 Robust Relative Algorithm

There are not many new approaches for the development of data-driven metrics that can be used as a predictor of student performance, instead, other work has been trying to address the weaknesses of the EQ. Watson developed the *Watwin Score* in [23], which additionally considers how long the student needs to correct an error. The algorithm compares the error resolve time to all students who had previously corrected this type of error and penalizes the student's score based on whether the student is slower or faster. While this approach performed much better in their data set after the end of the course - $R^2 = 0.4249$ using the Watwin algorithm and $R^2 = 0.1922$ using the EQ Algorithm - the course setting is also different from the studies by Jadud. The Watson data set [23] consists of 45 students who had to solve assignments on lab computers in 14 sessions over the duration of 18 weeks. They generated 45,001 compilation pairings and indicated that Jadud might have created pairings of compilation events where students worked on different files, which would lead to false pairing events.

In [24], Watson presented an improved version of the *Watwin Score* and called it the RR algorithm. Both algorithms also quantify a student's error compilation behavior. Therefore the steps in both algorithms also include the generation of compilation pairs, the scoring of the pairings based on a scoring algorithm, the normalization of each score and building the average out of all normalized scores. The RR scoring algorithm is displayed in Fig. 5.3.

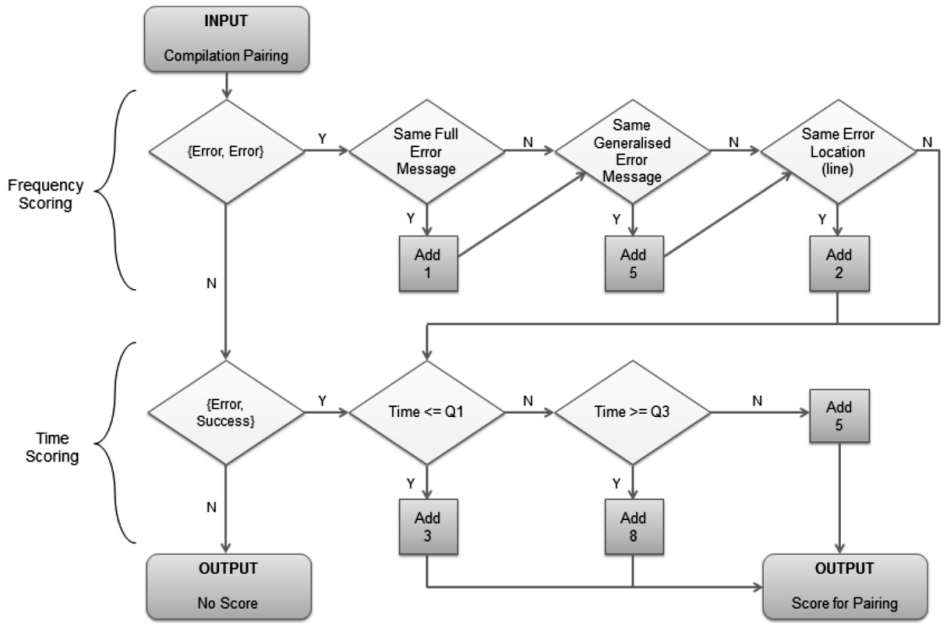


Figure 5.3: Robust Relative scoring algorithm from [24, p. 141].

There are two main differences between the initial Watwin algorithm and the RR. The first difference is a changed set of penalties. Instead of selecting the penalties based on a best fit within a single data set, the new penalties were generated with data from two different cohorts. The second difference is the comparison of a student's error resolve time to the resolve times of all students. The Watwin algorithm penalizes the score based on $\pm$ standard deviation from the mean resolve time, while the RR penalizes by using percentiles as shown in Fig. 5.3. Because the error resolve times are compared to a group of similar errors, the grouping method has a significant effect on the RR calculation. In this thesis, the errors were categorized as `parse errors`, `unbound names`, `incompatible types`, `redeclared names` and `others`, similar to [25].

5.3 Repeated Error Density

The Repeated Error Density (RED) is a metric that quantifies whether a student repeats the same error again and again, first introduced by Becker [26], since this is not reflected in the EQ. The author's intention was to present an alternative to the EQ that is less context dependent.

It does so by taking a sequence S of compilations from a student and examining it for repetition patterns of the same error. Given a sequence S, for example $S = ... \; xx \; ... \; xx \; ...$, where x is the occurrence of one particular error, the algorithm looks for strings of repeated errors s_i. In this case there are two strings of repeated errors each with one repetition r.

The RED is then calculated by:

$$RED = \sum_{i=1}^{n} \frac{r_i^2}{r_i + 1}. \tag{5.1}$$

In this equation n is the number of strings and $r_i + 1$ is the length of a string s_i containing r_i repeated errors. In the example above this would lead to $r_1 = r_2 = 1$ and therefore $RED = 1$. Becker implemented a system of enhanced error messages and reported a reduction of the RED and also the EQ within the intervention group in comparison to the control group.

It is unclear how the RED is calculated for a particular student. It is assumed, that the overall RED is the sum of all RED values for all sequences S over all error types. The author reported a possible weakness of the RED given larger data. When a student has more code submissions, it is likely to encounter more errors, which could inflate the RED. It was therefore stated that the RED would only work with small data sets. This thesis shows an implementation of the RED as outlined in this chapter followed by a comparison to the other student performance metrics.

5.4 Implementation

Comparing the different studies and approaches in which the EQ and the Watwin Score have been implemented, it is evident that there is a high variability in the outcome of the predictive capabilities of these metrics. This is due to many reasons. The most important difference is the data set itself. While most studies were conducted with data from introductory programming courses with novice programming students, the physical environment in which they coded and also the type of the coding assignments differ across the studies. Additionally, the way the data is prepared and the methods of implementation for the algorithms are not provided in enough detail to guarantee an identical replication. For the data cleaning process this involves excluding students with only a few submissions or those who did not work on a large enough number of unique assignments. Only Jadud mentioned that he excluded student sessions with less than seven files, without describing how he determined this cut-off [19].

In [25], the impact of the context on the performance of the EQ is analyzed by evaluating the metric on four different data sets that originated from different platforms and across various programming languages. As the EQ was developed and heavily tested on data from a Java course, in [25] it is evaluated on Python and C data sets, showing significant differences between contexts. The work suggests to instead adapt the data-driven metrics to fit the specific course setting. One necessary step to incorporate this into an individual metrics-driven algorithm is to find out which of the variables of the course setting, whether it is the programming language, the programming platform, the context of the assignments or even something else, affect the metric and to what degree.

In this thesis the three presented metrics, EQ, RR, and RED, were implemented and evaluated on all possible data, but also on specific slices of the data set, measuring the impact of the context on the overall model performance. Every problem the

students worked on is called an assignment and all code snapshots of one specific student-assignment combination are called a *submission package*, independent of the time and location the student submitted the snapshots. The metrics were calculated per submission package and then averaged. Different lengths of a submission package were evaluated and are also discussed in Chapter 7. Furthermore, it is important to note that the students learned the C programming language. To date, there has not been a metric tested on a data set within this context. Because of these differences, it is expected to see different results. Therefore the analysis aims for a transparent approach, allowing for replication of the results within similar enough contexts.

Chapter 6

The Course Setting and the Code Environment

As outlined in [5], there is a need for reproducibility in order to validate findings from different data sets and different research groups. In an effort to ameliorate this need, a detailed description of our data set is included in this chapter. This is followed by a brief explanation of Canvas - the LMS used with our course - and its analytics tools, as well as a detailed description of our online coding environment and its integration into Canvas.

6.1 Course Overview

The course is taught as an on-campus introductory C programming course and is accompanied by the school's online LMS. Throughout the term, students solve various coding exercises in different contexts. This results in a unique data set, the structure of which has to be considered in the analysis: Unlike most other studies, our data was not gathered in a laboratory environment, but in an on-campus class setting.

The students must prepare for each lecture beforehand, a process that includes watching brief video lessons and working through short quizzes that they must submit within the LMS. Those quizzes are mandatory and account for a small portion of the final grade. They mostly include short coding exercises and are solved in the LMS and a graded automatically by an auto-grader system. These quizzes are called *auto-grader* quizzes. Besides auto-grader quizzes, some quizzes also have a coding window integrated, but do not include a programming task to solve a problem. Instead they allow the student to practice the recently taught concepts and explore them further. This type of quiz is called a *try-out quiz*. Apart from the quizzes, students need to solve regular homework assignments, which are more complex. Similar to the quizzes, the homework assignments are unlocked at a specific time and need to be submitted before a deadline. For most of the homework assignments that involve coding, a coding environment without any pre-populated code is also provided.

In the past, students were connected to a server-side Linux environment in order to execute their code. They still do this in the first few homework assignments, and also for certain assignments, which results in some students preferring to use that environment. In past iterations of the course, students had to submit their source code for the homework assignments through a document upload in the LMS. Because this is the first time the students must use the coding environment as a main part of the course and the environment does not support managing deadlines and grading yet, the students still need to upload their solutions manually in the LMS.

During class, students also solve small programming exercises and can receive direct assistance. These are called *in-class* assignments and the students program in pairs most of the time. This is called *Pair-Programming*; it is a practice that originated in industry and recently found its application in an educational context. Students

work as a group of two on one computer and while one student is doing the actual coding, the other observes the process and discusses the actions taken by the programmer. These roles are frequently swapped and several studies have proven positive effects on learning when pair-programming is used in classrooms [41]. Because students code in pairs, however, the data assumed to only be linked to a single student is distorted, making some of it unsuitable for parts of the analysis. This data has to be excluded from the analysis.

In addition, students have to code a programming problem in a test setting, similar to a lab session - so called *gateway* assignments.

An overview of the different types of assignments and the number of coding assignments which were solved in the online coding environment up until the time of the first midterm and in total is given in Table 6.1.

Table 6.1: Number of the assignments within the coding environment.

Context	up to Midterm 1	Total
Quizzes (auto-grader)	8	14
Quizzes (try-out)	9	25
Homework	11	15
In-class	12	29
Gateway	1	3
Sum	41	86

The course demands a lot of effort from each student. Besides class attendance and exams, it requires students to prepare for class beforehand and also work on programming assignments in class. Furthermore, students work on projects, called *longer assignments*. For these, students do not code in the coding environment. Altogether, the final grade of the course is a composition of this effort and the breakdown of the weight of each assignment type is displayed in Table 6.2.
The course has a duration of 10 weeks, with two exams during the course and a final exam at the end of the course. The part of the course where students program

Table 6.2: Breakdown of the final grade.

Assignment	Weight
Quizzes	5%
Class Participation	5%
Homework	15%
Longer Assignments	25%
Midterm 1	15%
Midterm 2	15%
Final	20%

in C covers only seven weeks and starts after the first week of the course. This thesis' data set consists of the data collected from 93 students out of which five students did not finish the course.

6.2 Canvas

Canvas is an online LMS and assists instructors with the course. It allows the instructor to communicate with all course participants and distribute course material to them. It also enables the setting up of quizzes and assignments in advance, which can be hidden until a specific date and, once unhidden, remain accessible by the student until the submission deadline. That way, all students know what they have to do and until what date they have to submit it, as well as the amount of points they can earn for each individual assignment or quiz. The Canvas quizzes consist of a problem definition and can be extended with answer fields, multiple choice questions, or prompt students to upload their solutions in a specific format to Canvas. In this course, the quiz and homework assignments also provide a coding window from the online coding environment, integrating the coding assignments into Canvas which is clarified later in this chapter. Quiz assignments with automated feedback are mandatory and need to be solved to gain credit.

To assist with grading, Canvas has also integrated several tools that allow fast

and transparent grading. To some extent, direct comments on submissions are possible, when students submit `.pdf` or `.docx` files.

6.2.1 Canvas Analytics

Canvas has its own analytic tools that include an aggregated view of the student's platform activity data, separated into *Page Views* and *Participation*.

1. Page views: The number of pages a user clicked on.

2. Participation: An event created when the user submits or starts a quiz, submits an assignment, creates a wiki page, participates in discussions, joins a web conference or loads a collaboration to view/edit a document [42].

In addition, a boxplot is given that displays the points received by all students for each assignment. The instructor view is shown in Fig. 6.1.

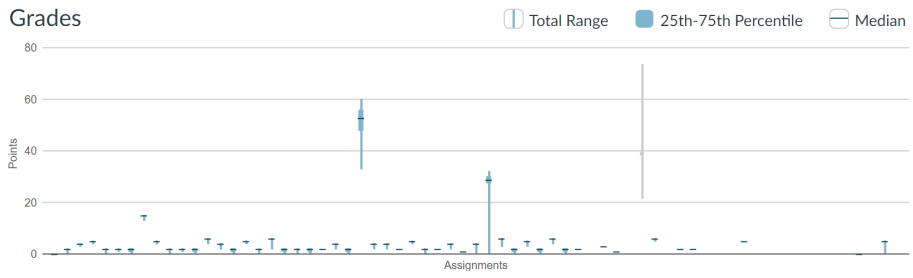
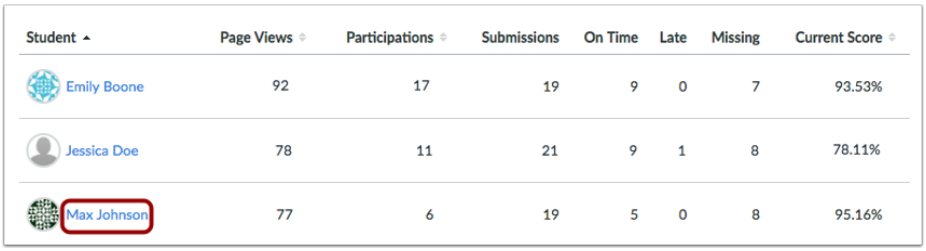


Figure 6.1: Canvas Analytics - Boxplot.

Lastly a table containing page view and participation data as well as information on whether the student submitted assignments on time are given for every student, an example with fictional student data is displayed in Fig. 6.2.



Student	Page Views	Participations	Submissions	On Time	Late	Missing	Current Score
Emily Boone	92	17	19	9	0	7	93.53%
Jessica Doe	78	11	21	9	1	8	78.11%
Max Johnson	77	6	19	5	0	8	95.16%

Figure 6.2: Canvas Analytics - Student table [42].

The category of submissions is the number of assignments that have been submitted by the student. The following three columns indicate how many assignments a student submitted before the deadline, after the deadline or missed submitting at all. At last, the student's current score is calculated by including all assignments that have been submitted and graded. When clicking on a student - as indicated in red - the identical plots are shown on that student - meaning the student's activity, communication activity, submission tree - and indicating the student's relative position.

While this is aggregated data, Canvas allows authorized users to make so called Application Program Interface (API) calls to also return finer grained data. Since API calls contain personal information, getting authorization is extremely restricted, but is necessary. To make use of the data - to build decision support tools or for research purposes - Canvas needs to create new API interfaces that would allow access to anonymized data, but linkable data. Beyond that, another obstacle is the fact that students who drop the course are removed from Canvas. This leads to an incomplete data set that of all things lacks the data from the students of interest. While it is possible to get aggregated data, it only provides a snapshot of the moment of the API call. As a consequence, an analysis comparable to that of the coding data cannot be easily executed for the Canvas data.

6.3 The "Code" Environment

Our "code" environment is an online embedded code editor that allows immediate compiling of source code written in either C or Python. The editor is an integration of the *Ace* project [43] and is written in JavaScript. The whole environment is part of the Thayer School's internal IT-ecosystem and any student from the college can access it with their regular student credentials. It was developed prior to this thesis and was used infrequently in past courses as an assisting tool. By the time this thesis was underway, there was already data accumulated from a past course. Part of this thesis was the full integration of the coding environment in the course setting, which demanded a rework of the auto-grader system and an adaptation of several coding assignments. In addition, a system of enhanced error messages - called friendly errors - was implemented. Fig. 6.3 shows a general view of the editor and how the students encounter the embedded coding window within Canvas. In this case, pre-populated source code was provided for the assignment.

Figure 6.3: Interface of the "code" environment.

The Ace code editor provides syntax highlighting but behaves otherwise like a normal text editor. A student can compile their code by pressing the *Run* button in the lower right corner. This creates a snapshot of the code, containing a unique Identification (ID) linked to the student, to the assignment and a time stamp. A server based compiler - wrapped in a safe environment - is then run on the code snapshot. This generates an output file and a compiler error file. If the compiler error file is not empty, the error will be displayed to the student on an overlay on the right side of the coding window. This is shown in Fig. 6.4.

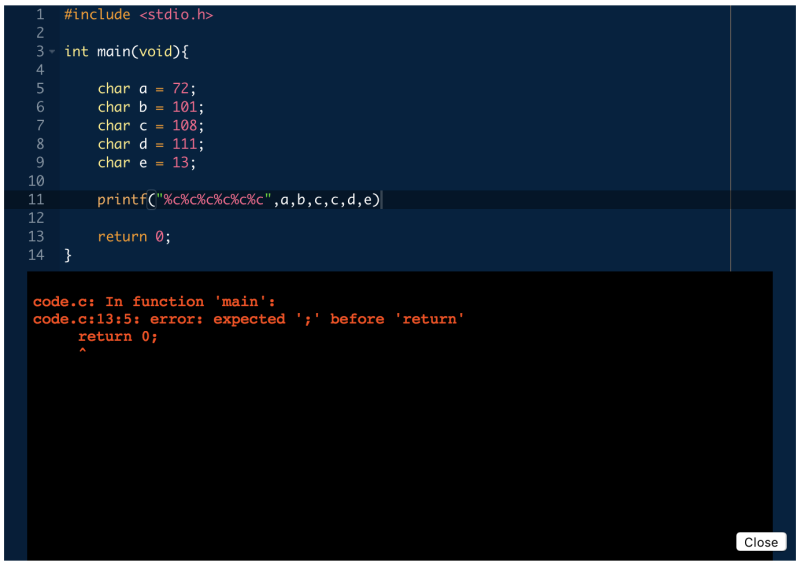


Figure 6.4: Display of a compiler error.

When the compiler runs successfully, the program output - if there is any - is displayed. An example is shown in Fig. 6.5.

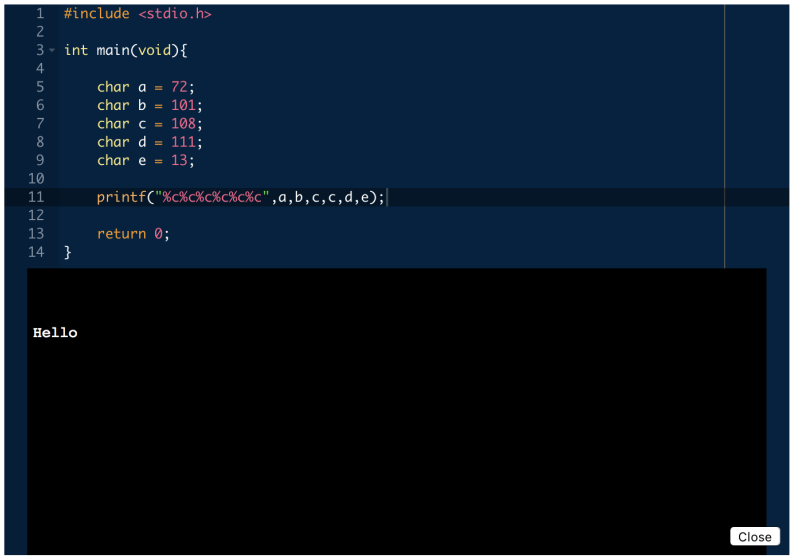


Figure 6.5: Display of program output.

6.3.1 Integration of the Coding Environment into the LMS

Only authorized users are able to connect to the coding platform. The authorization process with regular student credentials allows an identification of the user as well as a reliable way for students to log into the platform without prior registration. Because the IT-ecosystem has a running single sign-on solution, students are always automatically logged-in on the integrated coding environments within the LMS. This makes an additional authorization for the coding environment needless and the experience is seamless for students.

The in-browser coding environment is embedded into quizzes and assignments by using an Inline Frame (IFrame), which is a HTML document embedded into another HTML document. The platform then searches for previous student code and displays it, if it is found. This way every student has a personalized coding window for each coding problem and can resume previous work.

Each coding window can be pre-populated by the instructor with source code and deployed to students by using a query string in the form of `?starting_point`

= <NAME>. We call such pre-populated code windows *starting points*. Student's code is saved by a unique student ID and the name of the starting point. If no starting point with the given name is found, an empty coding window will be loaded. By using pre-populated code, the instructor can prepare quizzes and provide a skeleton which allows students to build on the code, starting from writing simple declarations and print statements to aggregating these in a bigger context.

When preparing for the upcoming class, students are introduced to new concepts by watching short video clips. Following this, students sometimes get a pre-populated coding window to try out the just seen concepts and programming principles to get familiar with them. Every assignment has a starting point, even if it is does not contain pre-populated code. On the one hand, this allows the student to return to his source code at any given time and allows to link the snapshot and log data to a specific assignment on the other hand.

6.3.2 Data Collection

As described in [5], from an EDM point of view data can be collected in different granularity levels, with keystrokes being the finest level. With the availability of cheap storage and the computational power of current server systems, there is no technical reason not to save all the data that can possibly be stored by the respective platform. Fig. 6.6 displays the different granularity levels, by which most EDM work concerning programming instruction can be categorized.

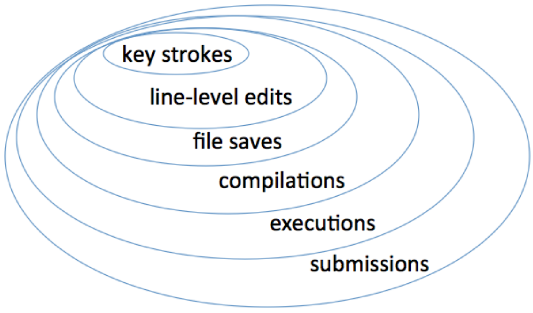


Figure 6.6: Granularity levels for data collection [5, p. 8].

The "code" environment covers all granularity levels, which allows researchers to chose the desired resolution.

The Websocket

While the code compilation workflow saves code snapshot data, an additional tool is used to capture the keystroke data. The coding environment uses a *websocket* to capture the log data. Every activity contains traceable information, the relevant data is displayed in Table 6.3.

Table 6.3: Websocket data.

Type	Example
Time stamp	2017-03-01T12:00:00.001
Browser Session	4e64637233f0899a8aa97
User Session	1d9f4934030dfc524e15d
User ID	001
Starting point	starting_point1.c
Action	connect / delta / run / close / focus / blur / disconnect

Tracking the browser session distinguishes the actions among multiple sessions. A single student could even code on two different devices and the data can still be separated into two sessions afterwards. Besides that, having the student's session information allows another user - for example a teaching assistant - to connect to

the student's session, run their code, make changes and write comments in order to assist the student.

The action events - listed in Table 6.3 - capture the type of interaction between the platform and the student. Besides connecting and disconnecting, a delta event captures all changes in the source code - including the location of the edit. The focus and blur events allow to track whether the student is actively on the platform or has it tabbed in the background.

In most studies in the field, the active time a student worked on a coding assignment was calculated by using the difference of the time stamps as an estimator for the time between to compilation events. This is not suitable in this course setting, because students can work on these assignments whenever they wish to. Often students were inactive for short periods of times or even paused their work and resumed it on the next day. An analysis of the websocket data allows the removal of inactive time and is necessary for an accurate estimation of the student's coding activity. The procedure is introduced later in the analysis.

6.3.3 Automated Feedback

Receiving feedback on their own code, not only on the syntax, but especially on the semantics of the programming problem is an important factor in programming instruction. Providing enough assistance for students gets more difficult even with the help of TAs, when the number of course participants grows. This results in the need of systems that can provide automatic feedback to students on syntactically correct, but semantically incorrect programs.

Feedback like this can be generated by a so called auto-grader. The complexity of an auto-grader varies greatly, from a simple *Input/Output* test to a decomposition of the underlying structure of the source code. Complexity further depends on the level of feedback given. An example that shows the possibilities of using a data

driven approach is the self-improving python programming tutor presented in [18]. The system transforms a student's semantically incorrect source code into an abstract structure and then finds a path from the student's source code to a solution while generating personalized hints for the student.

Prior to the current re-design, our coding environment already had some assignments with automated feedback integrated. A programming assignment with automated feedback consisted of a starting point, either empty or pre-populated, and a file that contained test cases and the corresponding output. This is a classical input/output approach which is easy to implement but comes with major downsides. It is not unusual for programming students to print out additional variables alongside the answer and even an unintended blank space that will break this testing approach. Instead, in our current system a new approach was implemented by directly testing the student's results *within* their source code. This requires a function call that is added right before the return statement of the main function. The function is called *check function* and is part of the pre-populated starting point. A feature of the coding environment allows parts of the source code to be hidden at the beginning and the end. This technique is described at the end of this chapter. The check function can require student variables as inputs, which can be already included in case the variable is declared as part of the starting point or the source code contains information that prompts the student to pass their declared variable to the function. In contrast, when students are prompted to write a function with a pre-defined name, the check function usually does not require arguments, because it can call the student's function directly. The check function then runs the test cases and determines if the function operates correctly or incorrectly. This system was implemented without changing the original platform's workflow of correctness checking. This implies that a predefined output is still needed for a positive outcome. Hence, the check function provides a generic output for correct scenar-

ios. A more interactive approach would demand a costly redesign of the workflow. However, this output is not visible for the student and the server does not evaluates the student's output.

Using an integrated check function has several benefits but also comes with a set of limitations. On the one hand, correctness checking through the check function solves major flaws in input/output testing. When students do not have to write functions - mainly because they have not yet been introduced to that concept - employing user input for further calculations allows this system to check for correctness and prevents academic dishonesty at the same time, because the values that are piped in as user input are unknown to the student. As a simplified example, a student can be prompted to retrieve a number from user input, calculate the square root of it and pass the result to the check function. The student can choose the user input when running his code and assuming that the number `4` was chosen as user input and the student wrote a print statement to display the result, then `2` would be displayed. Invisible for the student, the server will run the code a second time but using own predefined input. Assuming that the instructor predefined `1762`, then the check function requires the passed argument to be `42` in order to be correct. Students can only game the system if they know the predefined input or if they are lucky by trying out all possibilities.

On the other hand, it is not possible to check a program for correctness, when correctness checking involves testing whether students are correctly implementing certain print statements. A work-around for that problem would involve using both mechanisms side by side or an adjustment of the assignments to fit a check function style. Another negative impact of the check function approach is the presence of the function call itself. It can confuse programming novices at an early stage. To reduce this effect, students were briefly introduced to the check function.

The quizzes in this course, - as part of the LMS - were excellently suited for the

implementation of the auto-grader, because most quizzes had short, simple coding exercises integrated within via an IFrame. Student participation in these quizzes was guaranteed, as students had to solve quizzes in order to receive points that accounted for a percentage of their final grade. When students run their code and the auto-grader is successful, it returns a keyword that has to be submitted to finish the quiz. This proved useful for data collection, because students are obligated to solve the programming assignment in the coding environment in order to get the keyword.

Direct feedback provided by the auto-grader is not only beneficial to the student, but also helpful on the teaching side. The auto-grader assignments save time for the instructor and TAs, translating into more time for individual assistance for students who are in need of support.

6.3.4 Friendly Error Messages

Part of the work of this thesis was to create a workflow to implement enhanced compiler error messages into the current structure of the coding environment. This system was implemented before the start of the course and employed data from past uses of "code" in order to determine the most frequent error messages students received. This analysis is necessary because it is challenging to find a suitable explanation for any specific error without causing more confusion. The root cause of confusion is that C-compiler errors usually have multiple causes, demanding an explanation that is general enough to cover these cases, but still helpful for the student's context. It is, therefore, advantageous to spend the available development time only on the most frequent error messages that occur within this programming course. Based on a data set from 84 students and a total of 11,747 submissions and 4,440 errors, the most common errors in this context were identified. The top ten errors are listed in Table 6.4.

Table 6.4: Most common C errors from the historic data set.

	Error	Occurrence
1	error: 'X' undeclared (first use in this function)	649
2	error: conflicting types for 'X'	309
3	error: expected expression before 'X' token	261
4	error: ld returned 1 exit status	224
5	error: expected identifier or '(' before 'X'	224
6	error: unknown type name 'X'	194
7	error: expected expression before 'X'	189
8	error: expected ';' before 'X'	180
9	error: expected '=', ',', ';', 'asm' or '__attribute__' before 'X' token	177
10	error: incompatible type for argument i of 'X'	146

The top 20 errors in fact account for 76% of total errors. Friendly error messages were generated for the top 50 errors, covering 95% of the occurring errors. By comparing this historic data to the current data set, we were able to show that 15 out of 20 errors from the historic data set were also in the top 20 of the current data set. The top 20 errors from the current data set covered 77% of the total errors and the top 50 errors covered 95%. This shows that the data-driven approach to extract the most frequent errors proved effective, while only a fraction of the data was available when compared to the main data set. The top ten errors are displayed in Table 6.5 and were extracted from 68,248 submissions by 95 students.

Table 6.5: Most common C errors from the current data set.

	Error	Occurrence
1	error: 'X' undeclared (first use in this function)	2903
2	error: expected expression before 'X' token	1189
3	error: expected ';' before 'X'	1108
4	error: expected identifier or '(' before 'X'	860
5	error: expected ';' before 'X' token	768
6	error: conflicting types for 'X'	761
7	error: expected '=', ',', ';', 'asm' or '__attribute__' before 'X'	644
8	error: parameter name omitted	643
9	error: ld returned 1 exit status	618
10	error: expected '=', ',', ';', 'asm' or '__attribute__' before 'X' token	571

Knowing the most common errors allows the instructor to directly address these in class and prior to more complicated assignments. The instructor can offer targeted debugging training and gently introduce students to the compiler error messages.

It is important that students familiarize themselves with compiler error messages and learn to utilize these, which is why the friendly error message is displayed along with the original compiler error. It is worth noting that the most frequent errors can strongly vary depending on the course setting, the assignments, and also the students. Yet gathering the compiler error messages in an online coding environment, even without a link to a specific student code, can be set up without much effort. This is a first step in making use of course-specific data and improving the teaching of an ongoing course as well as the teaching of future courses.

Integration

To generate a friendly error message for a student, an additional script was integrated into the workflow, which looks as follows:

1. The student compilation attempt results in an error.
2. A script is called that takes the error file as an argument.
3. The error message is standardized by replacing variable or function names with a generic 'X', instead of just removing it. This preserves the original structure of the error message.
4. Based on the standardized error, a friendly error will be taken from a dictionary and written to an output file.
5. The server reads that output file and displays the content right after the compiler error message.

This workflow was chosen, because it could easily be integrated into the current system environment and also allows a dynamic feature handling where the course instructor can choose to use the feature or not. Fig. 6.7 shows the resulting view when encountering an error. The friendly error message is displayed in yellow.

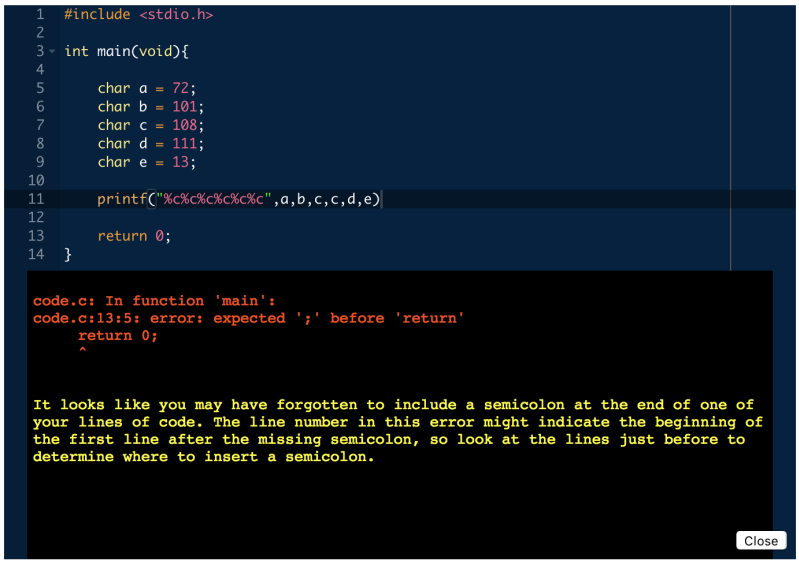


Figure 6.7: A friendly error message.

It is important to repeat the process of gathering course feedback and analyzing subsequent data to adapt the friendly error messages at the end of the course. This is an evolutionary process that occurs over time and is dependent on continuous improvement. Within this process, some standardized errors will have to be disaggregated and be further differentiated, and in other cases similar errors that occur frequently and result from the same mistake will be aggregated into one standardized error.

With ongoing discussions occurring within the literature about the effectiveness of enhancing compiler error messages, it is important to note that the correct implementation is clearly the key for success.

6.3.5 Tools and Features

The coding environment already comes with a handful of features that not only benefit students, but also assist instructors. While the auto-grader system and the friendly error messages have already been discussed, other features are briefly outlined in this section.

Masked Code

The coding environment has the option to mask portions of the code from students (mostly for the purpose of auto-grading) by using special strings within the source file of the starting points. These special strings mark the beginning and the end of the part that is visible to students. Everything that is above the first string or below the second string is not visible and not editable for students at all. When the *Run* button is executed, within a series of events, the full source code (including hidden patterns) is combined, the special strings are removed and the source code is then sent to the compiler. This allows to hide parts of the pre-populated source code that can be confusing for novice students at an early stage. Most importantly, this feature is essential to implement the already presented auto-grader system: the important check function is part of the source code and would lead to more confusion if shown to students. Besides, the check function sometimes contains the solution and should, thus, clearly be invisible to students. The pre-populated source code including the masked code is displayed in Fig. 6.8.

```
1  void check_function(int);
2  ----------student code starts here----------
3  #include <stdio.h>
4
5  int main(void)
6  {
7      int age;
8      age = 0;
9
10     printf("Please enter your age: ");
11     /* Your code here. */
12
13     /* And here some code to test whether everything worked correctly: */
14     printf("You entered that the age is %d.", age);
15
16     /* Do not change anything below here. */
17     check_function(age);
18     return(0);
19 }
20 ----------student code ends here----------
21 void check_function(int age)
22 {
23     int good = 0;
24
25     if (age == 03755) good = 1;
26     printf("\n----------check outputs starts here----------");
27     if (good) printf("Excellent");
28     printf("\n----------check outputs ends here----------\n");
29 }
```

User input | Start Fresh | Run

Figure 6.8: Implementation of the check function.

The Admin Dashboard

The instructor and the TAs can access an admin dashboard for a given starting point that displays a tile for each student who has worked on that assignment. A sample view of the dashboard is given in Fig. 6.9.

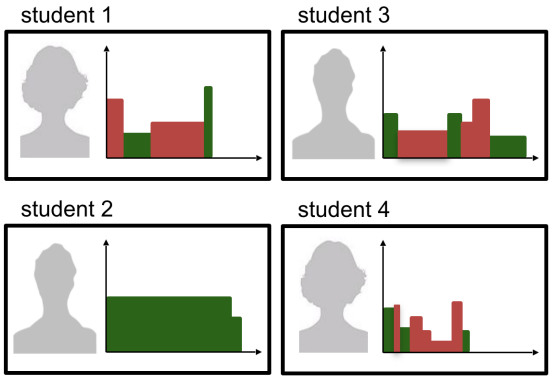


Figure 6.9: The admin dashboard.

Each student's tile contains a graph where the x-axis represents the time and the y-axis represents the number of student actions. Each block represents a run event and is colorized in red when the student encountered an error, otherwise in green. If the assignment has an auto-grader integrated, the whole box is additionally colorized for students who solved this assignment correctly.

With this dashboard, the instructor is able to oversee student's coding activities which is especially beneficial for in-class coding assignments. It is possible to click on a particular student and open up a collaborative code session within the student's code. The instructor can run the compiler, make edits and add comments to the source code. This way, students can get immediate remote assistance. In addition, TAs can use the dashboard to inspect and run students' source code while grading assignments. At the time of writing, using the dashboard for grading purposes is only supplementary, because assignments are not submitted through the coding environment and student code is not frozen at a deadline. Future implementation of a reliable workflow, for both the students and the TAs, could ease

the submission effort for students, reduce grading time for TAs and improve the feedback given to students.

Code Review

Given an assignment and a student, the instructor can review every student action in a chronological order. This allows to inspect how a student coded and can help determine whether and if so with what a student is struggling. At the time of writing, this feature lacks guidance for the instructor because it is difficult to decide after one week of coding which student's actions need to be inspected more closely and in particularly at what assignment to look at.

This thesis presents methods to flag at-risk students that can be implemented as a decision support tool for the instructor in order to use the inspection tools on flagged students. Afterwards, the instructor can decide whether the flagged student need an intervention and if so develop an intervention strategy that addresses students' deficits.

6.3.6 Student Survey

At the end of the course, students were asked in an anonymous survey to reflect on their experiences with the coding environment. This survey was completed by 78% of all students. Only 37% of these students stated that they coded in the coding environment for the majority of the time. Given the survey results, this is partly because of students missing features within the coding environment which would be necessary to work on more complex and interactive problems such as the longer assignments.

Overall, the biggest benefit the majority of students mentioned was the ease of starting to code and the ability to test ideas quickly by compiling their code in one click and getting immediate feedback. Students were also asked about their

impression with the friendly error messages to receive an idea of whether these messages affected the students in any way. Almost 70% of the students answered that the friendly error messages were helpful.

On the other hand, students also reported some major issues they had with the coding environment, which could have led students to avoid using the environment. Although the majority of students reported that they liked the instant feedback, 28% of the students reported to be confused by the check function call within their source code. Since students had many assignments that involved user input which had to be pre-determined before running the compiler, many students wished for a more interactive coding environment that is not reliant on pre-determined user input. Similarly, programming assignments that involved file-handling were dependent on files that were stored - by the instructor - on the server. Students could neither inspect those files nor use their own files. Lastly, students were frustrated with the sizing of the embedded coding window, which is restricted by the Canvas LMS. We are in the process of building a full-screen option to resolve this issue and we are looking into ways to implement an interactive mode as well as a different way to handle files because we believe that programming assignments that require these features are important to cover.

Chapter 7

Analysis

The goal of this chapter is to extract useful information from the available data set and to build predictive models that can be used to guide the instructor in deciding which students may be at-risk and in need of an intervention.

First, the data set is preprocessed, which includes data cleaning and data preparation. Then the metrics - discussed in Chapter 5 - are evaluated separately and are compared to each other. This is completed via the analysis and evaluation of time-dependent attributes and other relevant findings. Finally, these findings are combined to build predictive models and discussed for potential applications.

7.1 Data Preprocessing

Before starting with the analysis, the data has to be cleaned. Only data generated within the selected time frame and from course related students are of interest. Non course related persons - TAs, instructors and external users of the platform - were removed from the data set. Besides that, incomplete data was removed, which includes incomplete submission packages that resulted from two major server outage events. Following the data cleaning, the students' LMS data was joined with the students' coding data. The LMS data includes auditors and test students that also

needed to be removed. Out of the 95 students that started, two dropped the course for non course related reasons. Out of the remaining 93 students, 88 students finished the course and were graded, whereas five students withdrew from the course to avoid a poor or failing grade. These five students, as well as students that finished the course with a grade lower than 75%, are classified as at-risk students. The students that withdrew from the course are included in the analysis of the data up until the first midterm exam.

Recall (Section 6.1) that some of the assignments are solved during class, where students are programming in pairs, all in-class assignments with less than $2/3$ participation were excluded from the data set. This same cut-off was also used for the ungraded try-out assignments which led to a total removal of 46 assignments. The information of the available data set after the first preprocessing step is listed in Table 7.1.

Table 7.1: Content of the main data set before and after preprocessing - students that finished the course are displayed in parenthesis.

	Before	After
Students	95	93 (88)
Assignments	86	40
Submissions	68,248	59,155
Errors	24,996	21,599

Instead of using sessions that are restricted by a time limit, submission packages consist of all the work from one student on one assignment. While Jadud [19] was the only one using a cut-off value of at least seven submission for a session to be included in the analysis, he did not state how he determined the value. The only other mentioning of a minimum number of submissions per session is in [5], but they only take up on Jadud's suggestion and wrap this in a definition that requires seven distinct submissions with no longer than 20 minutes separation between two submissions. Instead, this thesis argues that this definition of a session is not

universal, due to the context. In most studies the data was generated in lab sessions where a single student only had a single number of sessions. On the contrary, in our data set a student worked, on average, on 33 assignments, resulting in an equal number of submission packages which we analyzed instead of sessions.

The main objective when trying to predict struggling students is to identify the students as early as possible in order to try some interventions. This requires sufficient data. The total available number of submissions at the end of every week as well as the submissions generated in the corresponding week are displayed in Fig. 7.1.

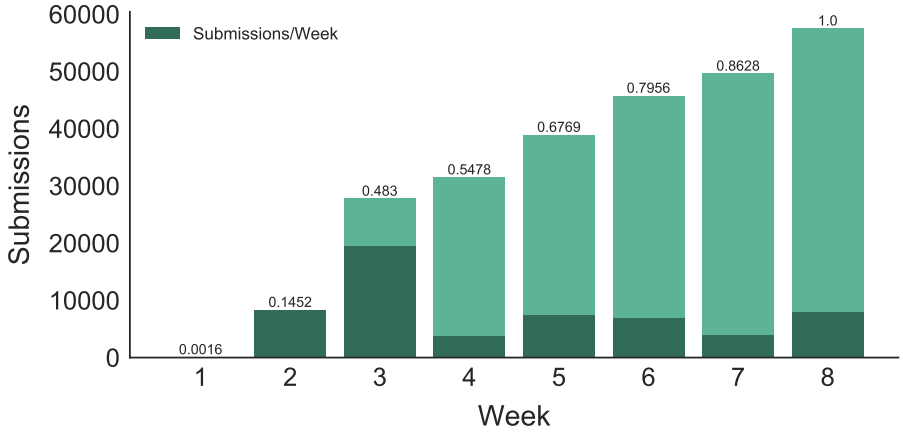


Figure 7.1: Student submissions over time.

When looking at Week 1, it is evident that there is not enough data available to make any predictions. Not only do the 474 submissions account for less than 1% of the total submissions over the duration of the course, but only 29 students have started to code up to this time. The reason for this is that the first week of the course does not include any coding. The available data are from students who already started with coding assignments ahead of time.

In contrast, by the end of Week 3, already 48.53% of the total data was available. This was the most coding-intensive week - on the coding platform - for the students, right before the midterm exam. Unfortunately, using this data to predict

the outcome of the midterm exam would be of little help, since there is little time left for an intervention. Lastly, it is important to separate the data based on the context of the assignment to allow for a differentiated analysis. A breakdown of the assignment types is given in Table 7.2, where n_a is the number of assignments, n_{sub} is the number of submissions and n_{err} is the number of errors.

Table 7.2: Available student data by the end of Week 3 and Week 8.

	after Week 3			after Week 8		
Context	n_a	n_{sub}	n_{err}	n_a	n_{sub}	n_{err}
Homework	11	17,185	4,048	15	29,778	9,050
Quizzes (AG)	7	7,237	2,845	13	11,729	4,998
Gateway	1	2,527	1,684	3	10,662	4,128
In-class	1	1,369	517	5	5,793	2,845
Quizzes (TO)	3	781	295	4	1,193	578
Total	32	29,099	9,389	40	59,155	21,600

7.2 Quantifying the Error Compilation Behavior

Before evaluating the predictive performance of the metrics described in Chapter 5, their underlying distribution as well as their value ranges are described. After an individual inspection, they are compared to each other and discussed in context.

7.2.1 Describing the Distributions

After implementing the algorithms and running the analysis on the preprocessed data after week eight of the course, the distribution of the Error Quotient (EQ), Robust Relative (RR) and Repeated Error Density (RED) is displayed in Fig. 7.2. Because the EQ is normalized, its values are limited between 0 and 1. In this data set, the values range from 0.05 to 0.39 with a mean value of 0.19. When testing the EQ distribution for normality using the Shapiro-Wilks test [44], the null hypothesis

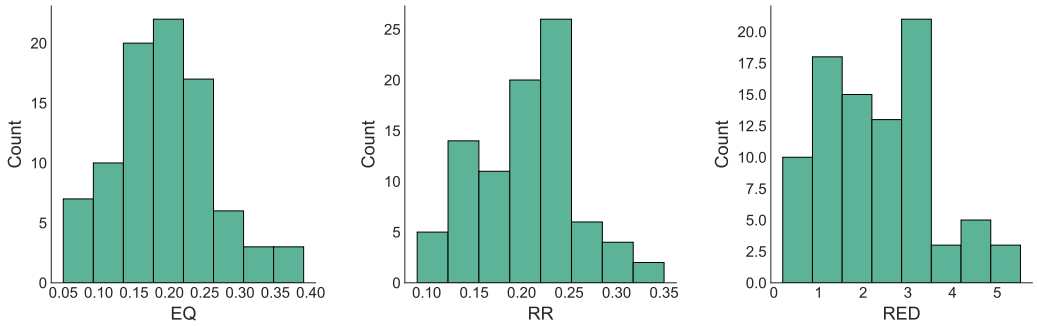


Figure 7.2: Distribution of the EQ, RR, RED.

that the data is normally distributed could not be rejected ($p = 0.241$). As Jadud reported, when testing the distribution of the EQ against the blackbox data set the kurtosis and skewness slightly differ from those of a normal distribution [21]. In contrary, testing whether the kurtosis and the skewness in this data set would differ significantly from those of a normal distribution, the null hypothesis could not be rejected ($p = 0.567$ using [45] and $p = 0.112$ using [46]). Based on this result the EQ is approximately normally distributed.

The RR values are also normalized and in this data set their values range from 0.09 to 0.35 with a mean value of 0.205. Applying the same test for normality of the distribution, of the kurtosis and of the skewness, the resulting p-values ($p = 0.515$, $p = 0.642$, $p = 0.452$) indicate that the RR is also approximately normally distributed.

Lastly the RED's values upper limit is not capped, but dependent on the length of a sequence. In this data set the values range from 0.2 to 5.52 with a mean value of 2.29. The resulting p-values for the normality tests ($p = 0.013$, $p = 0.375$, $p = 0.084$) indicate that the RED is not normally distributed, which might cause potential issues and can be a reason for the different performance that was visible within the analysis.

7.2.2 Individual Predictive Performance

To test the general ability of the EQ to serve as a predictor for the student's performance, the available EQ data up to the dates of the exams - the two midterms and the final - has been plotted with the corresponding results. Fig. 7.3 shows the results and the resulting regression curve after applying a linear model.

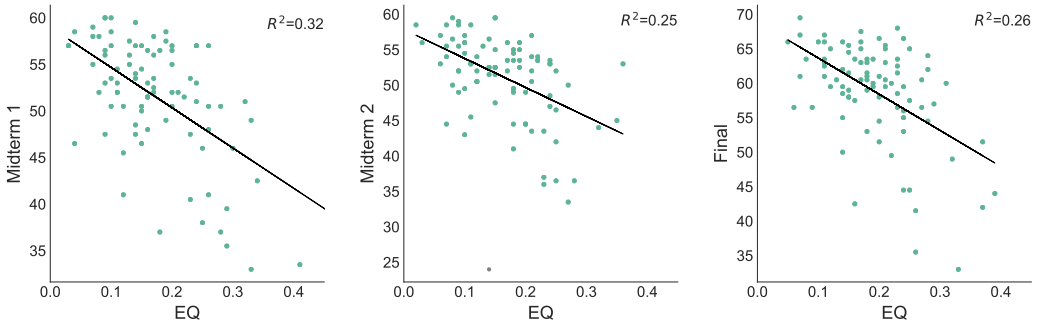


Figure 7.3: Correlation of the EQ and exam performance.

There is a strong correlation between the EQ and exam performance - in all three cases $p < 0.0001$. For the second model an outlier that differs extremely from the rest of the students has been excluded - marked in grey.

Similarly to the EQ, Fig. 7.4 displays the predictive power of the RR for a student's exam performance.

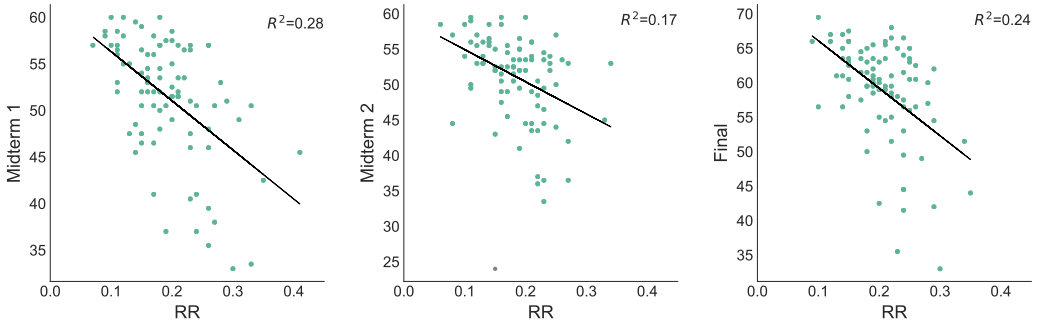


Figure 7.4: Correlation of the RR and exam performance.

The resulting model indicates a correlation between the RR and the student's exam outcome, too ($p < 0.001$). However, compared to the predictive power of the EQ, the model consisting only of the RR results in a significant lower R^2.

Lastly Fig. 7.5 shows the predictive power of the RED for a student's exam outcome.

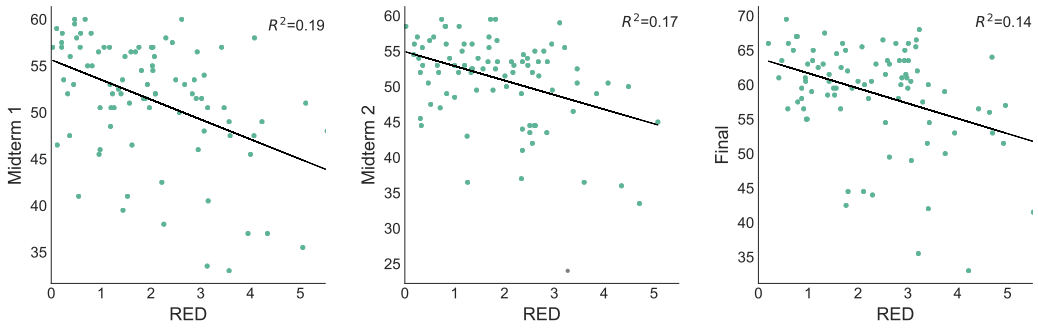


Figure 7.5: Correlation of the RED and exam performance.

Although the data in these figures is much more scattered than for the EQ and the RR, there is still a clear relationship visible. Interestingly, the amount of variance explained by these models - quantified by R^2 - continuously drops. The reason for that is exposed when the RED is evaluated in context of the next section.

The literature shows that when predicting students' performance, it is not always clear what 'performance' actually means. While some studies used the midterms or the final to define performance, others suggested using overall course performance. One reason to use exam performance would be the direct effect it has on the student. In fact, four out of the five students who dropped the course left after they received the evaluation of the first midterm exam. Then again, students usually care mostly about the overall course grade, which ideally should be a reflection of the student's learning success. For building prediction models, it is useful to test the metrics ability to predict both, exam performance and course performance. Fig. 7.6 shows the predictive performance of the three metrics for a student's overall course grade.

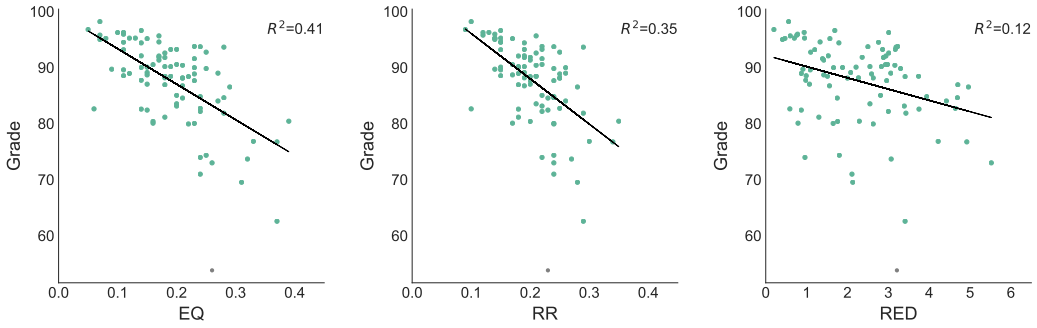


Figure 7.6: Correlation of the metrics and overall course performance.

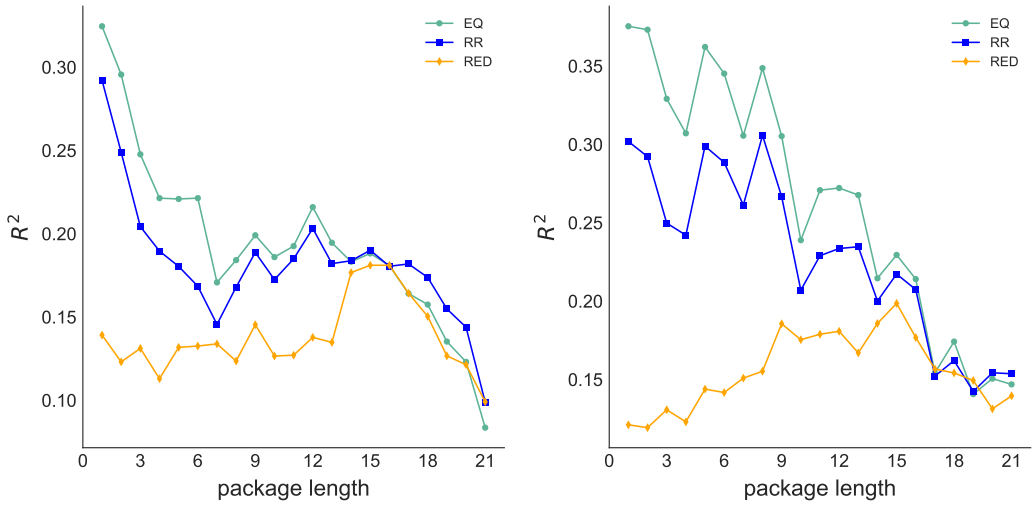
The same outlier that occurred in Fig. 7.3 was removed in this model for the equivalent reason. For both the EQ and the RR the model explains significantly more variance than the models using the exam scores as the response variable. The correlation of the EQ with the overall course performance is high ($p < 0.0001$). The model explains 41% of the total variance ($R^2 = 0.41$) performing significantly better than in any of the previous studies. The model using the RR explains 35% of the variance ($R^2 = 0.35$). Only for the RED another reduction in explained variance (from $R^2 = 0.14$ to $R^2 = 0.12$) is recorded.

Based on these results, a general suitability of the EQ and the RR as a student performance metric can be assumed.

7.2.3 Determination of a Minimum Number of Submissions

Because Jadud reported that a minimum of seven submissions per session were needed and this was blindly adopted by [5], the influence of the package length needed to be analyzed as part of our work. A parameter `min_package_length` was used to exclude submission packages, that had fewer submissions than this limit. Recall that for us, a *submission* consists of code snapshot data and websocket log data. Simple linear regression models were built on the resulting data set and the explained variance was captured for each metric. Considering that 80% of all

submission packages had less than 21 submissions, the upper limit was set to 21. Fig. 7.7 shows the results of this analysis:



(a) Y = first midterm score. (b) Y = course grade.

Figure 7.7: Influence of the minimum package length on the model performance.

Increasing the package-length parameter causes the analysis to exclude more and more data. This leads to a reduction of variance explained for the EQ and the RR. Only for the RED an increase of explained variance, compared to the data with a minimum package length of one, is recorded around a minimum of 15 submissions per package. The performance improvement for the RED when only including packages that have at least 15 submissions is reviewed in the discussion.

The rapid decrease of R^2 that is visible in Fig. 7.7a for package lengths of greater then 15 is evoked by the huge loss of data that occurs when omitting this many submissions.

7.2.4 Analyzing the Stability

Before discussing the metrics in context another important analysis has to be done, namely inspecting the number of packages for a single student. Since in other studies sessions were mostly generated during lab hours, the number of sessions in

these studies is typically restricted by the number of assigned laboratory exercises. Using our submission package approach, the maximum number of packages per student is limited by the total number of assignments. For our data set the maximum number of packages per student was therefore 40. Fig. 7.8 shows the high variability of all three metrics, when only a few packages are available.

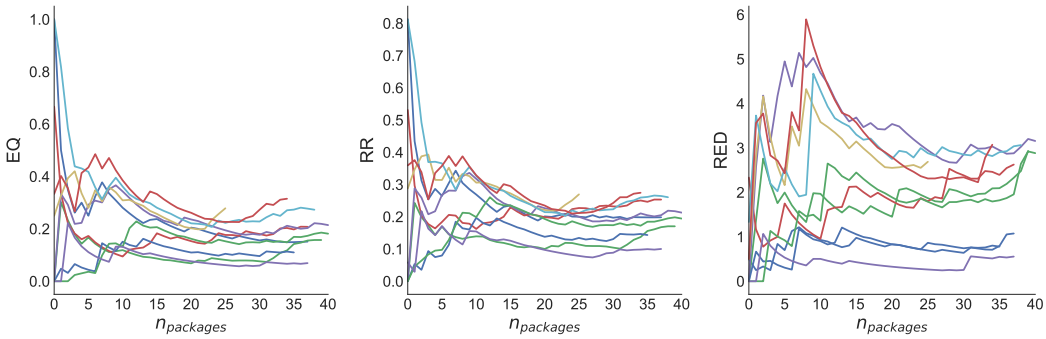


Figure 7.8: Influence of available packages on the metrics' accuracy.

At around 20 packages the metrics appear to stabilize. Because the EQ and the RR are highly correlated, the respective figures are very similar. For the RED there is a higher variance between the students for when the process stabilizes, which can potentially be explained by the non-normal distribution. At around 35 packages, the RED values seem to follow an upward trend. This could be explained by Becker's assumption in [26], that the RED will inevitably rise with more submissions.

Knowing when a metric finds its steady-state is important, because this could be a determining factor for the performance of early predictive models. By the end of week three, only 29% of the students had 21 or more packages and by the end of week four, 77% had crossed this threshold. For course design, it is important to note that the earlier and the more frequent students start programming, the faster these metrics stabilize and become more accurate.

7.2.5 Discussion

The above results show that each of the proposed metrics is correlating with the student's overall course performance with different levels of overall variance that can be explained by the models. The next step is to see whether these metrics are dependent on the context of the programming assignment and how the metrics perform compared to each other. The first part can be addressed by looking at the individual performance for each context which is displayed in Table 7.3, where $\bar{a}_{stud}$ is the average number of assignments per student and $\bar{a}_{len}$ the average number of available submissions per assignment.

Table 7.3: Student performance metrics in context.

Context	n_a	n_{sub}	n_{err}	$\bar{a}_{stud}$	$\bar{a}_{len}$	R^2_{EQ}	R^2_{RR}	R^2_{RED}	R^2_{errsub}
All	45	57110	20737	34.69	18.92	0.41	0.35	0.12	0.32
Homework	15	28738	8723	12.45	26.54	0.29	0.24	0.07	0.2
Quizzes	17	12311	5308	15.46	9.15	0.26	0.2	0.07	0.11
Quizzes (AG)	13	11206	4774	12.54	10.27	0.22	0.16	0.06	0.1
Gateway	3	10346	3904	2.86	41.55	0.2	0.18	0.14	0.09
In Class	5	5715	2802	3.97	16.76	0.04	0.05	0.0	0.06
Quizzes (TO)	4	1105	533	2.95	4.35	0.06	0.06	0.02	0.03

Besides the error compilation metrics, the error ratio - defined as errors per submission - was also evaluated and compared to the more complex metrics.

A first observation is that the more submissions and errors are available the better the EQ, RR and the error ratio perform. This might lead one to conclude that only the number of submissions and errors made are relevant for the performance of the metric. Interestingly, the analogous trend does not seem to hold true for the RED. While the RED has a weak performance using only the data from the gateway assignments, the performance worsens when only the data from the auto-grader quizzes is available, although both data sets have a similar number of submissions and errors. The main difference is that students coded four times as much per gateway assignment compared to a single quiz. This suggests that the RED relies

not only on the number of submissions, but also on the length of student packages.

Predictive Performance Over Time

The above results indicate a general suitability of the EQ, RR, and RED as predictors of student performance. The above models were built with all the available data to the date of the exams, to the end of the course respectively. In order to be used as a predictor that can flag struggling students at an early stage, predictive performance of these metrics needs to be evaluated at different points of the course. Fig. 7.9 displays the predictive power at the end of each week for a student's overall course performance comparing all metrics to each other.

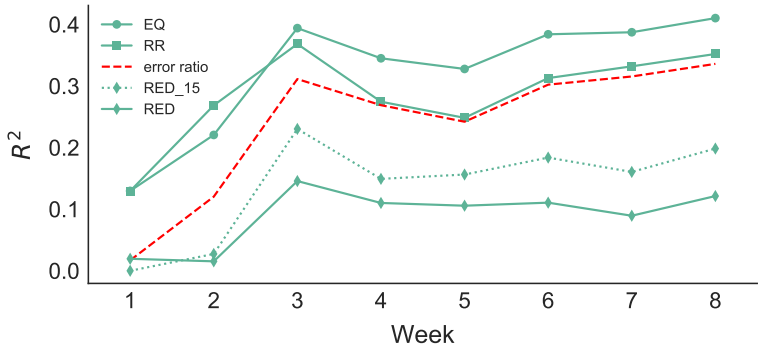


Figure 7.9: The student performance metrics over time.

The dashed line indicates the predictive performance of a model built with the error ratio. Interestingly, over time the RR does only just as well as simply taking the error ratio, despite having a reasonably better performance in week two. [24] reported a weaker performance of the EQ in comparison to the RR. Within our course setting, the RR has a weaker performance in every consecutive week, despite its great performance in week two. The RED is performing worse than the error ratio in every week, although the reduced model - where the package length is equal to or greater than 15 submissions (RED_15 in Fig. 7.9) - shows a slight improvement. The figure also displays that the predictive power of the metrics increases over time

which is directly related to the amount of available data at the end of each week. When looking back at Fig. 7.1, the spike from Week 2 to Week 3 is explained by an almost equal spike in submissions.

Since the main objective is to predict a student's performance as early as possible, preferably before the first midterm exam, it makes sense to pick the EQ over the other metrics in order to build a model that can be used for prediction. Because there is not enough data available in Week 1 and the first midterm already takes place after the end of Week 3 in the current course setting, the only possibility to do the prediction is currently within Week 3.

It is important to emphasize that for all of the metrics the amount of available data is of critical importance. All these metrics are built upon the student error compilation behavior and, except for the RED, they seem to not be influenced by the context of the assignments. The RED performed best on the data from the gateway assignments. This data was generated by students solving mandatory assignments in a separate lab session. Either this context, or the fact that average package length is about 41 submissions, is the reason for the performance increase. Based on the design of the algorithm of the RED, the latter seems more reasonable and this hypothesis is supported by the RED's slight improvement of performance, when the minimum package length is increased to 15. This should be considered for future course design because front-loading of coding assignments can drastically improve the metrics' accuracy.

Altogether, it must be noted that these metrics were developed by their respective authors and mostly tested in different course settings, containing fewer assignments, but longer sessions. The above results show that the error compilation metrics are - despite their variability - language agnostic. This knowledge should be utilized to further improve these metrics or to come up with alternatives that quantify the error compilation behavior.

7.3 Analyzing Time Dependent Data

Since learning to program requires much practice, our hypothesis is that students who invest a large amount of time into coding will perform better in the course. Hence, we want to utilize these time dependent variables to flag students who might not be spending enough time learning and are possible candidates of being at-risk of failing the course.

7.3.1 Time Spent Actively Coding

In order to quantify the amount of time a student spent coding, there are several approaches. If the student snapshot data and the corresponding timestamps are available, the simplest procedure is to take the time difference between two successive timestamps as the time a student spent coding. While this was done in other studies and was perfectly fine in their respective environments, using that approach within our course setting would yield non-representative measures of student coding activity. The reason is that students are not only coding in class, but also as part of the required quizzes and during assignments on their own schedules. In contrast to a lab setting, it is not uncommon for students to interrupt their work, take a break and finish the assignment at a later time. Students probably also spend time on social media because there is no immediate time limit to solve the assignments. Fig. 7.10 shows the student activity during a day and during a week over the duration of the whole course. As the figure shows, students code throughout the entire day, with coding activity peaking around noon and late at night. Visualizing the data from other studies in a similar way would only show activities during the lab hours. The fact that students can code whenever they want to requires a new approach to calculating the students' real active time. By using the data from the websocket log, the time between two submission timestamps can be analyzed and

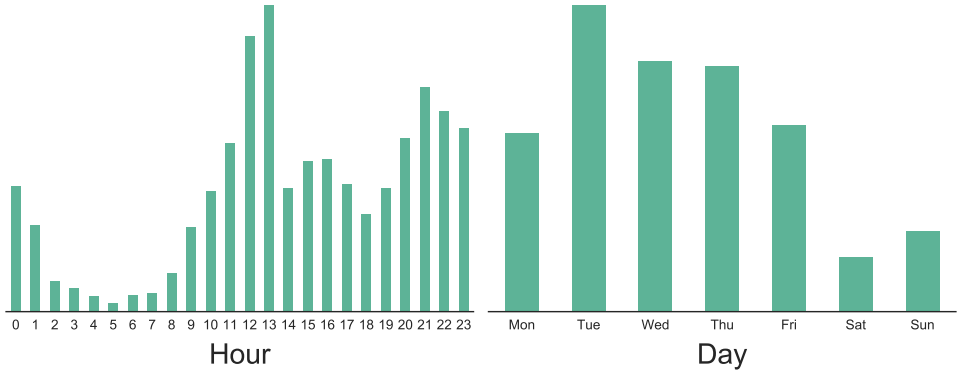


Figure 7.10: Students coding activity.

decomposed. Between two code submissions, the student is usually modifying the code. These actions are captured in the websocket log as explained in Section 6.3.2. Two consecutive actions are called a transition and their time difference is called a transition time. Each transition time is then categorized as: active time t_a (the student was actively working on the assignment); browsing time t_b (the student tabbed out of the assignment but came back within a specified time limit); idle time t_i (the student was still connected to the server but the next action was outside of the allowed time); and offline time t_o (the time between two timestamps where the student was not connected to the coding environment). The average proportions of these four different components for all students are displayed in Fig. 7.11.

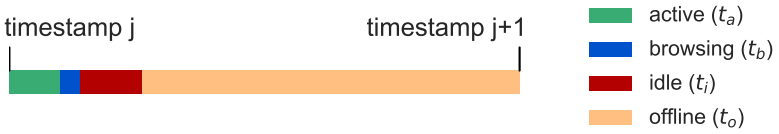


Figure 7.11: Student activity between two code submissions.

The time limit that distinguishes between active time and idle time was determined by maximizing the explanatory power of the portion of active time that showed significant correlation to overall course performance. The explanatory power incrementally increased up until a limit of 400 seconds. A limit greater than that neither increased nor decreased the explanatory power. This implies that a student who is inactive on the coding platform for less than 400 seconds, may still take outside actions that are related to the success in solving the programming assignment - such looking up syntax or researching methods to solve an error.

Plotting the overall active time against course performance as seen in Fig. 7.12 still does not reveal any kind of a trend.

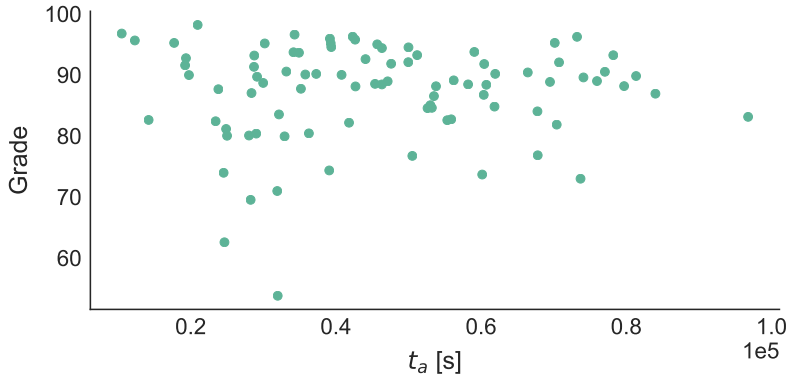


Figure 7.12: Scatterplot of active time against course performance.

There are two students with a low active time and a weak performance, but those are outliers, which will become clear in future plots.

Because all assignments are included in this analysis, there may be noise in the data: The time a student spends on an assignment is related to the type of assignment. This leads to the next step of breaking the active time down by assignment type.

7.3.2 Decomposition of Time Spent Coding

The necessity of breaking down the time spent coding also rises from the fact that the data is collected in an on-campus course setting instead of a laboratory environment. When t_a was generated for subsets of the data that corresponded to the different assignment types, the data was still as scattered as before for homework, gateway and in-class assignments, except for quiz assignments.

Both the quizzes with automated feedback and the try-out quizzes, as well as the quizzes in total indicated a negative relationship between t_a and course performance. Out of these, the average time $\bar{t_a}$ a student spent on an auto-graded quiz assignment showed the highest correlation to the course performance. The resulting linear model after removing two outliers is shown in Fig. 7.13.

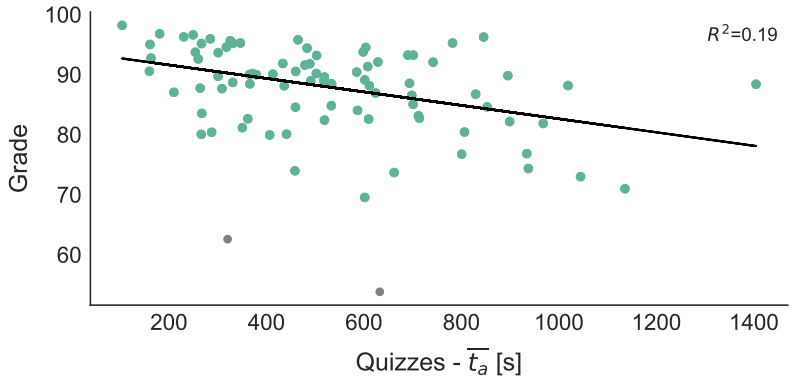


Figure 7.13: Scatterplot of time spent on quizzes and course performance.

Other factors that could be causing noise

All quizzes, as well as most of the homework assignments had the coding environment embedded and the students were encouraged to use the coding environment. But a survey - that was answered by 69 students after the end of the course - revealed that over 60% of the students coded either in the terminal environment or somewhere else rather than in the coding environment. This conflicts with the goal of mapping students' coding efforts to their overall performance, as much of the

effort is not captured at all. Since the full integration of the coding environment was still at an experimental stage, students had to submit a file containing their source code for each homework assignment. This led to many students not coding their homework assignments in the embedded coding environment, in part due to convenience reasons that were revealed in the survey. This is in contrast to the quiz assignments, where students needed to at least submit their solution in the coding environment to obtain a keyword, which then credited them points. Another reason that distinguishes quiz assignments with automated feedback from the homework assignments is, that once students obtain the keyword from the quiz auto-grader they will stop working on the quiz, since they know that it is solved and they automatically get points credited. On homework assignments however, students would often review and try to improve their homework up until shortly before the deadline, making it difficult to distinguish these students from low performing students based on amount of time spent coding. (Low performing students who spent greater amounts of time trying to solve the homework.)

Since excluding students with little active time from the data set would lead to exclusion of students of interest, instead, 30 students that used the coding environment the least (as measured by working on the least number of assignments captured by the coding environment) were removed from the data set. This does not mean that these students did not do the assignments, they simply did not do so in the coding environment sufficiently often. The 30 excluded students spent on average 42% less time on the coding platform, had 47% fewer source code submissions, worked on 23% fewer assignments in total and 37% fewer homework assignments. The resulting model after excluding the 30 students explains 34% of the variance and is displayed in Fig. 7.14.

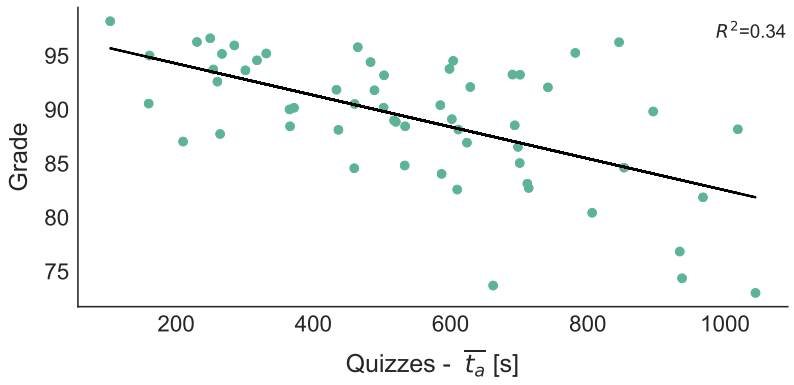


Figure 7.14: Scatterplot of time spent on quizzes and course performance after filtering.

It is remarkable that the overall course performance of a student can be explained - to some extent - by the average amount of time it takes them to solve these small programming problems, even though these only count for 5% of the student's final grade. This can partly be explained by the programming problems posed in quizzes themselves: They cover the basics of programming and essential coding principles. Based on these results, it can be concluded that students who struggle to understand these basic concepts are less likely to have a good final grade.

7.4 Mastering the Essentials

The auto-grader quizzes also revealed that students who did not solve the quizzes are less likely to be high-performing students.

The auto-grader quizzes can only be solved by entering a keyword that is returned to the student by the auto-grader when the programming assignment is solved correctly. At the same time this is captured by the server as a success event. Based on this information, the number of solved assignments can be extracted for every student and afterwards a student's *solved ratio* r can be calculated by dividing the sum of the solved assignments by the maximum of assignments solved by a

single student. For example, the student who solved the most assignments, solved 10 assignments. The student's solved ratio is therefore $10/10 = 1.0$. Another student solved 8 assignments, resulting in a solved ratio of $8/10 = 0.8$. This allows to rank a student relative to their peers. Fig. 7.15 shows the solved ratio and the corresponding students grade.

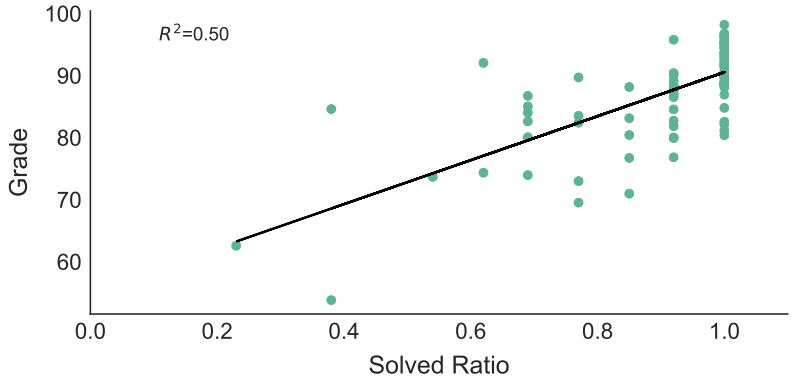


Figure 7.15: Correlation of the solved ratio r and the overall course performance.

There is a very strong correlation between the number of quiz assignments solved and the overall course performance. Using all available data, the model can explain almost up to 50% of the total variance, although the quizzes account only for five percent of the overall grade. As for the metrics that quantify the error compilation behavior, the predictive performance overtime using the solve ratio is compared to the EQ in Fig. 7.16.

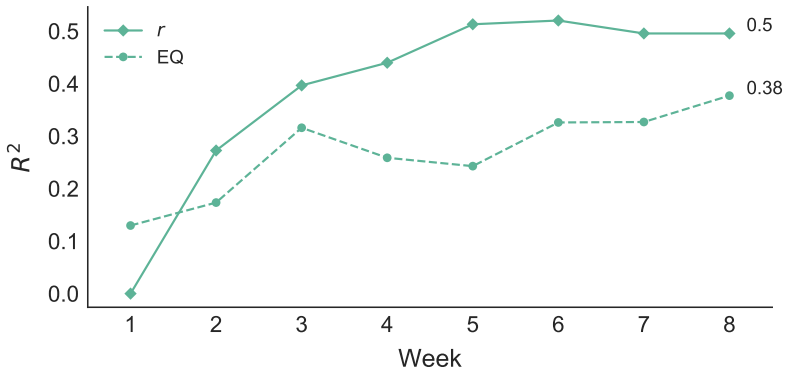


Figure 7.16: Predictive performance of r over time.

A simple linear regression model built on the solved ratio is visibly stronger than a model built on the EQ, with a relatively good performance already in Week 2.

This is again an indication that the quiz assignments play an important role in this course. Failing to master the basics at an early stage can have a negative impact during the rest of the term. Students who failed to solve the quizzes did so either out of laziness and a failure to recognize the importance of class preparation, or due to having serious difficulties trying to solve the quizzes and instead giving up. While the motivation might be unclear at that point, the data shows that failing the quizzes for either reason does have a strong impact on the student's overall performance, suggesting that this metric can be harnessed to build a predictive model.

7.5 Predicting At-risk Students

After evaluating the student performance metrics and other information extracted from the coding environment, the resulting knowledge can be used to build models capable of flagging students with a high probability of dropping the course or failing the class before their first midterm exam. These models need to be highly accurate at the earliest point in time possible. Because students only start to code in week two and the first midterm is at the end of week three, the days after the second week are of interest and need to be explored.

As stated earlier, there are two options to predict at-risk students. The first option is the actual prediction of the exam outcome or course outcome, while the second option is the classification of a student as either pass or fail. Both options are explored in this section.

7.5.1 Prediction

In the previous sections simple linear regression models were built for each metric. The next step is to build a multiple linear regression model including the time dependent variable and the solve ratio. Since the EQ, RR, and RED are all metrics based on the student's error compilation behavior, they do modestly correlate with each other. Therefore, it makes sense to select one metric at a time to use in the model.

Table 7.4 displays different models based on the full data set at the end of week eight with overall course performance being the response variable.

Table 7.4: Multiple linear regression models using the full data set.

Predictors	R^2	R^2_{adj}	AIC
EQ, r	0.62	0.61	532.84
RR, r	0.63	0.62	531.5
RED, r	0.54	0.53	550.95
EQ, $\overline{t_a}$	0.39	0.37	575.31
RR, $\overline{t_a}$	0.33	0.32	583.09
RED, $\overline{t_a}$	0.16	0.14	603.46
EQ, r, $\overline{t_a}$	0.63	0.61	533.99
RR, r, $\overline{t_a}$	0.63	0.62	532.07
RED, r, $\overline{t_a}$	0.54	0.53	551.33

Two observations can be made:

1. When the solved ratio is combined with the error compilation metrics, the RED is outperformed by the EQ and the RR, while the EQ and RR performed similarly.

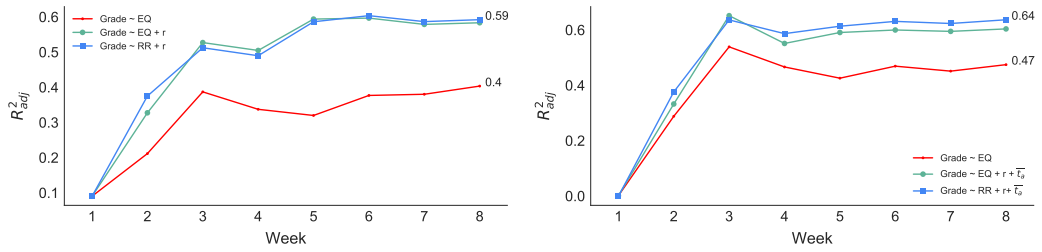
2. Using the full data set, $\bar{t_a}$ should be excluded because it does not improve the performance of the model: While R^2_{adj} is unchanged, the AIC increases when $\bar{t_a}$ is included.

When using the reduced data set, a multiple linear regression model built with the RR, r and $\bar{t_a}$ performs best as shown in Table 7.5.

Table 7.5: Multiple linear regression models using the reduced data set.

Predictors	R^2	R^2_{adj}	AIC
EQ, r	0.58	0.57	324.13
RR, r	0.61	0.6	320.4
EQ, $\overline{t_a}$	0.57	0.55	326.43
RR, $\overline{t_a}$	0.58	0.57	324.47
EQ, r, $\overline{t_a}$	0.62	0.6	320.25
RR, r, $\overline{t_a}$	0.66	0.64	315.21

Utilizing this information, two models can be built based on a reduced data set and a full data set. For the full data set, a multivariate model is built by combining the solve ratio r with either the EQ or the RR. For the reduced data set, the time spent on auto-grader assignments is added to the model. The multivariate models are compared to a univariate model of the EQ. Fig. 7.17 shows the predictive performance of the models - with the overall course performance being the response variable - for every week.



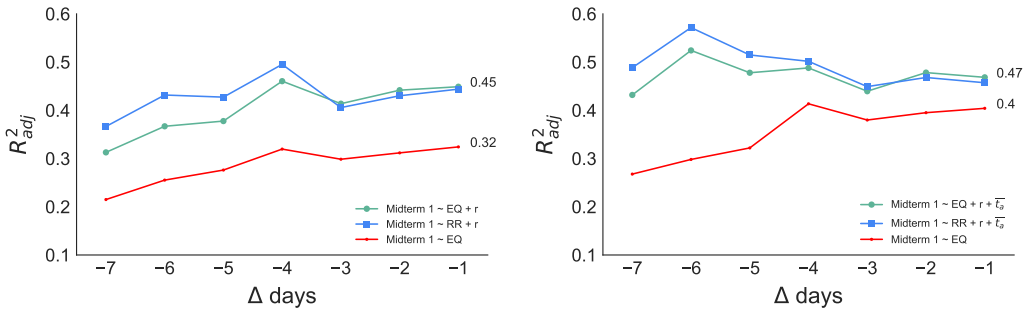
(a) Full data set. (b) Reduced data set.

Figure 7.17: Predictive performance of the multivariate models over time.

The univariate model is outperformed by the multivariate models in both cases, whereas the performance for both the EQ and the RR in the combined models does not vary significantly. However, the univariate model of the EQ shows an improvement when applied to the reduced data set. This implies that the accuracy of the EQ was biased by students who mainly coded on another platform and there were not enough packages available for the EQ to stabilize and find its true value.

When using the same predictors but identifying the first midterm outcome

as the response, all models are generally weaker than the models from Fig. 7.17. However, the univariate model still improves when applied to the reduced data set by more than 30%. This is displayed in Fig. 7.18, where the x-axis represents the days left before the first midterm exam:



(a) Full data set. (b) Reduced data set.

Figure 7.18: Predicting the midterm outcome in week three.

Using only one week of student data, the multivariate models already explain over 30% of the variance and are more accurate than simply using the EQ. This again emphasizes the importance of submission data availability. The predictions of midterm outcomes or overall course outcomes can be used to rank students. Starting from the student who is predicted to perform most poorly, the instructor can then screen the students one by one and decide, if and how to intervene to help the students.

7.5.2 Classification

When building a machine learning classifier that should also be implemented and evaluated across universities, it is important that the same features are also available in that data set. Therefore, two approaches are presented: first, using high-level features and secondly, using fine-grained features at the assignment level similar to [28]. Students with a midterm score lower than 70%, as well as students with an overall course performance lower than 75%, are labeled as at-risk.

Besides the features that have been discussed earlier in this chapter, additional features can be extracted from the data set. These include: number of assignments, broken down by context; number of times restarted the assignment; number of times tabbed out of the environment; average steps to solve auto-grader assignments; average steps to solve an error; number of error states; total number of actions; total number of keystrokes; number of times spent less than 75% time on try-out quizzes; number of times spent more than 75% time on auto-grader quizzes; average time it toke to solve an assignment; average time spent on try-out quizzes; average time spent on auto-grader quizzes; keystroke latency;

Additionally, fine-grained features can be extracted for each assignment - for example an EQ of a specific assignment. For every model, RFE was used to automatically select the five most relevant features.

Weekly Classification Performance

With the data available at the end of each week of the course, we trained and tested classifiers using only high-level features. Beginning with the data available at the end of week two, this yields seven models in total. Table 7.6 lists the classifier metrics for each model.

Table 7.6: Prediction of course failure.

Week	Acc	AUC	κ	F1-Score	at-risk Precision	Specificity
2	0.8	0.85	0.45	0.87	0.5	0.67
3	0.83	0.89	0.74	0.89	0.67	1.0
4	0.89	0.94	0.9	0.93	0.86	1.0
5	0.9	0.96	1.0	0.94	1.0	1.0
6	0.89	0.95	0.79	0.93	0.83	0.83
7	0.89	0.94	0.71	0.93	0.71	0.83
8	0.9	0.95	0.71	0.94	0.71	0.83

The classifier for week two has a relatively weak performance - expressed by the low κ score and the low class-specific precision and recall ratios. Nonetheless, the

overall performance of the classifiers is useful and improves slightly over time. Most importantly, the specificity is on average 88%, which means that on average 88% of the at-risk students are identified.

The most important features selected by RFE are displayed in Fig. 7.19. The number of times a certain feature was selected is given in parenthesis in the figure caption. The figure shows the differences between the two populations, students at-risk of failing the course, and students who are predicted to pass. A two-sided t-test of independence is used to test for a significant difference between the means of the two populations. At a significance level of 5%, p-values lower than 0.05 indicate a significant difference and were calculated for each feature.

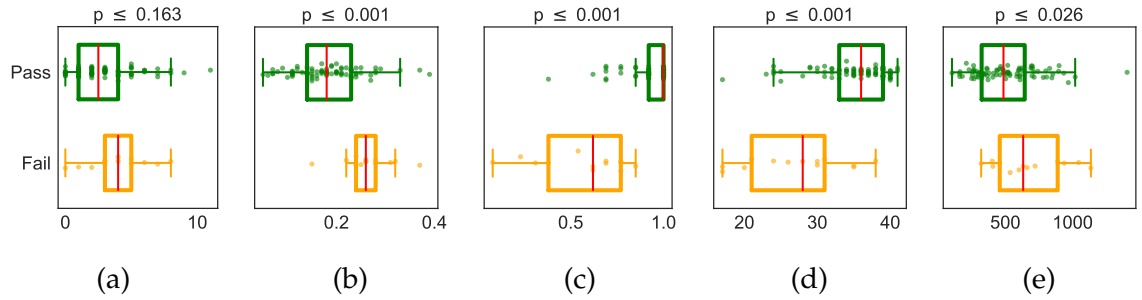


Figure 7.19: Important features for predicting overall course performance (high-level). (a) Number of times the student spent more time on an auto-grader assignment than 75% of their peers (7). (b) EQ of the student (7). (c) Solved ratio for the student (7). (d) Number of unique error states (3). (e) Average time the student spent on auto-grader assignments $\bar{t_a}$ (3).

Three out of the five features that were selected were discussed extensively in the previous sections. The EQ (Fig. 7.19b) as well as the solved ratio r (Fig. 7.19c) were selected as features for all seven models, meaning that the EQ and r contribute significantly to the models' success of correctly classifying at-risk students in every week. Furthermore, $\bar{t_a}$ (Fig. 7.19e) was selected in three out of seven models. It is evident and also backed by the low p-value that at-risk students on average solve fewer auto-grader assignments, spent more time on them and also have a higher EQ.

Moreover, a metric that indicates whether the amount of time a student spent on an auto-grader assignment is greater than the third quantile (Fig. 7.19a) was selected for every model as well. Although there is no significant difference in the means, it is possible that there are other interaction effects between this feature and others that make it important

Lastly, students who have fewer unique error states are more likely to be at-risk (Fig. 7.19d). Recall that for a student who has ten code submissions for an assignment and encountered ten consecutive errors, the number of error states for that assignment is 1. But if the student had one successful compilation event within the ten submissions, for example at the third submission, then there are two error states in that assignment - one before and one after the success. Struggling students might have fewer error states because their error states include more unsuccessful submissions. On the other hand, high-performing students are able to fix an error in fewer steps and have therefore more, but shorter, error states.

Classification Performance in Week Three

As the goal is to intervene as early as possible, daily classifiers were built for up to seven days before the first midterm exam. Similar to the regression models, Δ days indicates the number of days left before the first midterm exam. The classifier performances, trained on the high-level data are presented in Table 7.7.

Table 7.7: Prediction of poor performance on the first midterm exam (high-level features).

Δ days	Acc	AUC	κ	F1-Score	at-risk Precision	Specificity
-7	0.76	0.82	0.39	0.84	0.45	0.71
-6	0.77	0.83	0.52	0.85	0.5	1.0
-5	0.81	0.9	0.58	0.87	0.54	1.0
-4	0.77	0.9	0.6	0.85	0.6	0.86
-3	0.82	0.9	0.56	0.88	0.62	0.71
-2	0.8	0.89	0.56	0.87	0.62	0.71
-1	0.82	0.9	0.66	0.88	0.67	0.86

While not as reliable as the previous models, these models have an average specificity of 83% which makes them still useful. For these classifiers, the most informative high-level features are displayed in Fig. 7.20.

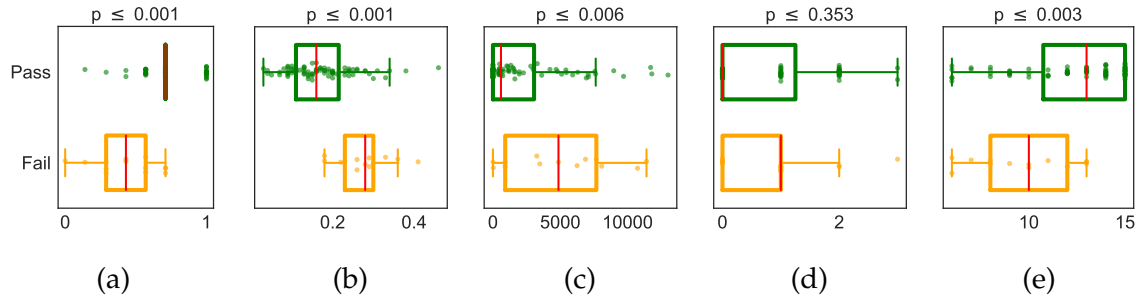


Figure 7.20: Important features for predicting midterm outcome (high-level). (a) Solved ratio of the student (7). (b) EQ of the student (7). (c) Idle time t_i on auto-grader quiz assignments (6). (d) Number of times the student spent more time on an auto-grader quiz assignment than 75% of their peers (3). (e) Number of unique error states (2).

Interestingly, four out of the five most frequently selected features are identical to the most frequent features of the previous model. The feature that was not in the top five features in the previous models is the amount of time a student was idle on auto-grader assignments (Fig. 7.20c). A possible reason for this mean difference is that high-performing students might solve an auto-grader assignment without the need to pause or interrupt their coding. On the other hand, at-risk students possibly either spend more time thinking or browsing for the answer - and passing the idle limit of 400 seconds - or they are distracted or discouraged and do non-course related things when they intended to work on the quiz assignment.

Repeating the same process, but with fine-grained features, results in different models whose performance are displayed in Table 7.8.

Table 7.8: Prediction of poor performance on the first midterm exam (fine-grained features).

Δ days	Acc	AUC	κ	F1-Score	at-risk Precision	Specificity
-7	0.85	0.89	0.63	0.9	0.71	0.71
-6	0.85	0.92	0.6	0.91	0.6	0.86
-5	0.8	0.91	0.66	0.87	0.67	0.86
-4	0.84	0.93	0.74	0.9	0.75	0.86
-3	0.86	0.94	0.66	0.91	0.67	0.86
-2	0.84	0.95	0.66	0.9	0.67	0.86
-1	0.86	0.93	0.74	0.91	0.75	0.86

Comparing these models, which are trained with fine-grained features, to the former models shows an overall increase in model performance. Looking at the selected features in Fig. 7.21, it is evident that the fine-grained features are dominant.

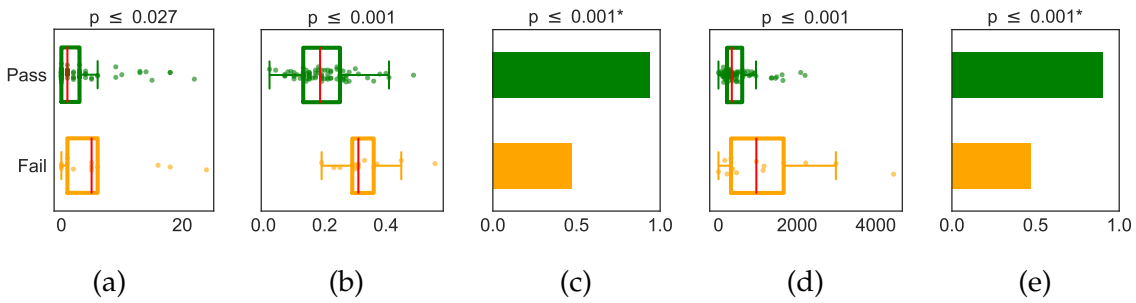


Figure 7.21: Important features for predicting midterm outcome (fine-grained). (a) Number of errors made in quiz assignment four (6). (b) EQ of the student (5). (c) Binary variable - 1 if the student solved quiz assignment three, else 0 (4). (d) Amount of time spent on quiz assignment four (4). (e) Binary variable - 1 if the student solved quiz assignment five, else 0 (4).

Nevertheless, the EQ was selected again five out of seven times (Fig. 7.21b). In addition, two variables from quiz assignment four - which is an auto-grader assignment - were selected: At-risk students encountered more errors on average than their peers on this assignment (Fig. 7.21a) and they also spent on average more time working on it (Fig. 7.21d), indicating that they required more time to solve it. Besides that, two binary variables were selected, which were also derived from auto-grader assignments. The bar plots show the proportions of each group that

solved quiz assignment three and five respectively. The marked p-values were computed using the Fisher's exact test [47]. The test is used to examine whether there is a significant difference between the proportions of one variable among values from the other variable. Both binary variables show that more than 50% of the at-risk students did not solve quiz assignments three and five.

Quiz assignment three was due ten days before the first midterm exam and had a participation of 93% when the first model was built. Quiz assignment four was available one week before the midterm exam and the feature was used for the prediction for almost every day continuously outperforming other features. However, the participation rate at seven days before the midterm exam did not exceed 50%.

The selection of these features might either be a coincidence or show that student success on that assignment directly had an impact on their midterm success. When looking at Fig. 7.22, a principal component analysis [48] of the selected features, it becomes evident why the features have been chosen within the RFE.

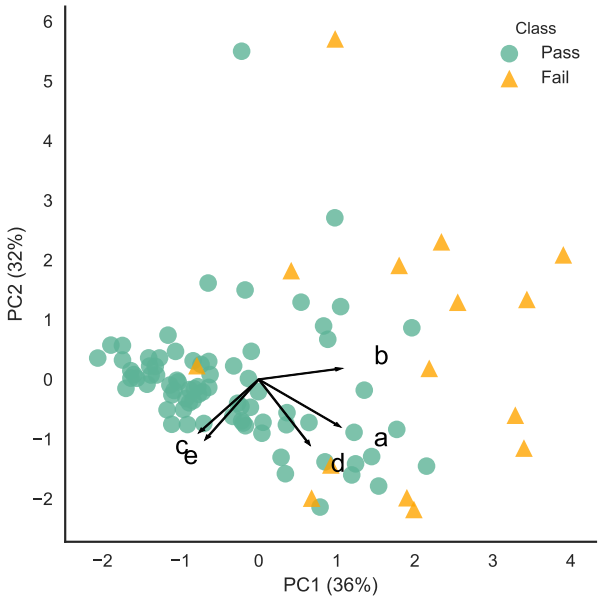


Figure 7.22: Principal Component Analysis of the most common fine-grained features.

While there are not two perfectly distinguishable clusters, it is evident that the closer a student is to the lower left corner, the more likely they are to pass the first midterm. The arrows c and e represent binary features of whether a student solved quiz assignments three and five, respectively. The EQ is represented by b. The higher a student's EQ, the more that student is pushed into the direction of arrow b. Moreover, a and d represent the number of errors and the time spent on quiz assignment four. The two outliers at the top correspond to two students who had no data for assignment four. It is important to note that at-risk students are not similar to each other and do not compose a single cluster. Instead, while they are clearly distinguishable from high-performing students, at-risk students can struggle in many different ways.

A characteristic all three quiz assignments (three, four and five) have in common is that they have the highest participation and, at the same time, more students than average needed more time to solve these programming problems. Furthermore, quiz assignment three has an average EQ of 0.48, which is more than double the overall student mean EQ after week eight (0.19). In addition, 75% of all submissions resulted in errors, indicating that this assignment was difficult for the vast majority. For all three assignments, the students received pre-populated source code.

For quiz assignment three, students were instructed to write a function, which takes two float arguments as an input and returns the sum of the two. This was the first assignment, where students had to write a function.

Quiz assignment four was about branch statements. The students were instructed to read user input. They then had to determine whether the number read is within a given range. Based on the three possible outcomes, a score had to be assigned - by using if/else statements - and returned to the check function.

Quiz assignment five was about loops. Students were instructed to add up the numbers from n to $2n$, where n was set by user-input. This was the first time as well where students had to write a loop.

Discussion

These models can be utilized and integrated into a future classroom setting. Although the models that were trained on fine-grained features performed better, an issue arises when implementing these: In order to be of value, the exact same course setting would need to be retained. Instead, high-level features allow for a more general implementation with an acceptable performance. The classifiers had on average a high specificity, which is important in order to not miss at-risk students. Even though high precision is beneficial and would save time in inspecting falsely flagged students, it is not as crucial for the implementation. Furthermore, the instructor does not need a binary classification. Instead, the logistic regression model can return the probability of a student belonging to either class. This allows the instructor, similar to the regression models, to rank students and follow the same procedure in inspecting students.

Since both approaches, predicting the midterm outcome and the overall course outcome, are useful, the former can be used to continuously rank and monitor students based on their probability score of passing the course. Although the course has a tight schedule and it is of utmost importance to identify and help students before the first midterm exam, it does not mean that students will not struggle afterwards.

Another benefit of performing feature-selection on assignment-level data is that important assignments are highlighted. This allows evaluation of these assignments from a pedagogical point of view and gathering insights as to why certain assignments contribute to student learning more than others. Applying these methods to a variety of courses at different universities has the potential of collaboratively engineering new assignments that have a greater impact on student learning.

Chapter 8

Discussion

Following the analysis, this chapter discusses limitations that need to be considered when results of the analysis are compared to other work. In addition, the key findings are summarized and the work is concluded with implications for future work.

8.1 Limitations

Several circumstances limit the results and methods applied in this thesis. For one, the error compilation metrics were developed and tested with data gathered from Java programming courses as described in Chapter 5. Findings of our analysis might not be directly transferable to other studies, although the results strongly indicate a suitability of these metrics across programming languages.

Furthermore, the main findings on time dependent variables are limited by a smaller data set, caused by students, who did not use our coding environment as extensively as would have been necessary to collect enough data. Partly, this was caused by the absence of critical features from our coding environment for many students, such as the ability to interact with the compiled program.

In addition, as for all studies that use assignment data within their analysis, it is

difficult to compare this data to data from other universities, when the assignments are not similar at all. This issue can partly be addressed by cross-validating our results with a data set from the same course in the next academic year.

Cross-validation with another data set is necessary for the prediction models as well, because the models have only been trained and tested on a small data set. Ideally these models should also be cross-study validated.

8.2 Conclusion

This work shows a successful integration of an online coding environment within an online learning management system of an on-campus course. It is demonstrated how the data from such a course setting can be utilized to gather insights on student learning and to improve the students' learning experience.

The most common compiler errors that students received have been identified correctly by using a data set from a past course with only a fraction of available data compared to the main data set.

Furthermore, three error compilation metrics that were initially developed in a Java course setting were evaluated in our course setting and showed a strong correlation with student exam performances and their overall course performance. It was also found that these metrics are not dependent on a context, but rather their accuracy relies on the amount of available data. While the predictive performance of the EQ and the RR was good enough to build predictive models, the RED definitely needs more tweaking in order to be as useful as the alternatives.

The implementation of an auto-grader system was furthermore very successful. Not only was the direct feedback helpful for the students, but the two metrics derived from the system were proven to be two very useful predictors of student performance. These metrics included the number of auto-grader quiz assignments

a student solved compared to their peers, as well as the amount of time a student needs on average to solve auto-grader quiz assignments. A multivariate linear regression model, built with these two metrics and with either the EQ or the RR, showed superior performance compared to univariate models built with either the EQ or the RR, two weeks into the course and with only one week of coding data. For the time-dependent variables, it was perceptible that it is important to separate the data based on the context.

Machine learning was used to build predictive models that could identify at-risk students within a week before the first midterm exam, leaving enough time for an intervention to support these students. It was shown that these models can be useful to assist the course instructor in identifying at-risk students by ordering the students based on their class passing probabilities. A positive side effect of using machine learning models is to gain insights on important assignments and also the possibility to detect new indicators for student performance.

Overall, these findings lead to the conclusion that a data-driven approach in introductory programming classes can not only improve the learning experience for students, but also the teaching experience for educators. This benefit is maximized, if the characteristics of a data-driven approach are considered while designing the course, which among others includes the front-loading of coding assignments to have enough data available for building accurate models as early as possible. While the auto-grader and the enhanced error messages already helped this course, the findings of student interactions with specific assignments as well as the prediction models will be useful for future courses.

8.3 Future Work

This was the first full integration of a coding environment into an introductory programming course. Among the obvious benefits to novice programming students is their ability to start coding instantly in the environment, without having to overcome the typical hurdles presented by coding environments. In addition it allowed the instructor to oversee the activity and to offer fast and direct remote assistance to support-seeking students.

Nevertheless, there is still much to improve within the coding environment. Addressing frequent problems that students had when interacting with the platform and also improving and enhancing the server-sided logging are two major areas of future work. The focus should be on adding features that are necessary for allowing students to work on more complex programming problems. In addition, features that are directed toward the instructor and the TAs can also be improved to ease grading and supporting-giving processes. One option is to implement a code submission option within the environment that allows students to submit their solution through the coding environment, linked to Canvas by utilizing its API. Improvements of this type on the coding environment could probably entice the more advanced students to use the coding environment by choice. These students, who currently often prefer to code in different environments, are a critical part of this type of analysis. In our analysis, data from such inactive students had to be excluded. Another important feature is the actual implementation of the predictive models into an admin dashboard for the instructor, that would automatically run the models and visualize the results, most importantly highlighting flagged students. Once such a system is implemented, intervention strategies can be developed and monitored on how to best help students.

Because our analysis revealed the importance of short programming assignments with automated feedback that cover basic programming principles, a focus

should be on including more of these and a further development of the auto-grader system. The auto-grader can be improved by adding more test cases to check functions and evaluate correctness. Another area is the improvement of the auto-grader workflow into a more manageable structure for the instructor.

A different approach would be the integration of an auto-grader system that utilizes abstract syntax trees for correctness checking [18]. This seems to be the most promising approach for auto-grader systems, but it is also more challenging, especially for the C programming language. Incorporating abstract syntax trees has the additional benefit of opening the possibility of implementing the automated hint-generation system that was discussed briefly in Chapter 6.

A post-course survey revealed that students generally found friendly error messages helpful. By taking advantage of teaching two classes simultaneously in the future, this impression can be validated by only activating the enhanced error system in an intervention group and comparing, for example, student performance metrics to a control group. Besides validating the effectiveness of friendly error messages, a system can be implemented in the future that enables students to rate the helpfulness of the friendly error messages the moment they occur, automatically identifying friendly error messages that are not useful. An improvement that would demand more work, but would be even more beneficial for students, is to develop and implement a system similar to the *HelpMeOut* system [11] that collects student fixes and provides these as suggestions to a student with the same error. Since error messages in C have multiple causes, a data-driven approach could identify common causes for an error, based on the programming assignment and provide tailored explanations specific to that assignment. Further personalizing friendly error messages by incorporating source code specific information, such as the variable names, is another opportunity to improve the effectiveness of friendly error messages.

Future work should also focus on addressing limitations that were previously discussed. This includes further validation of the prediction models. Moreover, the view of our analysis was limited to the work presented in this thesis, meaning that there is still more data to be explored. As the feature selection process revealed, there is even more information hidden in the details. This requires a separate analysis and discussion of features that are already available in the data set or were not even explored yet.

Ultimately, these findings have the potential to reveal more information on how students learn to program and how this information can be utilized to improve the teaching of programming.

Bibliography

[1] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd. Morgan Kaufmann Publishers Inc., 2011.

[2] International Educational Data Mining Society, [Online]. Available: `http://www.educationaldatamining.org/` (visited on 05/06/2017).

[3] R. S. Baker and K. Yacef, "The state of educational data mining in 2009: A review and future visions," *Journal of Educational Data Mining*, vol. 1, no. 1, 2009.

[4] C. Romero and S. Ventura, "Educational data mining: A review of the state of the art," *Transactions on Systems, Man, and Cybernetics, Part C*, vol. 40, no. 6, pp. 601–618, 2010.

[5] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. Á. 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*, ser. ITICSE-WGR '15, ACM, 2015, pp. 41–63.

[6] T. Flowers, C. A. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through gauntlet," in *34th Annual Frontiers in Education, 2004. FIE 2004.*, 2004, T3H/10–T3H/13 Vol. 1.

[7] V. J. Traver, "On compiler error messages: What they say and what they mean," *Advances in Human-Computer Interaction*, vol. 2010, 3:1–3:26, 2010.

[8] J. Jackson, M. Cobb, and C. Carver, "Identifying top java errors for novice programmers," in *Proceedings Frontiers in Education 35th Annual Conference*, 2005, T4C–T4C.

[9] B. A. Becker, "An effective approach to enhancing compiler error messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16, ACM, 2016, pp. 126–131.

[10] B. A. Becker, "An exploration of the effects of enhanced compiler error messages for computer programming novices," Dublin Institute of Technology, Master's thesis, 2015.

[11] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, ACM, 2010, pp. 1019–1028.

[12] Z. Alharbi, J. Cornford, L. Dolder, and B. D. L. Iglesia, "Using data mining techniques to predict students at risk of poor performance," in *2016 SAI Computing Conference (SAI)*, 2016, pp. 523–531.

[13] C. Romero, S. Ventura, and E. García, "Data mining in course management systems: Moodle case study and tutorial," *Computers & Education*, vol. 51, no. 1, pp. 368–384, 2008.

[14] L. P. Macfadyen and S. Dawson, "Mining LMS data to develop an "early warning system" for educators: A proof of concept," *Computers & Education*, vol. 54, no. 2, pp. 588–599, 2010.

[15] Y.-H. Hu, C.-L. Lo, and S.-P. Shih, "Developing early warning systems to predict students' online learning performance," *Computers in Human Behavior*, vol. 36, no. C, pp. 469–478, 2014.

[16] S. B. Kotsiantis, C. J. Pierrakeas, and P. E. Pintelas, "Preventing student dropout in distance learning using machine learning techniques," in *Knowledge-Based Intelligent Information and Engineering Systems: 7th International Conference, KES 2003, Oxford, UK, September 2003. Proceedings, Part II*, V. Palade, R. J. Howlett, and L. Jain, Eds. Springer Berlin Heidelberg, 2003, pp. 267–274.

[17] K. R. Koedinger, E. Brunskill, R. S. J. de Baker, E. A. McLaughlin, and J. C. Stamper, "New potentials for data-driven intelligent tutoring system development and optimization," *AI Magazine*, vol. 34, pp. 27–41, 2013.

[18] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: A self-improving python programming tutor," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017.

[19] M. C. Jadud, "An exploration of novice compilation behavior," University of Kent, Canterbury, Doctoral dissertation, 2006.

[20] M. M. T. Rodrigo, E. S. Tabanao, M. B. E.Lahoz, and M. C. Jadud, "Analyzing online protocols to characterize novice java programmers," *Philippine Journal of Science*, vol. 138, no. 2, pp. 177–190, 2009.

[21] M. C. Jadud and B. Dorn, "Aggregate compilation behavior: Findings and implications from 27,698 users," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15, ACM, 2015, pp. 131–139.

[22] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice java programmers through the analysis of online protocols," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11, ACM, 2011, pp. 85–92.

[23] 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*, ser. ICALT '13, IEEE Computer Society, 2013, pp. 319–323.

[24] C. Watson, "An exploration of traditional and data driven predictors of programming performance," Durham University, Doctoral dissertation, 2015.

[25] A. Petersen, J. Spacco, and A. Vihavainen, "An exploration of error quotient in multiple contexts," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, ser. Koli Calling '15, ACM, 2015, pp. 77–86.

[26] B. A. Becker, "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*, ser. ITiCSE '16, ACM, 2016, pp. 296–301.

[27] 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*, ser. SIGCSE '14, ACM, 2014, pp. 469–474.

[28] 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*, ser. ICER '15, ACM, 2015, pp. 121–130.

[29] K. Castro-Wunsch, A. Ahadi, and A. Petersen, "Evaluating neural networks as a method for identifying students in need of assistance," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17, ACM, 2017, pp. 111–116.

[30] J. Leinonen, K. Longi, A. Klami, and A. Vihavainen, "Automatic inference of programming performance and experience from typing patterns," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16, ACM, 2016, pp. 132–137.

[31] E. B. Costa, B. Fonseca, M. A. Santana, F. F. de Arajo, and J. Rego, "Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses," *Computers in Human Behavior*, vol. 73, no. C, pp. 247–256, 2017.

[32] A. Müller and S. Guido, *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly, 2016.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine Learning*, vol. 46, no. 1, pp. 389–422, 2002.

[35] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, 2002.

[36] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[37] D. Montgomery and E. Peck, *Introduction to linear regression analysis*, ser. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley, 2001.

[38] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Identifying at-risk novice Java programmers through the analysis of online protocols," *Philippine Computing Science Congress*, 2008.

[39] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, ACM, 2014, pp. 223–228.

[40] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the Second International Workshop on Computing Education Research*, ser. ICER '06, ACM, 2006, pp. 73–84.

[41] G. Braught, T. Wahls, and L. M. Eby, "The case for pair programming in the computer science classroom," *Transations on Computing Education*, vol. 11, no. 1, 2:1–2:21, 2011.

[42] Canvas. How do I view analytics for a student in a course? [Online]. Available: `https://community.canvaslms.com/docs/DOC-10297` (visited on 05/25/2017).

[43] Ace. The High Performing Code Editor for the Web, [Online]. Available: `https://ace.c9.io/` (visited on 05/09/2017).

[44] S. S. Shaprio and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[45] F. J. Anscombe and W. J. Glynn, "Distribution of the kurtosis statistic b2 for normal samples," *Biometrika*, vol. 70, no. 1, pp. 227–234, 1983.

[46] R. B. D'Agostino, A. Belanger, and D. J. R. B., "A suggestion for using powerful and informative tests of normality," *The American Statistician*, vol. 44, no. 4, pp. 316–321, 1990.

[47] J. H. McDonald, *Handbook of Biological Statistics*. Sparky House Publishing, 2014, vol. 3.

[48] I. T. Jolliffe, *Principal Component Analysis*. Springer New York, 2013.