FACULTY
OF SCIENCES

DODONA

IMPROVING PROGRAMMING EDUCATION THROUGH AUTOMATED ASSESSMENT, LEARNING ANALYTICS, AND EDUCATIONAL DATA MINING

Charlotte Van Petegem

Supervisors:
Prof. Dr. Peter Dawyndt
Prof. Dr. Ir. Bart Mesuere
Prof. Dr. Bram De Wever

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of Doctor of Computer Science.

Academic year: 2023–2024

GHENT
UNIVERSITY

Table of Contents

Dankwoord

Het dankwoord. Ondertussen het stuk van mijn doctoraat dat ik het meeste heb uitgesteld. Niet omdat ik geen mensen wil bedanken, maar omdat ik dat goed en zeer oprecht wil doen.[1] De voorbije zes jaar is er veel gebeurd. Er zijn ups en downs geweest. Er zijn een hoop individuen die mij in die zes jaar gesteund hebben wanneer het moeilijk ging, blij voor mij waren wanneer het goed ging, en mij allemaal op hun manier geholpen hebben. Die verdienen hier allemaal een bedanking.

De eerste bedanking gaat uit naar mijn promotoren. Peter, Bart en Bram, bedankt voor jullie begeleiding de voorbije jaren. Het spreekt voor zich dat mijn doctoraat er niet zou gekomen zijn zonder jullie.

Ook mijn jury wil ik graag bedanken. Hun kritische vragen en opmerkingen hebben de finale tekst van mijn doctoraat zonder enige twijfel verbeterd. Christophe, Wesley, Frank, Kim en Raija, bedankt! Verder zou ik graag Dominique willen bedanken om de rol als voorzitter van mijn jury op te nemen.

Een eerder atypische bedanking gaat uit naar alle artiesten waarvan ik de muziek gebruikt heb om tijdens het schrijven van mijn doctoraat de concentratie te behouden.[2] Dit zijn Anohni, Boygenius[3], Charlotte Cardin, Eliza McLamb, Jan Swerts, Katy Kirby, Marika Hackman, Pinegrove, SOPHIE, Spinvis en Tate McRae.

Veel van mijn tijd aan de UGent is naast onderzoek ook aan onderwijs gespendeerd. Daarbij had ik veel contact met mijn mede-begeleiders. Merci aan Adnan, Annick, Antoine, Dieter, Ellen, Felipe, Felix, Henri, Lotte, Louise, Niko, Oliver, Silvija, Tibo, Toon en Yentl voor de fijne samenwerking.

[1] Het dankwoord is vaak ook het meest-gelezen deel van een doctoraat, wat het uitstelgedrag alleen maar in de hand heeft gewerkt.

[2] Ik limiteer me tot de periode van het schrijven van mijn doctoraat, want als ik alles had opgelijst dat die rol vervuld heeft in de voorbije zes jaar zou dit boek een stuk dikker geworden zijn.

[3] En ook het solo-werk van Lucy Dacus, Phoebe Bridgers en Julien Baker.

Er zijn ook nog veel andere collega's die mijn tijd als doctoraatsstudent aan de UGent opgefleurd hebben en waarvan ik blij ben dat ik velen als vrienden mag benoemen. Ik heb genoten van de vele lunchpauzes, koffiepauzes, spelletjesavonden, TWIkends, … Daarom ook bedankt aan Alexis, Asums, Benjamin, Dieter, Felix, Heidi, Jonathan, Jorg, Louise, Mustapha, Nico, Niels, Niko, Oliver, Pieter Goetschalckx, Pieter Verschaffelt, Rien, Robbert, Roy, Simon, Steven, Tibo, Tom en Toon.

De voorbije vier jaar was ik ook lid van de faculteitsraad. Boris, Evert, Francis, Jonathan, Jozefien, Lien, Pieter, Tom en Toon, mijn collega-AAP-vertegenwoordigers, wil ik bedanken voor de samenwerking de voorbije jaren.

Ook vanuit mijn leven buiten de UGent wil ik hier de kans nemen om een hoop mensen te bedanken. Sowieso voltallig Zeus WPI, met eervolle vermelding van Jasper. Met jullie over alles en niets discussiëren op Mattermost was niet altijd bevorderlijk voor het schrijven van mijn doctoraat, maar ik heb er mij toch mee geamuseerd. Tijdens de lockdown-periodes werden mijn woensdagavonden ook vaak opgevrolijkt door Bart, Felix, Heidi, Rien, Robbert, Steven en Toon.[4] Het boulderen de voorbije maanden met Tom "gewoon doorstappen"[5] Naessens, Felix, Titouan, Deborah, Ruben, Arthur, Nicky, Francis, Tom Lauwaerts, Jorg, Louise, Rien, Charlotte en Heidi was een zeer goeie manier om mijn gedachten te verzetten na een dag schrijven, en ik ben zeer blij in het boulderen een nieuwe hobby met veel leuke mensen gevonden te hebben. Bij deze categorie horen ook nog de mensen met wie ik D&D speel (of gespeeld heb). Ook bedankt dus aan Alexis, Arne, Asmus, Bart, Felix, Heidi, Jorg, Kenneth, Lea, Louise, Maxiem en Niko. Ten laatste, mijn mede-vrijwilligers bij het Rode Kruis: Henk, Jietse, Jonas, Kristel, Luc, Nancy, Pascal, Philippe, Rien, Rikie, Sarah, Wim, Wouter, …

Een naam waarvan ik niet wil dat die hierboven verloren gaat in de lange lijsten aan namen en die ook een speciale bedanking verdient is die van Dieter. Dieter, jij was degene die mij echt heeft doen thuisvoelen aan de vakgroep. Ik heb je enorm gemist toen je je doctoraat afgewerkt had en naar andere oorden vertrokken was, en ik ben heel blij dat we elkaar nog steeds regelmatig zien.

De andere persoon die ik hier nog even apart wil bedanken is Toon. Toon komt voor in ongeveer alle categorieën hierboven die niet met onderzoek

[4] Al ben ik 0 A.D. ondertussen wel volledig beu gespeeld, vrees ik.

[5] Een uitspraak die ik in mijn achterhoofd heb gehouden wanneer het schrijven van mijn doctoraat wat trager ging.

te maken hebben, maar eigenlijk verdient hij ook daarbij een bedanking. Toon, je stond altijd klaar voor een babbel, over welk onderwerp dan ook. In jouw dankwoord sprak je je verbazing uit dat we zo goed overeen kunnen komen, ook al zijn we zeer verschillend. Ik kan dat gevoel alleen maar beamen, maar ik denk dat een van de redenen dat we zo goed overeen komen is dat we elkaar uitdagen, op vele vlakken. Doordat je tegenwoordig in Berlijn zit zien we elkaar niet zo heel vaak meer, maar gelukkig horen we elkaar nog ietwat regelmatig.

Ten laatste, de groep die het meest voor het uitstellen van dit dankwoord te schrijven heeft gezorgd: mijn familie. Woorden kunnen niet uitdrukken hoeveel steun ik gehad heb van mijn familie in de voorbije zes jaar. Ik vind het dus ook moeilijk om uit te drukken hoe hard ik ze wil bedanken. Mama, papa, Hannelore, Tomas, Seppe[6], Robbe, Esther, Kero Kero[7]: bedankt, bedankt, bedankt. Duizendmaal bedankt.

Charlotte Van Petegem

2024-06-19

[6]Het beste metekindje ter wereld.
[7]Ja, ook mijn kat verdient een bedanking.

Summary in English

Ever since programming has been taught, its teachers have sought to automate and optimize their teaching. Due to the ever-increasing digitalization of society, programming is also being taught to ever more and ever larger groups, and these groups often include students for whom programming is not necessarily their main subject. This has led to the development of myriad automated assessment tools (Ala-Mutka, 2005; Douce et al., 2005; Ihantola et al., 2010; Paiva, Leal, et al., 2022). One of those platforms is Dodona[8], which is the platform this dissertation is centred around.

Chapters 2, 3, and 4 focus on Dodona itself. In Chapter 2 we give an overview of the user-facing features of Dodona, from user management to how feedback is represented. Chapter 3 then focuses on how Dodona is used in practice, by presenting some facts and figures of its use, students' opinions of the platform, and an extensive case study on how Dodona's features are used to optimize teaching. This case study also provides insight into the educational context for the research described in Chapters 5 and 6. Chapter 4 focuses on the technical aspects of developing Dodona and its related ecosystem of software tools. This includes a discussion of the technical challenges related to developing a platform like Dodona, and how the Dodona team adheres to modern standards of software development.

Chapters 5 and 6 are a bit different. These chapters each detail a learning analytics/educational mining study we did, using the data that Dodona collects about the learning process. Learning analytics and educational data mining stand at the intersection of computer science, data analytics, and the social sciences, and focus on understanding and improving learning. They are made possible by the increased availability of data about students who are learning, due to the increasing move of education to digital platforms (Romero et al., 2008). They can also serve different actors in the educational landscape: they can help learners directly, help teachers to evaluate their own teaching, allow developers of education platforms to know what to focus on, allow educational institutions to

[8]`https://dodona.be`

guide their decisions, and even allow governments to take on data-driven policies (Ferguson, 2012).

Chapter 5 discusses a study where we tried to predict whether students would pass or fail a course at the end of the semester based solely on their submission history in Dodona. It also briefly details a study we collaborated on with researchers from Jyväskylä University in Finland, where we replicated our study in their educational context, with data from their educational platform.

In Chapter 6, we first give an overview of how Dodona changed manual assessment in our own educational context. We then finish the chapter with some recent work on a machine learning method we developed to predict what feedback teachers will give when manually assessing student submissions.

Finally, Chapter 7 concludes the dissertation with some discussion on Dodona's opportunities and challenges for the future.

Nederlandstalige samenvatting

Al van bij de start van het programmeeronderwijs, proberen docenten hun taken te automatiseren en optimaliseren. De digitalisering van de samenleving gaat ook steeds verder, waardoor steeds meer en grotere groepen studenten leren programmeren. Deze groepen bevatten ook vaker studenten voor wie programmeren niet het hoofdonderwerp van hun studies is. Dit heeft geleid tot de ontwikkeling van zeer veel platformen voor de geautomatiseerde beoordeling van programmeeropdrachten (Ala-Mutka, 2005; Douce et al., 2005; Ihantola et al., 2010; Paiva, Leal, et al., 2022). Eén van deze platformen is Dodona[9], het platform waar dit proefschrift over handelt.

Hoofdstukken 2, 3 en 4 focussen op Dodona zelf. In Hoofdstuk 2 geven we een overzicht van de gebruikersgerichte features van Dodona, van gebruikersbeheer tot hoe feedback getoond wordt. Hoofdstuk 3 focust zich dan op hoe Dodona in de praktijk gebruikt wordt, door statistieken over het gebruiken te presenteren, de meningen van studenten over het platform te presenteren en met een uitgebreide case study waarin getoond wordt hoe de verschillende features van Dodona kunnen bijdragen tot het optimaliseren van onderwijs. Deze case study presenteert ook de context waarin Hoofdstukken 5 en 6 zich situeren. Hoofdstuk 4 focust op het technische aspect van het ontwikkelen van Dodona en het ecosysteem van software gerelateerd aan Dodona. Dit bevat onder meer een bespreking van de technische uitdagingen gerelateerd aan het ontwikkelen van een platform zoals Dodona en hoe het Dodona-team zich aan de moderne standaarden van softwareontwikkeling houdt.

Hoofdstukken 5 en 6 verschillen van de vorige hoofdstukken, in de zin dat ze elk een *learning analytics/educational data mining* studie bespreken. Deze studies werden uitgevoerd met de data die Dodona verzamelt over het leerproces. *Learning analytics* en *educational data mining* bevinden zich op het kruispunt tussen informatica, datawetenschap en de sociale wetenschappen, en focussen zich op het begrijpen en verbeteren van leren.

[9] `https://dodona.be`

Ze worden mogelijk gemaakt door de toegenomen beschikbaarheid van data over lerende studenten, wat op zijn beurt komt door de toegenomen beweging van onderwijs naar digitale platformen (Romero et al., 2008). Ze kunnen ook dienen voor verschillende actoren in het onderwijsveld: ze kunnen studenten direct helpen, docenten helpen om hun eigen onderwijs te evalueren, ontwikkelaars van onderwijsplatformen laten weten waar ze zich op moeten focussen, de beslissingen van onderwijsinstellingen helpen gidsen, en zelfs overheden toelaten om op data gebaseerd beleid te voeren (Ferguson, 2012).

Hoofdstuk 5 bespreekt een studie waarin we geprobeerd hebben te voorspellen of studenten al dan niet zouden slagen voor een vak op het einde van het semester, enkel en alleen gebaseerd op hun indiengedrag op Dodona. Daarnaast wordt er kort een samenwerking besproken met onderzoekers van de universiteit van Jyväskylä in Finland, waar we onze studie herhaald hebben in hun educationele context, gebruikmakend van data afkomstig van hun platform.

In Hoofdstuk 6 geven we eerst een overzicht van hoe Dodona het manueel verbeteren in onze eigen educationele context veranderd heeft. We sluiten dan het hoofdstuk af met een recent door ons ontwikkelde *machine-learning*-methode om te voorspellen welke feedback docenten zullen geven tijden het manueel verbeteren van indieningen van studenten.

We sluiten af in Hoofdstuk 7 met een bespreking van de mogelijkheden en uitdagingen waar Dodona in de toekomst voor staat.

List of Publications

Van Petegem, C., Demeyere, K., Maertens, R., Strijbol, N., De Wever, B., Mesuere, B., Dawyndt, P., submitted. Mining patterns in syntax trees to automate code reviews of student solutions for programming exercises. *International Journal of Artificial Intelligence in Education.*

Strijbol, N., Sels, B., **Van Petegem, C.**, Maertens, R., Scholliers, C., Mesuere B., Dawyndt, P., submitted. TESTed-DSL: a domain-specific language to create programming exercises with language-agnostic automated assessment. *Software Testing, Verification and Reliability.*

Maertens, R., Van Neyghem, M., Geldhof, M., **Van Petegem, C.**, Strijbol, N., Dawyndt, P., Mesuere, B., 2024. Discovering and exploring cases of educational source code plagiarism with Dolos. *SoftwareX* 26, 101755. `https://doi.org/10.1016/j.softx.2024.101755`

Zhidkikh, D., Heilala, V., **Van Petegem, C.**, Dawyndt, P., Järvinen, M., Viitanen, S., De Wever, B., Mesuere, B., Lappalainen, V., Kettunen, L., & Hämäläinen, R., 2024. Reproducing Predictive Learning Analytics in CS1: Toward Generalizable and Explainable Models for Enhancing Student Retention. *Journal of Learning Analytics*, 1-21. `https://doi.org/10.18608/jla.2024.7979`

Van Petegem, C., Maertens, R., Strijbol, N., Van Renterghem, J., Van der Jeugt, F., De Wever, B., Dawyndt, P., Mesuere, B., 2023. Dodona: Learn to code with a virtual co-teacher that supports active learning. *SoftwareX* 24, 101578. `https://doi.org/10.1016/j.softx.2023.101578`

Strijbol, N., **Van Petegem, C.**, Maertens, R., Sels, B., Scholliers, C., Dawyndt, P., Mesuere, B., 2023. TESTed – An educational testing framework with language-agnostic test suites for programming exercises. *SoftwareX* 22, 101404. `https://doi.org/10.1016/j.softx.2023.101404`

Van Petegem, C., Deconinck, L., Mourisse, D., Maertens, R., Strijbol, N., Dhoedt, B., De Wever, B., Dawyndt, P., Mesuere, B., 2022. Pass/Fail Prediction in Programming Courses. *Journal of Educational Computing Research* 68–95. `https://doi.org/10.1177/07356331221085595`

Maertens, R., **Van Petegem, C.**, Strijbol, N., Baeyens, T., Jacobs, A.C., Dawyndt, P., Mesuere, B., 2022. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning.* `https://doi.org/10.1111/jcal.12662`

Nüst, D., Eddelbuettel, D., Bennett, D., Cannoodt, R., Clark, D., Daróczi, G., Edmondson, M., Fay, C., Hughes, E., Kjeldgaard, L., Lopp, S., Marwick, B., Nolis, H., Nolis, J., Ooi, H., Ram, K., Ross, N., Shepherd, L., Sólymos, P., Swetnam, T.L., Turaga, N., **Van Petegem, C.**, Williams, J., Willis, C. Xiao, N., 2020. The Rockerverse: Packages and Applications for Containerisation with R. *The R Journal*, 12(1), 437–461. `https://doi.org/10.32614/RJ-2020-007`

List of Software Artefacts

Development of software was an important part of the work that went into this PhD. Chapter 4 discusses (part of) the development of Dodona and its ecosystem of software, but for convenience, all repositories that I worked on as part of this PhD are listed below, together with a short description. For brevity, only open-source repositories that are used in production are listed.

Dodona

`https://github.com/dodona-edu/dodona`

This is the main Dodona repository. It contains all Dodona's application code, from frontend to backend, and the machinery required to run background jobs. Dodona is a Ruby-on-Rails web application. Lightweight web components are used to add interactivity to the frontend.

Dodona documentation

`https://github.com/dodona-edu/dodona-edu.github.io`

Dodona is used by many people, and thus needs extensive documentation. This repository contains all user-facing documentation, both for teachers and students, and is accessible in a user-friendly format on `https://docs.dodona.be/`. This includes guides on how to get started with Dodona, how to add new exercises to Dodona, and much more.

TESTed

`https://github.com/dodona-edu/universal-judge`

TESTed is a universal judge, in that exercise authors only have to create an exercise once to have it be available in all the programming languages TESTed supports. Supported programming languages include Python, JavaScript, Java, Kotlin, C#, C, Haskell, …

R judge

`https://github.com/dodona-edu/judge-r`

Judge for the R programming language. This judge also has support for showing generated figures in the feedback and can even do introspection on GGPlot objects.

Papyros

`https://github.com/dodona-edu/papyros`

Web IDE that can execute Python code in the browser, using Pyodide to do so. It also has a built-in debugger, based on the Python Tutor.

Docker images

`https://github.com/dodona-edu/docker-images`

This repository contains Dockerfiles corresponding to Dodona's judges. These Dockerfiles make sure every judge has all the libraries and binaries required for the judge to function.

1 Introduction

Ever since programming has been taught, its teachers have sought to automate and optimize their teaching. Due to the ever-increasing digitalization of society, programming is also being taught to ever more and ever larger groups, and these groups often include students for whom programming is not necessarily their main subject. This has led to the development of myriad automated assessment tools (Ala-Mutka, 2005; Douce et al., 2005; Ihantola et al., 2010; Paiva, Leal, et al., 2022), of which we give a historical overview in this introduction. We also discuss learning analytics and educational data mining, and how these techniques can help us to cope with the growing class sizes. We then give an overview of programming education in Flanders, including recent societal changes around this topic. Afterwards, we give a brief overview of the remaining chapters of this dissertation. We then conclude the introduction with a brief section on the ethics and research integrity of the work presented in this dissertation.

1.1 Automated assessment in programming education

Increasing interactivity in learning has long been considered important, and also something that can be achieved through the addition of (web-based) IT components to a course (Van Petegem et al., 2004). This is also the case when learning to program: learning how to solve problems with computer programs requires practice, and programming assignments are the main way in which such practice is generated (Gibbs & Simpson, 2005). Cheang et al. (2003) identified the labor-intensive nature of assessing programming assignments as the main reason why students are given only few assignments when in an ideal world they should be given many more. Automated assessment allows students to receive immediate and personalized feedback on each submitted solution without the need for human intervention. Because of its potential to provide feedback loops that are scalable and responsive enough for an active learning environment, automated source code assessment has become a driving force in programming courses.

1.1.1 Humble beginnings

Automated assessment was introduced into programming education in the late 1950s (Hollingsworth, 1960). In this first system, programs were submitted in assembly on punch cards. For the reader who is not familiar with punch cards, an example of one can be seen in Figure 1.1. The assessment was then performed by combining the student's punch cards with the autograder's punch cards. In the early days of computing, the time of tutors was not the only valuable resource that needed to be shared between students; the actual compute time was also a shared and limited resource. Their system made more efficient use of both. Hollingsworth (1960) already notes that the class sizes were a main motivator to introduce their auto-grader. At the time of publication, they had tested about 3 000 student submissions which, given a grading run took about 30 to 60 seconds, represents about a day and a half of computation time.

They also immediately identified some limitations, which are common problems that modern assessment systems still need to consider. These limitations include handling faults in the student code, making sure students can not modify the grader, and having to define an interface through which the student code is run.

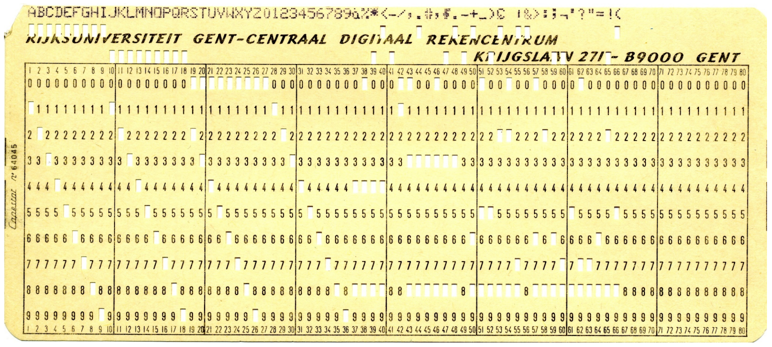


Figure 1.1: Example of a punch card. Scan from the archive of Ludo Coppens, provided by Bart Coppens in personal correspondence.

In the next ten years, significant advances were already made. Students could submit code written in a text-based programming language instead of assembly, and the actual testing was done by running their code using modified compilers and operating systems.

Naur (1964) was the first to explicitly note the difference between the formal correctness, and the efficiency and completeness of the programs being tested. The distinction between formal correctness and completeness that he makes can be somewhat confusing from a modern standpoint: we would only consider a program or algorithm formally correct if it is complete (i.e. gives the correct response in all cases). In more modern terminology, Naur's "formally correct" would be called "free of syntax errors".

Forsythe & Wirth (1965) note another issue when using automatic graders: students could use the feedback they get to hard-code the expected response in their programs. This is also still an issue that modern assessment systems (or the teachers creating exercises) need to consider. Forsythe & Wirth solve this issue by randomizing the inputs to the student's program. While not explicitly explained by them, we can assume that to check the correctness of a student's answer, they calculate the expected answer in the grader program as well. Note that in this system, they were still writing a grading program for each individual exercise.

Hext & Winings (1969) introduce another innovation: their system could be used for exercises in multiple different programming languages. They are also the first to implement a history of student's attempts in the assessment tool itself, and mention explicitly that enough data should be recorded in this history so that it can be used to calculate a mark for a student.

Other grader programs were in use at the time, but these did not necessarily bring any new innovations or ideas to the table (Berry, 1966; Braden & Perlis, 1965; Temperly & Smith, 1968).

The systems described above share an important limitation, which is inherent to the time at which they were built. Computers were big and heavy, and had operators who did not necessarily know whose program they were running or what those programs were. The Mother of All Demos by Engelbart & English (1968), widely considered the birth of the *idea* of the personal computer, only happened after these systems were already running. So, it should not come as a surprise that the feedback these systems gave was slow to return to the students.

1.1.2 Tool- and script-based assessment

We now take a leap forward in time to the late 1970s. The way people use computers has changed significantly, and the way assessment systems are implemented changed accordingly. Note that while the previous section was complete (as far as we could find in published literature), this section is decidedly not so. At this point, the explosion of automated assessment systems/automated grading systems for programming education had already set in. To describe all platforms would take a full dissertation in and of itself. So from now on, we will pick and choose systems that brought new and interesting ideas that stood the test of time.[10]

ACSES, by Nievergelt (1976), was envisioned as a full course for learning computer programming. They even designed it as a full replacement for a course: it was the first system that integrated both instructional texts and exercises. Students following this course would not need personal instruction. In the modern day, this would probably be considered a massive open online course (MOOC).[11]

Another good example of this generation of grading systems is the system by Isaacson & Scott (1989). They describe the functioning of a UNIX shell script that automatically e-mails students if their code did not compile, or if they had incorrect outputs. It also had a configurable output file size limit and time limit. Student programs would be stopped if they exceeded these limits. Like all assessment systems up to this point, they only focus on whether the output of the student's program is correct, and not on the code style.

Reek (1989) takes a different approach. He identifies several issues with gathering students' source files, and then compiling and executing them in the teacher's environment. Students could write destructive code that destroys the teacher's files, or even write a clever program that alters their grades (and covers its tracks while doing so). Note that this is not a new issue: as we discussed before, this was already mentioned as a possibility by Hollingsworth (1960). This was, however, the first system that tried to solve this problem. Avoiding that teachers need to run their students' programs was therefore an explicit goal of his TRY system. Another goal was avoiding giving the inputs that the program was tested on to students. These goals were mostly achieved using the UNIX `setuid` mechanism.

[10] The ideas, not the platforms. As far as we know none of the platforms described in this section are still in use.

[11] Except that it obviously was not an online course; TCP/IP would not be standardized until 1982.

Note that students were using a true multi-user system, as in common use at the time. Every attempt was also recorded in a log file in the teacher's personal directory. Generality of programming language was achieved through intermediate build and test scripts that had to be provided by the teacher.

This is also the first study we could find that pays explicit attention to how expected and generated output is compared. In addition to the basic character-by-character comparison, it is also supported to define the interface for a function that students have to call with their outputs. The instructor can then link an implementation of this function in the build script.

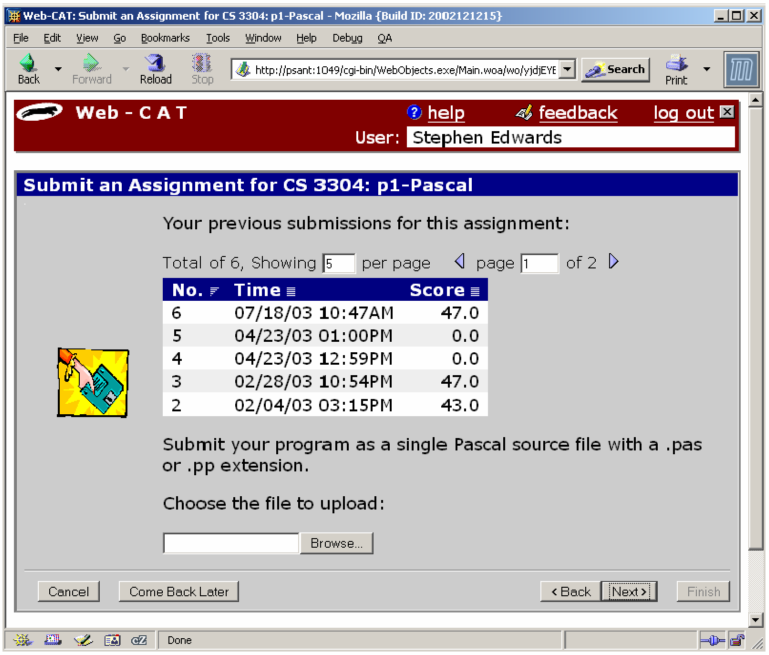
Even later, automated assessment systems were built with graphical user interfaces. A good example of this is ASSYST (Jackson & Usher, 1997). ASSYST also added evaluation on other metrics, such as runtime or cyclomatic complexity as suggested by Hung et al. (1993).

1.1.3 Moving to the web

After Tim Berners-Lee invented the web in 1989 (Berners-Lee et al., 1992), automated assessment systems also started moving to the web. Especially with the rise of Web 2.0 (O'Reilly, 2007) and its increased interactivity, this became more and more common. Systems like the one by Reek (1989) also became impossible to use because of the rise of the personal computer. Mainly because the typical multi-user system was used less and less, but also because the primary way people interacted with a computer was no longer through the command line, but through graphical user interfaces.

Higgins et al. (2003) developed CourseMarker, which is a more general assessment system (not exclusively developed for programming assessment). This was initially not yet a web-based platform, but it did communicate over the network using Java's Remote Method Invocation mechanism. The system it was designed to replace, Ceilidh, did have a basic web submission interface (Hughes et al., 1998). Designing a web client was also mentioned as future work in the paper announcing CourseMarker.

Perhaps the most famous example of the first web-based platforms is Web-CAT (Shah, 2003). In addition to being one of the first web-based automated assessment platforms, it also asked the students to write their own tests. The coverage that these tests achieved was part of the testing done by the platform. Tests are written using standard unit testing frameworks (Edwards & Pérez-Quiñones, 2007). An example of Web-CAT's submission screen can be seen in Figure 1.2.


Web-CAT: Submit an Assignment for CS 3304: p1-Pascal - Mozilla {Build ID: 2002121215}
File Edit View Go Bookmarks Tools Window Help Debug QA
Back Forward Reload Stop
http://psant:1049/cgi-bin/WebObjects.exe/Main.woa/wo/yjdjEYE
Search Print
Web - CAT
help feedback log out
Done

User: Stephen Edwards

Submit an Assignment for CS 3304: p1-Pascal

Your previous submissions for this assignment:

Total of 6, Showing 5 per page page 1 of 2

No.	Time	Score
6	07/18/03 10:47AM	47.0
5	04/23/03 01:00PM	0.0
4	04/23/03 12:59PM	0.0
3	02/28/03 10:54PM	47.0
2	02/04/03 03:15PM	43.0

Submit your program as a single Pascal source file with a .pas or .pp extension.

Choose the file to upload:

Browse...

Cancel Come Back Later < Back Next > Finish

Figure 1.2: Web-CAT's submission screen for students. Image taken from Edwards (2006).

This is also the time when we first start to see mentions of plagiarism and plagiarism detection in the context of automated assessment, presumably because the internet made plagiarizing a lot easier. In one case at MIT over 30% of students were found to be plagiarizing (Wagner, 2000). Daly & Horgan (2005) analysed plagiarizing behaviour by watermarking student submissions, where the watermark consisted of added whitespace at the end of lines. If students carelessly copied another student's submission, they would also copy the whitespace. Around this time, Schleimer et al. (2003) also published MOSS (Measure of Software Similarity), a popular tool for checking code similarity.

Another important platform is the Sphere Online Judge (SPOJ) (Kosowski et al., 2008). SPOJ is especially important in the context of this dissertation, since it was the platform we used before Dodona. SPOJ specifically notes the influence of online contest platforms (and in fact, is a platform that can be used to organize contests). Online contest platforms usually differ from the automated assessment platforms for education in the way they handle feedback. For online contests, the amount of feedback given to participants is often far less than the feedback given in education to students. Although, depending on the educational vision of the teacher, this happens in education as well.

The SPOJ paper also details the security measures they took when executing untrusted code. They use a patched Linux kernel's `rlimits`, the `chroot` mechanism, and traditional user isolation to prevent student code from malicious action.

Another interesting idea was contributed by Brusilovsky & Sosnovsky (2005) in QuizPACK. They combined the idea of parametric exercises with automated assessment by executing source code. In QuizPACK, teachers provide a parameterized piece of code, where the value of a specific variable is the answer that a student needs to give. The piece of code is then evaluated, and the result is compared to the student's answer. Note that in this platform, it is not the students themselves who are writing code.

1.1.4 Adding features

At this point in history, the idea of a web-based automated assessment system for programming education is no longer new. But still, more and more new platforms are being written. For a possible explanation, see Figure 1.3.

All of these platforms support automated assessment of code submitted by students, but try to differentiate themselves through the features they offer. The FPGE platform by Paiva, Queirós, et al. (2022) offers gamification, iWeb-TD (Fonseca et al., 2023) integrates a full-fledged editor, PLearn (Vasyliuk & Lytvyn, 2023) recommends extra exercises to its users, JavAssess (Insa & Silva, 2018) tries to automate grading, and GradeIT (Parihar et al., 2017) features automatic hint generation.

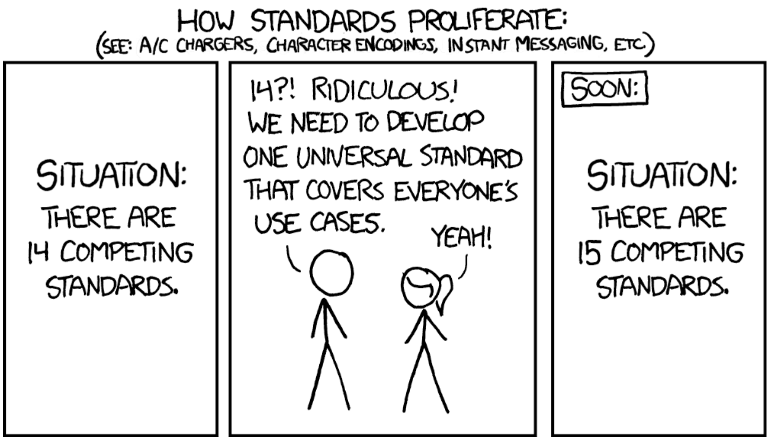

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 1.3: Comic on the proliferation of standards, which is also applicable to the proliferation of automated assessment platforms. Created by Randall Munroe, released under the CC BY-NC 2.5 licence via `https://xkcd.com/927/`.

1.2 Learning analytics and educational data mining

Learning analytics and educational data mining stand at the intersection of computer science, data analytics, and the social sciences, and focus on understanding and improving learning. They are made possible by the increased availability of data about students who are learning, due to the increasing move of education to digital platforms (Romero et al., 2008). They can also serve different actors in the educational landscape: they can help learners directly, help teachers to evaluate their own teaching, allow developers of educational platforms to know what to focus on, allow educational institutions to guide their decisions, and even allow governments to take on data-driven policies (Ferguson, 2012). Learning analytics and educational data mining are overlapping fields, but in general, learning analytics is seen as focusing on the educational challenge, while educational data mining is more focused on the technical challenge (Ferguson, 2012). The analytics focusing on governments or educational institutions is called academic analytics.

In this dissertation, we will use the definitions of educational data mining and learning analytics from Romero & Ventura (2020). They define educational data mining to be “concerned with developing methods for exploring the unique types of data that come from educational environments” and learning analytics as “the measurement, collection, analysis, and reporting of data about learners and their contexts, for purposes of understanding and optimizing learning and the environments in which it occurs”. In short, educational data mining is concerned with developing methods and tools to use educational data, while learning analytics is used to gather and present insights into learning.

Chatti et al. (2012) defined a reference model for learning analytics and educational data mining based on four dimensions: *i)* What data is gathered and used? *ii)* Who is targeted by the analysis? *iii)* Why is the data analysed? *iv)* How is the data analysed? This gives an idea to researchers what to focus on when conceptualizing, executing, and publishing their research.

An example of educational data mining research is Daud et al. (2017), where the students’ background (including family income, family expenditures, gender, martial status, …) is used to predict the student’s learning outcome at the end of the semester. Evaluating this study using the reference model by Chatti et al. (2012), we can see that the data used is very personal and hard to collect. As mentioned in the study, the primary target audience of the study are policymakers. The data is analysed to evaluate the influence of a student’s background on their performance, and this is done by using a number of machine learning techniques (which are compared to one another).

Another example of the research in this field is a study by Akçapınar et al. (2019). They focus on the concept of an early warning system, where student performance can be predicted early and appropriate action could be undertaken. Their study uses data from a blended learning environment, where students can see the lesson’s resources, participate in discussions, and write down their own thoughts about the lessons. Here, the primary target audience is the student. Although the related actions are not yet in scope of the study, the primary goal is to develop such an early warning system. Again, a number of machine learning techniques are compared, to determine which one gives the best results.

1.3 Programming education in Flanders

In Flanders (Belgium), programming is taught in lots of ways, and at many levels. This includes secondary and higher education, but it is also something children can do in their free time, as a hobby. There are also trainings for the workforce, but these are not the focus of this dissertation.

Programming education as a hobby for children is provided by organizations such as CoderDojo[12] and CodeFever[13]. CoderDojo is a non-profit organization that relies on volunteers to organize free sessions for children from 7 up to 18 years old. They use tools like Scratch (Maloney et al., 2010), AppInventor (Patton et al., 2019), and Code.org[14] to teach children the basics of programming in a fun, gamified way. CodeFever is also a non-profit organization, but does ask for registration fees for enrolling in one of their courses. They focus on children aged 8 to 15, and primarily use Scratch and JavaScript to teach programming concepts.

In secondary education, things recently changed. Before 2021, education related to computing was very much up to the individual school and teacher. While there were some schools where programming was taught, this was mostly a rare occurrence except for a few specific IT-oriented programmes. In 2021, however, the Flemish parliament approved a new set of educational goals. In these educational goals, there was an explicit focus on digital competence, where for a lot of educational programmes, this explicitly included programming. Not much later though, one of the umbrella organizations for schools challenged the new educational goals in Belgium's constitutional court. They felt that the government was overreaching in the specificity of the educational goals.[15] The constitutional court agreed, after which the government went back to the drawing board, and made a lot of the goals less detailed. Digital competence is still a part of the new educational goals, but programming is no longer explicitly listed. However, for some programmes there are specific educational goals that list competences related to computer science that students should have when finishing secondary education. These programmes are mostly focused on the sciences or have more mathematics. The listed competences include programming, algorithms, data structures, numerical methods, etc. For programmes focused on IT, there is an even bigger list of related

[12]`https://coderdojobelgium.be/nl` (in Dutch)
[13]`https://www.codefever.be/nl` (in Dutch)
[14]`https://code.org/`
[15]Traditionally, the educational goals were quite loose, allowing the umbrella organizations to add their own accents to the subjects being taught.

competences that the students should have. Python is the most common programming language used at this level, but other programming languages like Java and C# are also used.

In higher education, programming has made its way into a lot of programmes. Almost all students studying exact sciences or engineering have at least one programming course, but programming is also taught outside these domains (e.g. as part of a statistics course). Here we see the greatest diversity in the programming languages that are taught. Python, Java, and R are the most common languages for students for whom computer science is not the main subject. Computer science students are taught a plethora of languages, from Python and Java to Prolog, Haskell and Scheme.

1.4 Structure of this dissertation

This dissertation tries to answer the following central research question: How can we use data from an automated assessment platform to improve learning and teaching in programming education? An important prerequisite for answering this question is the existence of an automated assessment platform. For this dissertation we use Dodona[16] as that automated assessment platform. Dodona is an online learning environment that recognizes the importance of active learning and just-in-time feedback in courses involving programming assignments. We started Dodona because our own educational needs outgrew SPOJ (Kosowski et al., 2008), the platform we were using before. SPOJ was chosen because it was one of the rare platforms that allowed the addition of courses, exercises (and even judges) by teachers. This also influenced the development of Dodona. Every year since its inception in 2016, more and more teachers have started using Dodona. It is now used in most higher education institutions in Flanders, and many secondary education institutions as well.

The development and use of Dodona is an important part of the work that went into this dissertation, and therefore constitutes the first part of this dissertation. Chapter 2 answers the following question: What features does a platform like Dodona need? We therefore give an overview of the user-facing features of Dodona, from user management to how feedback is represented. Chapter 3 answers the question: How is Dodona used in practice? We do this by presenting some facts and figures of its use,

[16]`https://dodona.be/`

students' opinions of the platform, and an extensive case study on how Dodona's features are used to optimize teaching. This case study also provides insight into the educational context for the research described in Chapters 5 and 6. Chapter 4 answers the question: What goes into building a platform like Dodona? We therefore focus on the technical aspect of developing Dodona and its related ecosystem of software. This includes a discussion of the technical challenges related to developing a platform like Dodona, and how the Dodona team adheres to modern standards of software development.

In the second part of this dissertation, we focus on the educational data mining studies we performed to improve learning and teaching. Chapter 5 asks whether we can predict student performance and whether we can do so in a way that makes it clear which factors influence this prediction. The chapter discusses an educational data mining study where we tried to predict whether students would pass or fail a programming course at the end of the semester based solely on their submission history in Dodona. It also briefly details a study we collaborated on with researchers from Jyväskylä University in Finland, where we replicated our study in their own educational context, with data from their own educational platform.

Chapter 6 then looks at the teacher's side of our central question and answers the question on how we can optimize the process of giving manual feedback. We first give an overview of how Dodona changed manual assessment in our own educational context and then finish the chapter with some recent work on a machine learning method we developed to predict what feedback teachers will give when manually assessing student submissions.

Finally, Chapter 7 concludes the dissertation with some discussion on Dodona's opportunities and challenges for the future.

1.5 On ethics and research integrity

Ethics and integrity have been very important during the development of Dodona. Early in its development, we met with the Data Protection Officer of Ghent University to create a privacy policy. We also only keep the data required for running the platform. This results in very little personal information being stored; only the users' names, usernames, and email addresses are stored in their profile. The only other data stored is data generated in the platform: submissions, evaluations, questions, answers, etc. In this case too, we only keep the information required for the correct functioning of these features. The development of Dodona is also done in the open: the platform has been open-source since August 2019.

The same philosophy has been extended to our research. All data used in Chapter 5 was pseudonymized before the analysis was started and no data was collected specifically to enable this research. Conversely, the research was restricted to data that was already collected by Dodona for its regular operations. The data used in the study was also not published. This is of course not conducive to the verifiability of the research, which is why we were very happy to see that our method could be reproduced in another context. The research presented in Chapter 6 also doesn't rely on any personal information: only the IDs and locations of the saved feedback items were used, in addition to the relevant code. Here also, the underlying data was not published, to avoid any inadvertent personal information present in the code being published.

2 A closer look at Dodona

In this chapter, we will give an overview of Dodona's most important features. This chapter answers the question what features a platform like Dodona needs. The most important feature is automated assessment, but as we show in this chapter, a lot more features than that are needed.

This chapter is partially based on **Van Petegem, C.**, Maertens, R., Strijbol, N., Van Renterghem, J., Van der Jeugt, F., De Wever, B., Dawyndt, P., Mesuere, B., 2023. Dodona: Learn to code with a virtual co-teacher that supports active learning. *SoftwareX* 24, 101578. The work described in this chapter was performed by the whole Dodona team. It is difficult to pinpoint who did what. The code and its history can be looked at[17], but it will never give a full view of the true collaborative effort of Dodona.

2.1 User management

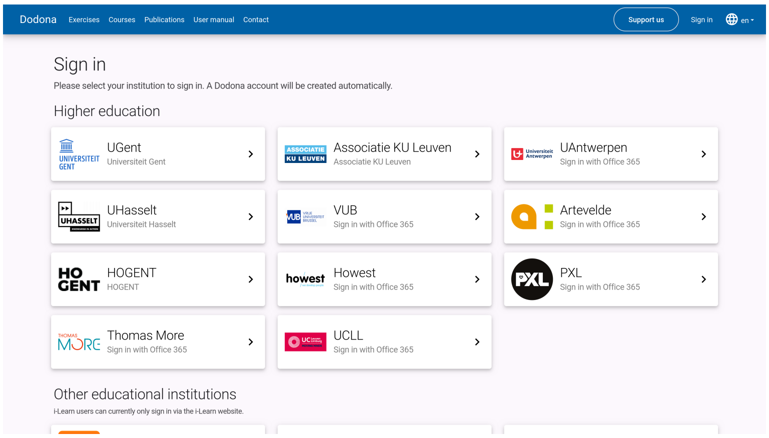
Establishing the identity of its users is very important for an educational platform. For this reason, instead of providing its own authentication and authorization, Dodona delegates authentication to external identity providers (e.g. educational and research institutions) through SAML (Farrell et al., 2002), OAuth (Hardt, 2012; Leiba, 2012) and OpenID Connect (Sakimura et al., 2014). The configured OAuth providers are Microsoft, Google, and Smartschool. This support for **decentralized authentication** allows users to benefit from single sign-on when using their institutional account across multiple platforms and teachers to trust their students' identities when taking high-stakes tests and exams in Dodona.

Dodona automatically creates user accounts upon successful authentication and uses the association with external identity providers to assign an institution to users. These institutions can have multiple sign-in methods.

[17]`https://github.com/dodona-edu/dodona/commits/main/`

If a user uses more than one of those methods, these logins are linked to the same user. Institutions within Dodona can be used by teachers to establish filters about who is allowed to register for their courses, establishing an extra level of trust that their students have correctly signed in. Institutions are also categorized internally in secundary education, higher education, and other (e.g. the Flemish government).

By default, newly created users are assigned a student role. Teachers and instructors who wish to create content (courses, learning activities and judges), must first request teacher rights using a streamlined form[18]. The sign in page can be seen in Figure 2.1. After logging in, a user sees an overview of the courses they are registered with.



Sign in

Please select your institution to sign in. A Dodona account will be created automatically.

Higher education

UGent Universiteit Gent	Associatie KU Leuven Associatie KU Leuven	UAntwerpen Sign in with Office 365
UHasselt Universiteit Hasselt	VUB Sign in with Office 365	Artevelde Sign in with Office 365
HOGENT HOGENT	Howest Sign in with Office 365	PXL Sign in with Office 365
Thomas More Sign in with Office 365	UCLL Sign in with Office 365	

Other educational institutions

i-Learn users can currently only sign in via the i-Learn website.

Figure 2.1: Sign in page showing the different options for users to sign in.

2.2 Course management

In Dodona, a **course** is where teachers and instructors effectively manage a learning environment by instructing, monitoring and evaluating their students and interacting with them, either individually or as a group. A Dodona user who created a course becomes its first administrator and can promote other registered users as **course administrators**. In what follows, we will also use the generic term teacher as a synonym for course

[18]`https://dodona.be/rights_requests/new/`

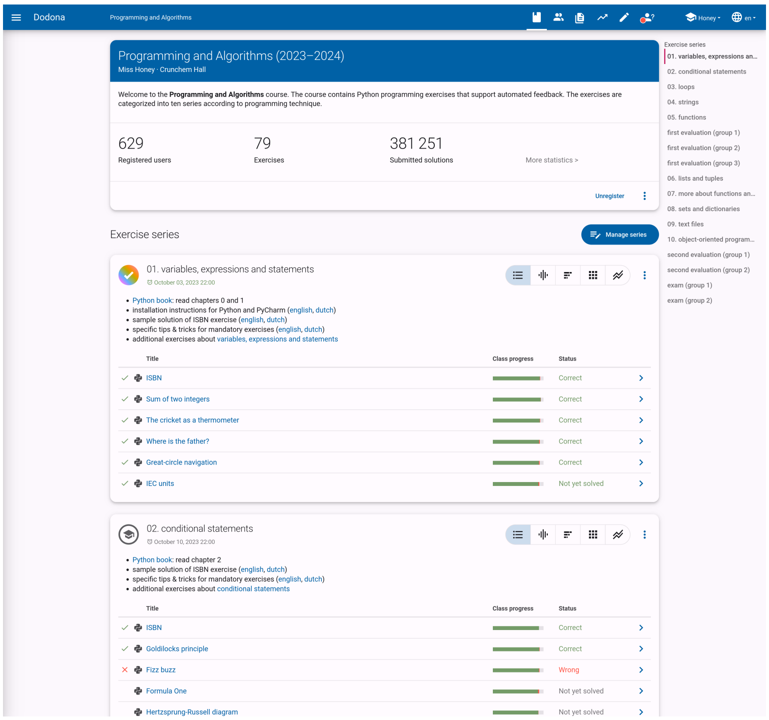
administrators if this Dodona-specific interpretation is clear from the context, but keep in mind that courses may have multiple administrators.

The course itself is laid out as a **learning path** that consists of course units called **series**, each containing a sequence of **learning activities** (Figure 2.2). Among the learning activities we differentiate between **reading activities** that can be marked as read and **programming assignments** with support for automated assessment of submitted solutions. Learning paths are composed as a recommended sequence of learning activities to build knowledge and skills progressively, allowing students to monitor their own progress at any point in time. Courses can either be created from scratch or from copying an existing course and making additions, deletions and rearrangements to its learning path.

Students can **self-register** to courses in order to avoid unnecessary user management. A course can either be announced in the public overview of Dodona for everyone to see, or be limited in visibility to students from a certain educational institution. Alternatively, students can be invited to a hidden course by sharing a secret link. Independent of course visibility, registration for a course can either be open to everyone, restricted to users from the institution the course is associated with, or new registrations can be disabled altogether. Registrations are either approved automatically or require explicit approval by a teacher. Registered users can be tagged with one or more labels to create subgroups that may play a role in learning analytics and reporting.

Students and teachers more or less see the same course page, except for some management features and learning analytics that are reserved for teachers. Teachers can make content in the learning path temporarily inaccessible and/or invisible to students. Content is typically made inaccessible when it is still in preparation or if it will be used for evaluating students during a specific period. A token link can be used to grant access to invisible content, e.g. when taking a test or exam from a subgroup of students.

Students can only mark reading activities as read once, but there is no restriction on the number of solutions they can submit for programming assignments. Submitted solutions are automatically assessed and students receive immediate feedback as soon as the assessment has completed, usually within a few seconds. Dodona stores all submissions, along with submission metadata and generated feedback, such that the submission and feedback history can be reclaimed at all times. On top of automated



Dodona Programming and Algorithms Honey en

Programming and Algorithms (2023–2024)
Miss Honey · Crunchem Hall

Welcome to the **Programming and Algorithms** course. The course contains Python programming exercises that support automated feedback. The exercises are categorized into ten series according to programming technique.

629 Registered users | 79 Exercises | 381 251 Submitted solutions | More statistics >

Unregister

Exercise series

Manage series

01. variables, expressions and statements
October 03, 2023 22:00

- Python book: read chapters 0 and 1
- installation instructions for Python and PyCharm (english, dutch)
- sample solution of ISBN exercise (english, dutch)
- specific tips & tricks for mandatory exercises (english, dutch)
- additional exercises about variables, expressions and statements

	Title	Class progress	Status
✓	ISBN		Correct
✓	Sum of two integers		Correct
✓	The cricket as a thermometer		Correct
✓	Where is the father?		Correct
✓	Great-circle navigation		Correct
✓	IEC units		Not yet solved

02. conditional statements
October 10, 2023 22:00

- Python book: read chapter 2
- sample solution of ISBN exercise (english, dutch)
- specific tips & tricks for mandatory exercises (english, dutch)
- additional exercises about conditional statements

	Title	Class progress	Status
✓	ISBN		Correct
✓	Goldilocks principle		Correct
✗	Fizz buzz		Wrong
	Formula One		Not yet solved
	Hertzsprung-Russell diagram		Not yet solved

Exercise series
- 01. variables, expressions an...
- 02. conditional statements
- 03. loops
- 04. strings
- 05. functions
- first evaluation (group 1)
- first evaluation (group 2)
- first evaluation (group 3)
- 06. lists and tuples
- 07. more about functions an...
- 08. sets and dictionaries
- 09. text files
- 10. object-oriented program...
- second evaluation (group 1)
- second evaluation (group 2)
- exam (group 1)
- exam (group 2)

Figure 2.2: Main course page (administrator view) showing some series with deadlines, reading activities and programming assignments in its learning path. At any point in time, students can see their own progress through the learning path of the course. Teachers have some additional icons in the navigation bar (top) that lead to an overview of all students and their progress, an overview of all submissions for programming assignments, general learning analytics about the course, course management and a dashboard with questions from students in various stages from being answered (Figure 2.9). The red dot on the latter icon notifies that some student questions are still pending.

assessment, student submissions may be further assessed and graded manually by a teacher (see Chapter 6).

Series can have a **deadline**. Passed deadlines do not prevent students from marking reading activities or submitting solutions for programming assignments in their series. However, learning analytics, reports and exports usually only take into account submissions before the deadline. Because of the importance of deadlines and to avoid discussions with students about missed deadlines, deadlines are not only announced on the course page. The student's home page highlights upcoming deadlines for individual courses and across all courses. While working on a programming assignment, students will also see a clear warning starting from ten minutes before a deadline. Courses also provide an iCalendar link (Stenerson & Dawson, 1998) that students can use to publish course deadlines in their personal calendar application.

Because Dodona logs all student submissions and their metadata, including feedback and grades from automated and manual assessment, we use that data to integrate reports and learning analytics in the course page (Ferguson, 2012). This includes heatmaps (Figure 2.3) and punch cards (Figure 2.4) of user activity, graphs showing class progress (Figure 2.5), and so on.

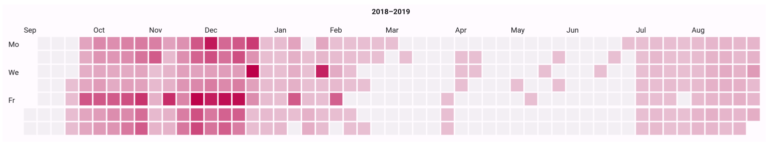


Figure 2.3: Heatmap showing on which days in the semester students are more active or less active.

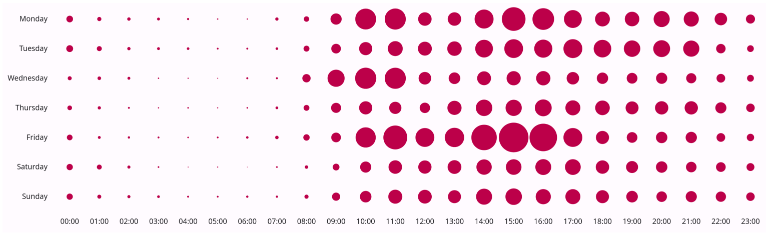


Figure 2.4: Punchcard showing when during the week students are working on their exercises.

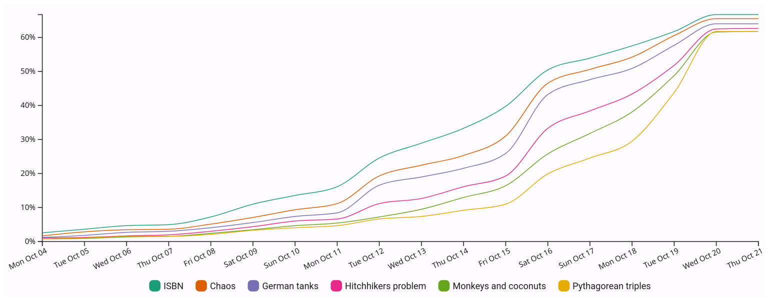


Figure 2.5: Graph showing the percentage of students that correctly solved the exercises in a certain series over time.

We also provide export wizards that enable the extraction of raw and aggregated data in CSV format for downstream processing and educational data mining (Baker & Yacef, 2009; Romero & Ventura, 2010). This allows teachers to better understand student behaviour, progress and knowledge, and might give deeper insight into the underlying factors that contribute to student actions (Ihantola et al., 2010). Understanding, knowledge and insights that can be used to make informed decisions about courses and their pedagogy, increase student engagement, and identify at-risk students (see Chapter 5).

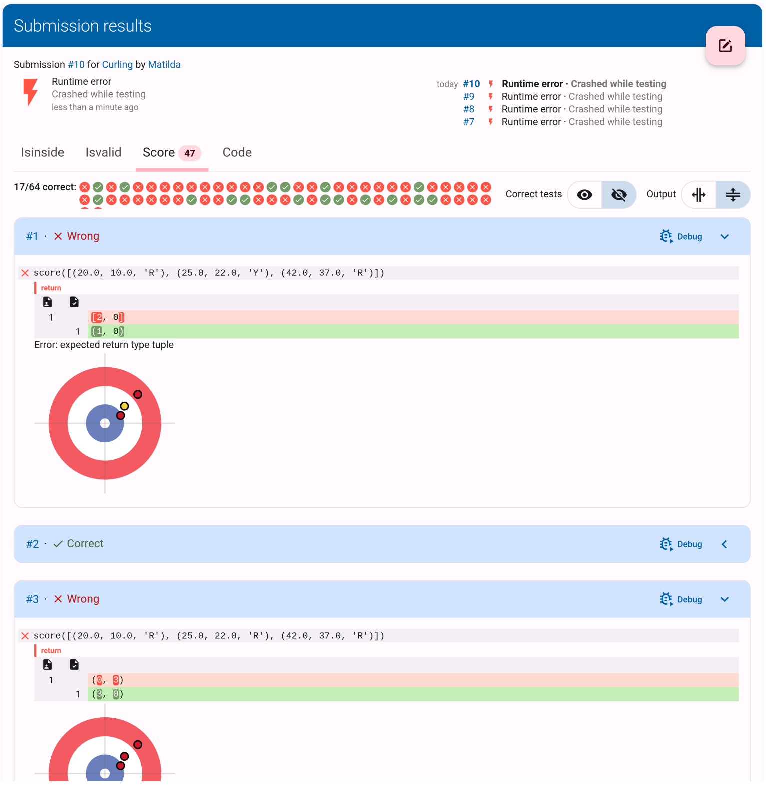
2.3 Exercises

There are two types of assignments in Dodona: reading activities and programming exercises. While reading activities only consist of descriptions, programming exercises need an additional **assessment configuration** that sets a programming language and a judge (for more information on judges, see Section 2.4). This configuration is used to perform the automated assessment. The configuration may also set a Docker image, a time limit, a memory limit and grant Internet access to the container that is instantiated from the image, but these settings have proper default values. The configuration might also provide additional **assessment resources**: files made accessible to the judge during assessment. The specification of how these resources must be structured and how they are used during assessment is completely up to the judge developers. Finally, the configuration might also contain **boilerplate code**: a skeleton students can use to start the implementation that is provided in the code editor along with the description. Directories that contain a learning activity also have their own internal directory structure that includes a **description** in HTML or Markdown. Descriptions may reference data files and multimedia content included in the repository, and such content can be shared across all learning activities in the repository. Embedded images are automatically encapsulated in a responsive lightbox to improve readability. Mathematical formulas in descriptions are supported through MathJax (Cervone, 2012).

Where automatic assessment and feedback generation is outsourced to the judge linked to an assignment, Dodona itself takes up the responsibility for rendering the feedback. This frees judge developers from putting effort in feedback rendering and gives a coherent look-and-feel even for students that solve programming assignments assessed by different judges. Because the way feedback is presented is very important (Mani et al., 2014), we took great care in designing how feedback is displayed to make its interpretation as easy as possible (Figure 2.6). Differences between generated and expected output are automatically highlighted for each failed test (Myers, 1986), and users can swap between displaying the output lines side-by-side or interleaved to make differences more comparable. We even provide specific support for highlighting differences between tabular data such as CSV files, database tables and data frames. Users have the option to dynamically hide contexts whose test cases all succeeded, allowing them to immediately pinpoint reported mistakes in feedback that contains lots of succeeded test cases. To ease debugging the source code of submissions for Python assignments, the Python Tutor (Guo, 2013) can be launched directly from any context with a combination of the submitted source code and the test code from the context. Students typically report this as one of the most useful features of Dodona.

2.4 Judges

The range of approaches, techniques and tools for software testing that may underpin assessing the quality of software under test is incredibly diverse. Static testing directly analyses the syntax, structure and data flow of source code, whereas dynamic testing involves running the code with a given set of test cases (Graham et al., 2021; Oberkampf & Roy, 2010). Black-box testing uses test cases that examine functionality exposed to end-users without looking at the actual source code, whereas white-box testing hooks test cases onto the internal structure of the code to test



Submission results

Submission #10 for Curling by Matilda

Runtime error
Crashed while testing
less than a minute ago

today #10 Runtime error · Crashed while testing
#9 Runtime error · Crashed while testing
#8 Runtime error · Crashed while testing
#7 Runtime error · Crashed while testing

Isinside Isvalid Score 47 Code

17/64 correct:

Correct tests Output

#1 · ✕ Wrong Debug

```
score([(20.0, 10.0, 'R'), (25.0, 22.0, 'Y'), (42.0, 37.0, 'R')])
return
1       [2, 0]
    1   (1, 0)
```

Error: expected return type tuple

#2 · ✓ Correct Debug

#3 · ✕ Wrong Debug

```
score([(20.0, 10.0, 'R'), (25.0, 22.0, 'R'), (42.0, 37.0, 'R')])
return
1       (0, 3)
    1   (3, 0)
```

Figure 2.6: Dodona rendering of feedback generated for a submission of the Python programming assignment "Curling". The feedback is split across three tabs: `isinside`, `isvalid` and `score`. 48 tests under the `score` tab failed as can be seen from the badge in the tab header. The "Code" tab displays the source code of the submission with annotations added during automatic and/or manual assessment (Figure 2.11). The differences between the generated and expected return values were automatically highlighted and the judge used HTML snippets to add a graphical representation (SVG) of the problem for the failed test cases. In addition to highlighting differences between the generated and expected return values of the first (failed) test case, the judge also added a text snippet that points the user to a type error.

specific paths within a single unit, between units during integration, or between subsystems (Nidhra & Dondeti, 2012). So, broadly speaking, there are three levels of white-box testing: unit testing, integration testing and system testing (Dooley, 2011; Wiegers, 1996). Source code submitted by students can therefore be verified and validated against a multitude of criteria: functional completeness and correctness, architectural design, usability, performance and scalability in terms of speed, concurrency and memory footprint, security, readability (programming style), maintainability (test quality) and reliability (Staubitz et al., 2015). This is also reflected by the fact that a diverse range of metrics for measuring software quality have come forward, such as cohesion/coupling (Stevens et al., 1999; Yourdon & Constantine, 1979), cyclomatic complexity (McCabe, 1976) or test coverage (Miller & Maloney, 1963).

To cope with such a diversity in software testing alternatives, Dodona is centred around a generic infrastructure for **programming assignments that support automated assessment**. Assessment of a student submission for an assignment comprises three loosely coupled components: containers, judges and assignment-specific assessment configurations. Judges have a default Docker image that is used if the configuration of a programming assignment does not specify one explicitly. Dodona builds the available images from Dockerfiles specified in a separate git repository. More information on this underlying mechanism can be found in Chapter 4. An overview of the existing judges and the corresponding number of exercises and submissions in Dodona can be found in Table 2.1.

2.5 Repositories

Where courses are created and managed in Dodona itself, other content is managed in external git **repositories** (Figure 2.7). In this distributed content management model, a repository either contains a single judge or a collection of learning activities. Setting up a **webhook** for the repository guarantees that any changes pushed to its default branch are automatically and immediately synchronized with Dodona. This even works without the need to make repositories public, as they may contain information that should not be disclosed such as programming assignments that are under construction, contain model solutions, or will be used during tests or exams. Instead, a **Dodona service account** must be granted push and pull access to the repository. Some settings of a learning activity can be modified through the web interface of Dodona, but any changes are always

Judge	# exercises	# submissions
Bash	289	675 902
C	77	31 822
C#	256	44 294
Compilers	3	38
HTML	187	24 947
Haskell	76	76 556
Java 8	93	90 084
Java 21	450	730 383
JavaScript	36	68
Markdown	14	354
Prolog	54	37 609
Python	8 481	13 798 051
R	1 293	958 069
SQL	298	114 725
Scheme	277	125 138
TESTed	1 139	333 507
Turtle	17	446

Table 2.1: Overview of the judges in Dodona, together with the corresponding number of exercises and submissions in Dodona. The data was gathered in March 2024. The TESTed judge is a special case in that it supports multiple programming languages. More information on it can be found in Section 4.4. The number of exercises and submissions for the JavaScript judge is undercounted: most of its exercises were converted to TESTed exercises, which also moved the submissions to those exercises to TESTed.

pushed back to the repository in which the learning activity is configured so that it always remains the master copy.

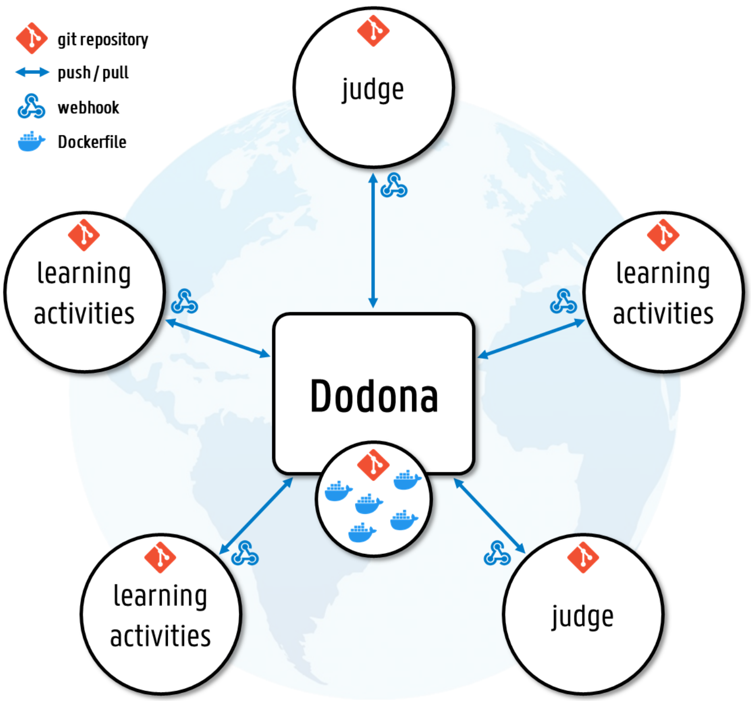


Figure 2.7: Distributed content management model that allows to seamlessly integrate custom learning activities (reading activities and programming assignments with support for automated assessment) and judges (frameworks for automated assessment) into Dodona. Content creators manage their content in external git repositories, keep ownership over their content, control who can co-create, and set up webhooks to automatically synchronize any changes with the content as published on Dodona.

Due to the distributed nature of content management, creators also keep ownership over their content and control who may co-create. After all, access to a repository is completely independent of access to its learning activities that are published in Dodona. The latter is part of the configuration of learning activities, with the option to either share learning activities so that all teachers can include them in their courses or to restrict inclusion of learning activities to courses that are explicitly granted access. Dodona automatically stores metadata about all learning activities such as content type, natural language, programming language and repository to increase their findability in our large collection. Learning activities may also be tagged with additional labels as part of their configuration. Any repository containing learning activities must have a predefined directory structure[19].

2.6 Internationalization and localization

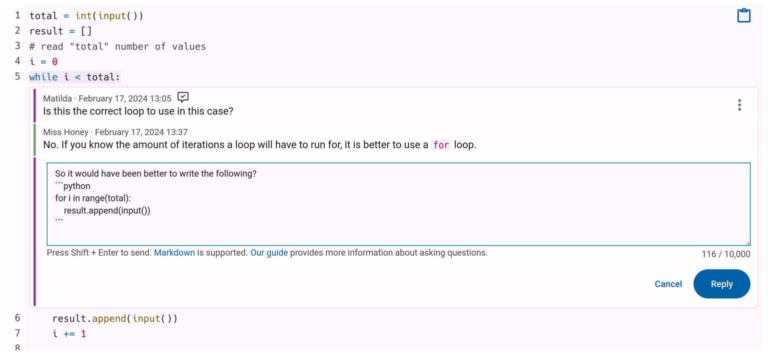
Internationalization (i18n) is a shared responsibility between Dodona, learning activities and judges. All boilerplate text in the user interface that comes from Dodona itself is supported in English and Dutch, and users can select their preferred language. Content creators can specify descriptions of learning activities in both languages, and Dodona will render a learning activity in the user's preferred language if available. When users submit solutions for a programming assignment, their preferred language is passed as submission metadata to the judge. It's then up to the judge to take this information into account while generating feedback.

Dodona always displays **localized deadlines** based on a time zone setting in the user profile, and users are warned when the current time zone detected by their browser differs from the one in their profile.

2.7 Questions, answers and code reviews

A downside of using discussion forums in programming courses is that students can ask questions about programming assignments that are either disconnected from their current implementation or contain code snippets that may give away (part of) the solution to other students (Nandi et al., 2012). Dodona therefore allows students to address teachers with questions they directly attach to their submitted source code. We support both general questions and questions linked to specific lines of their submission (Figure 2.8). Questions are written in Markdown (e.g. to include markup, tables, syntax highlighted code snippets or multimedia), with support for MathJax (e.g. to include mathematical formulas).

[19]`https://docs.dodona.be/en/references/exercise-directory-structure/`



```python
total = int(input())
result = []
# read "total" number of values
i = 0
while i < total:
```

Matilda · February 17, 2024 13:05
Is this the correct loop to use in this case?

Miss Honey · February 17, 2024 13:37
No. If you know the amount of iterations a loop will have to run for, it is better to use a `for` loop.

So it would have been better to write the following?
```python
for i in range(total):
    result.append(input())
```

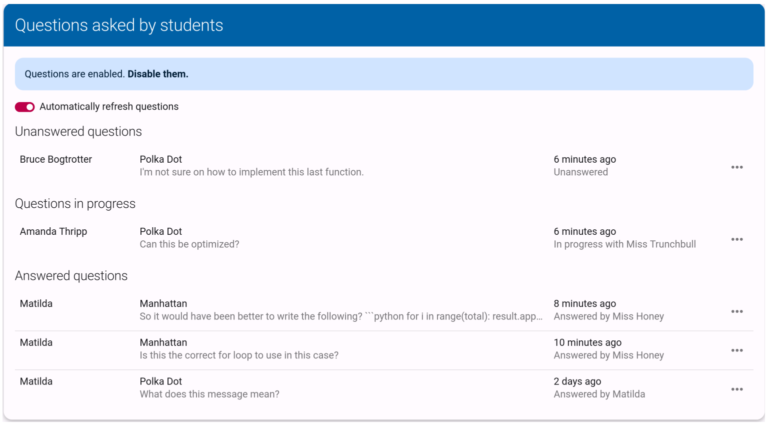
Press Shift + Enter to send. Markdown is supported. Our guide provides more information about asking questions.
116 / 10,000

```python
    result.append(input())
    i += 1
```

Figure 2.8: A student (Matilda) previously asked a question that has already been answered by her teacher (Miss Honey). Based on this response, the student is now asking a follow-up question that can be formatted using Markdown.

Teachers are notified whenever there are pending questions (Figure 2.2). They can process these questions from a dedicated dashboard with live updates (Figure 2.9). The dashboard immediately guides them from an incoming question to the location in the source code of the submission it relates to, where they can answer the question similar to how students ask questions. To avoid questions being inadvertently handled simultaneously by multiple teachers, they have a three-state lifecycle: pending, in progress and answered. In addition to teachers changing question states while answering them, students can also mark their own questions as being answered. The latter might reflect the rubber duck debugging (Hunt, 1999) effect that is triggered when students are forced to explain a problem to someone else while asking questions in Dodona. Teachers can (temporarily) disable the option for students to ask questions in a course, e.g. when a course is over or during hands-on sessions or exams when students are expected to ask questions face-to-face rather than online.

Manual source code annotations from students (questions) and teachers (answers) are rendered in the same way as source code annotations resulting from automated assessment. They are mixed in the source code displayed in the "Code" tab, showing their complementary nature. It is not required that students take the initiative for the conversation. Teachers can also start adding source code annotations while reviewing a submission. Such **code reviews** will be used as a building block for manual assessment.


Questions asked by students

Questions are enabled. **Disable them.**

Automatically refresh questions

Unanswered questions

Bruce Bogtrotter	Polka Dot I'm not sure on how to implement this last function.	6 minutes ago Unanswered	•••

Questions in progress

Amanda Thripp	Polka Dot Can this be optimized?	6 minutes ago In progress with Miss Trunchbull	•••

Answered questions

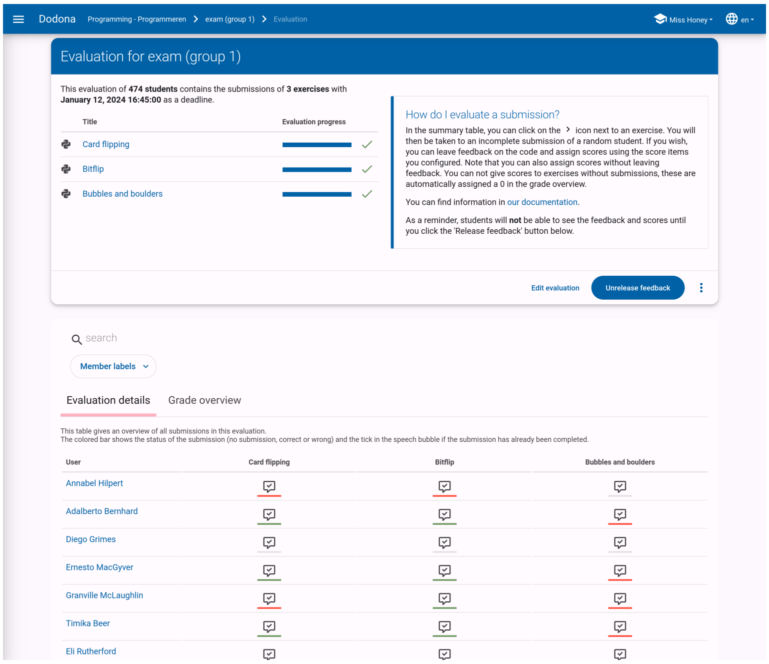
Matilda	Manhattan So it would have been better to write the following? ```python for i in range(total): result.app...	8 minutes ago Answered by Miss Honey	•••
Matilda	Manhattan Is this the correct for loop to use in this case?	10 minutes ago Answered by Miss Honey	•••
Matilda	Polka Dot What does this message mean?	2 days ago Answered by Matilda	•••

Figure 2.9: Live updated dashboard showing all incoming questions in a course while asking questions is enabled. Questions are grouped into three categories: unanswered, in progress and answered.

2.8 Manual assessment

Teachers can create an **evaluation** for a series to manually assess student submissions for its programming assignments after a specific period, typically following the deadline of some homework, an intermediate test or a final exam. An example of an evaluation overview can be seen on Figure 2.10. The evaluation embodies all programming assignments in the series and a group of students that submitted solutions for these assignments. Because a student may have submitted multiple solutions for the same assignment, the last submission before a given deadline is automatically selected for each student and each assignment in the evaluation. This automatic selection can be manually overruled afterwards. The evaluation deadline defaults to the deadline set for the associated series, if any, but an alternative deadline can be selected as well.

Evaluations support **two-way navigation** through all selected submissions: per assignment and per student. For evaluations with multiple assignments, it is generally recommended to assess per assignment and not per student, as students can build a reputation throughout an assessment (Malouff & Thorsteinsson, 2016). As a result, they might be rated more favourably with a moderate solution if they had excellent solutions for assignments that were assessed previously, and vice versa (Malouff et al., 2013). Assessment per assignment breaks this reputation as it interferes



Dodona Programming - Programmeren > exam (group 1) > Evaluation   Miss Honey   en

Evaluation for exam (group 1)

This evaluation of **474 students** contains the submissions of **3 exercises** with **January 12, 2024 16:45:00** as a deadline.

Title	Evaluation progress
Card flipping	✓
Bitflip	✓
Bubbles and boulders	✓

How do I evaluate a submission?

In the summary table, you can click on the > icon next to an exercise. You will then be taken to an incomplete submission of a random student. If you wish, you can leave feedback on the code and assign scores using the score items you configured. Note that you can also assign scores without leaving feedback. You can not give scores to exercises without submissions, these are automatically assigned a 0 in the grade overview.

You can find information in our documentation.

As a reminder, students will **not** be able to see the feedback and scores until you click the 'Release feedback' button below.

Edit evaluation   Unrelease feedback

search

Member labels

Evaluation details   Grade overview

This table gives an overview of all submissions in this evaluation.
The colored bar shows the status of the submission (no submission, correct or wrong) and the tick in the speech bubble if the submission has already been completed.

User	Card flipping	Bitflip	Bubbles and boulders
Annabel Hilpert	✓ 🟥	✓ 🟥	✓ ⬜
Adalberto Bernhard	✓ 🟩	✓ 🟩	✓ 🟥
Diego Grimes	✓ ⬜	✓ ⬜	✓ ⬜
Ernesto MacGyver	✓ 🟩	✓ 🟩	✓ 🟥
Granville McLaughlin	✓ 🟥	✓ 🟩	✓ 🟥
Timika Beer	✓ 🟩	✓ 🟩	✓ 🟥
Eli Rutherford	✓	✓	✓

Figure 2.10: Pseudonymized overview of an evaluation in Dodona. For each student, both the correctness of their submission and whether it has been graded is shown.

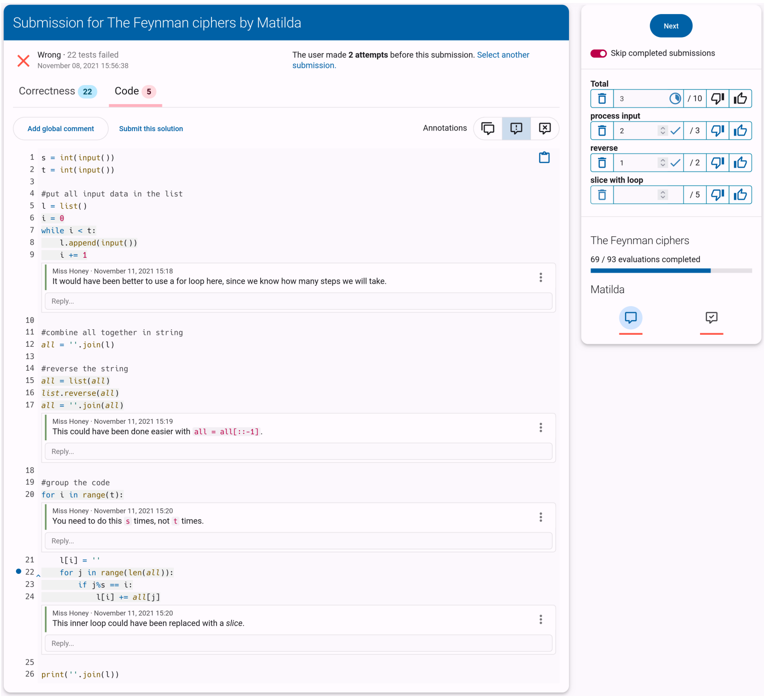
less with the quality of previously assessed assignments from the same student. Possible bias from the same sequence effect is reduced during assessment per assignment as students are visited in random order for each assignment in the evaluation. In addition, **anonymous mode** can be activated as a measure to eliminate the actual or perceived halo effect conveyed through seeing a student's name during assessment (Lebuda & Karwowski, 2013). While anonymous mode is active, all students are automatically pseudonymized. Anonymous mode is not restricted to the context of assessment and can be used across Dodona, for example while giving in-class demos.

When reviewing a selected submission from a student, assessors have direct access to the feedback that was previously generated during automated assessment: source code annotations in the "Code" tab and other structured and unstructured feedback in the remaining tabs. Moreover, next to the feedback that was made available to the student, the specification of the assignment may also add feedback generated by the judge that is only visible to the assessor. Assessors might then complement the assessment made by the judge by adding **source code annotations** as formative feedback and by **grading** the evaluative criteria in a scoring rubric as summative feedback (Figure 2.11).

Previous annotations can be reused to speed up the code review process, because remarks or suggestions tend to recur frequently when reviewing submissions for the same assignment. Grading requires setting up a specific **scoring rubric** for each assignment in the evaluation, as a guidance for evaluating the quality of submissions (Dawson, 2017; Popham, 1997). The evaluation tracks which submissions have been manually assessed, so that analytics about the assessment progress can be displayed and to allow multiple assessors working simultaneously on the same evaluation, for example one (part of a) programming assignment each.

2.9 Conclusion

As we have shown in this chapter, a platform like Dodona needs a lot more features than just automated assessment and feedback. Features like course management and user management allow teachers to manage their students, while infrastructure around exercises such as repositories and our judges are required to allow them to easily add exercises. Additional features like Q&A, code reviews, and evaluations make sure that teachers



Submission for The Feynman ciphers by Matilda

Wrong · 22 tests failed
November 08, 2021 15:56:38

The user made **2 attempts** before this submission. Select another submission.

Correctness 22 | Code 5

Add global comment | Submit this solution | Annotations

```
s = int(input())
t = int(input())

#put all input data in the list
l = list()
i = 0
while i < t:
    l.append(input())
    i += 1
```

Miss Honey · November 11, 2021 15:18
It would have been better to use a for loop here, since we know how many steps we will take.

Reply...

```

#combine all together in string
all = ''.join(l)

#reverse the string
all = list(all)
list.reverse(all)
all = ''.join(all)
```

Miss Honey · November 11, 2021 15:19
This could have been done easier with `all = all[::-1]`.

Reply...

```

#group the code
for i in range(t):
```

Miss Honey · November 11, 2021 15:20
You need to do this `s` times, not `t` times.

Reply...

```
    l[i] = ''
    for j in range(len(all)):
        if j%s == i:
            l[i] += all[j]
```

Miss Honey · November 11, 2021 15:20
This inner loop could have been replaced with a *slice*.

Reply...

```

print(''.join(l))
```

Next

Skip completed submissions

Total	3 / 10
process input	2 / 3
reverse	1 / 2
slice with loop	/ 5

The Feynman ciphers

69 / 93 evaluations completed

Matilda

Figure 2.11: Manual assessment of a submission: a teacher (Miss Honey) is giving feedback on the source code by adding inline annotations and is grading the submission by filling up the scoring rubric that was set up for the programming assignment “The Feynman ciphers”.

can interact with their students, while having the context they are talking about near their interactions. Creating a platform like Dodona entails a lot of work to get these things right.

3 Dodona in educational practice

This chapter answers the question how Dodona is used and shows how it creates an active learning environment. We start by mentioning some facts and figures, and discussing a user study we performed. We then explain how Dodona can be used on the basis of a case study. This case study also provides insight into the educational context for the research described in Chapters 5 and 6.

This chapter is partially based on **Van Petegem, C.**, Maertens, R., Strijbol, N., Van Renterghem, J., Van der Jeugt, F., De Wever, B., Dawyndt, P., Mesuere, B., 2023. Dodona: Learn to code with a virtual co-teacher that supports active learning. *SoftwareX* 24, 101578. The course described in this chapter was mostly developed by prof. Peter Dawyndt, but has also seen numerous contributions by teaching assistents.

3.1 Facts and figures

Dodona's design decisions have allowed it to spread to more than 1 800 schools and higher education institutions, mainly in Flanders (Belgium) and the Netherlands. The renewed interest in embedding computational thinking in formal education has undoubtedly been an important stimulus for such a wide uptake (Wing, 2006). All other educational institutions use the instance of Dodona hosted at Ghent University, which is free to use for educational purposes.

Dodona currently hosts a collection of 17 500 learning activities that are freely available to all teachers, allowing them to create their own learning paths tailored to their teaching practice. In total, 70 000 students have submitted more than 18 million solutions to Dodona in the seven years that it has been running (Figures 3.1, 3.2 & 3.3).

In the year 2023, the highest number of monthly active users was reached in November, when 9 678 users submitted at least one solution. About half of these users are from secondary education, a quarter from Ghent University,

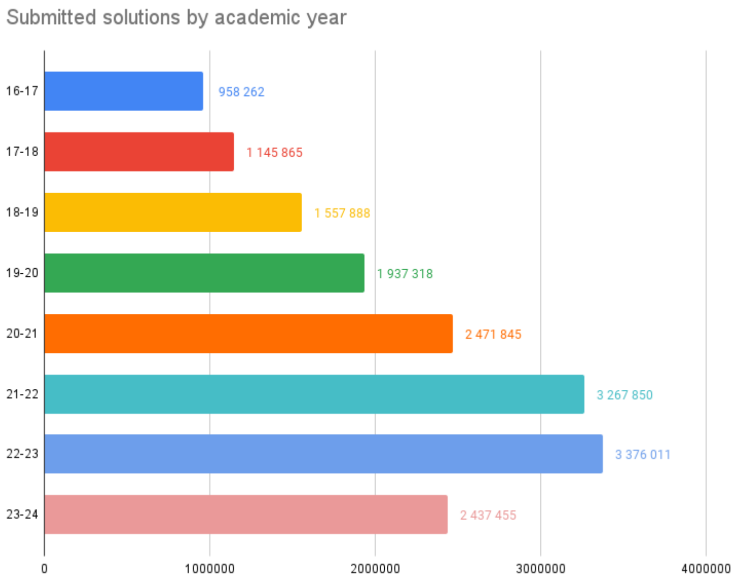


Figure 3.1: Overview of the number of submitted solutions by academic year. Note that the data for the academic year 2023–2024 is incomplete, since the academic year has not finished yet at the time of data collection (March 2024).

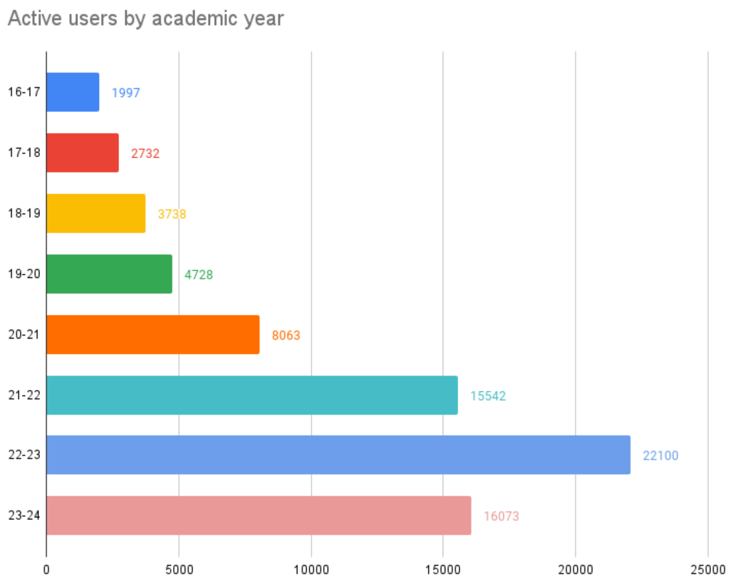


Figure 3.2: Overview of the number of active users by academic year. Users were active when they submitted at least one solution for a programming assignment during the academic year. Note that the data for the academic year 2023–2024 is incomplete, since the academic year has not finished yet at the time of data collection (March 2024).

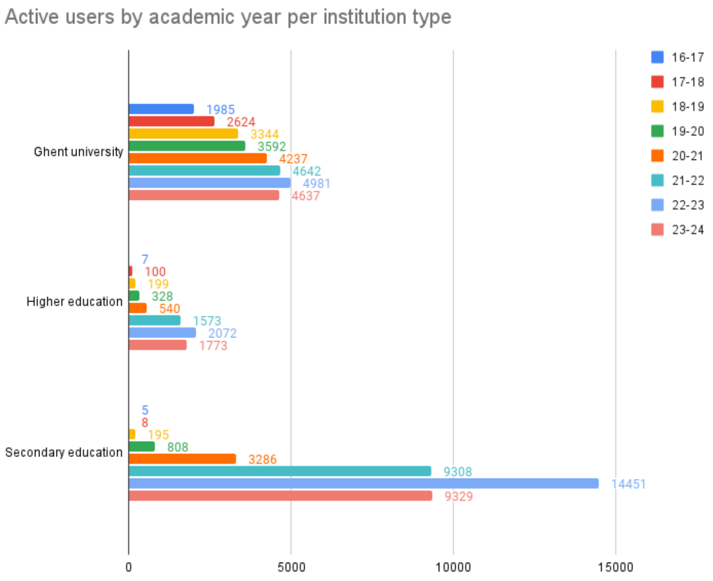


Figure 3.3: Overview of the number of active users by academic year per institution type. Users were active when they submitted at least one solution for a programming assignment during the academic year. Note that the data for the academic year 2023–2024 is incomplete, since the academic year has not finished yet at the time of data collection (March 2024).

and the rest mostly from other higher education institutions. Every year, we see the largest increase of new users during September, where the same ratios between Ghent University, higher, and secondary education are kept. The record for most submissions in one day was recently broken on the 12th of January 2024, when the course described in Section 3.2 had one exam for all students for the first time in its history, and those students submitted 38 771 solutions in total. Interestingly enough, the day before (the 11th of January) was the third-busiest day ever. The day with the most distinct users was the 23rd of October 2023, when there were 2 680 users who submitted at least one solution. This is due to the fact that there were a lot of exercise sessions on Fridays in the first semester of the academic year; a lot of the other Fridays at the start of the semester are also in the top 10 of busiest days ever (both in submissions and in number of users). The full top 10 of submissions can be seen in Table 3.1. The top 10 of active users can be seen in Table 3.2.

Date	# submissions
2024-01-12	38 771
2023-10-23	38 431
2024-01-11	38 148
2020-01-22	33 161
2023-10-09	32 668
2019-01-23	32 464
2023-10-02	32 447
2019-01-24	32 113
2023-11-06	30 896
2023-10-16	30 103

Table 3.1: Top 10 of days with the most submissions on Dodona. This analysis was done in March 2024.

In addition to the quantitative figures above, we also performed a qualitative user experience study of Dodona in 2018. 271 tertiary education students responded to a questionnaire that contained the following three questions: *i)* What are the things you value while working with Dodona? *ii)* What are the things that bother you while working with Dodona? *iii)* What are your suggestions for improvements to Dodona? Students praised its user-friendliness, beautiful interface, immediate feedback with visualized differences between expected and generated output, integration of the Python Tutor, linting feedback and large number of test cases. Negative points were related to differences between the students' local

Date	# active users
2023-10-23	2 680
2023-10-09	2 659
2023-11-20	2 581
2023-10-02	2 381
2023-10-16	2 364
2023-11-06	2 343
2023-10-17	2 287
2023-11-27	2 274
2022-10-03	2 265
2023-11-13	2 167

Table 3.2: Top 10 of days with the most users who submitted at least once on Dodona. This analysis was done in March 2024.

execution environments and the environment in which Dodona runs the tests, and the strictness with which the tests are evaluated. Other negative feedback was mostly related to individual courses the students were taking instead of the platform itself.

3.2 Use in a programming course

Since the academic year 2011–2012 we have organized an introductory Python course at Ghent University (Belgium) with a strong focus on active and online learning. Initially the course was offered twice a year in the first and second term, but from academic year 2014–2015 onwards it was only offered in the first term. The course is taken by a mix of undergraduate, graduate, and postgraduate students enrolled in various study programmes (mainly formal and natural sciences, but not computer science), with 442 students enrolled for the 2021–2022 edition[20].

3.2.1 Course structure

Each course edition has a fixed structure, with 13 weeks of educational activities subdivided in two successive instructional units that each cover five topics of the Python programming language – one topic per week – followed by a graded test about all topics covered in the unit (Figure 3.4).

[20] `https://dodona.be/courses/773/`

The final exam at the end of the term evaluates all topics covered in the entire course. Students who fail the course during the first exam in January can take a resit exam in August/September that gives them a second chance to pass the course.

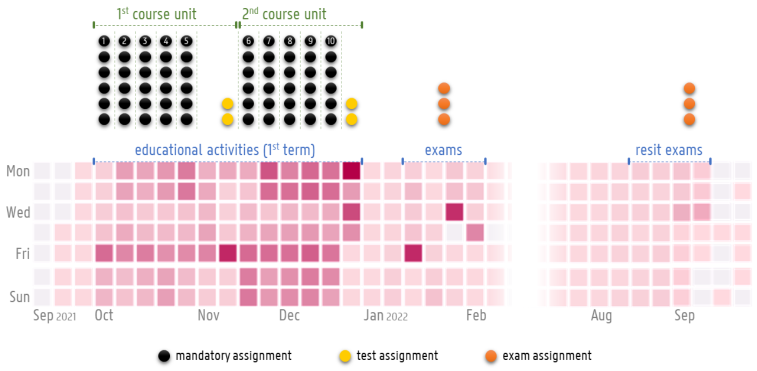


Figure 3.4: **Top**: Structure of the Python course that runs each academic year across a 13-week term (September–December). Students submit solutions for ten series with six mandatory assignments, two tests with two assignments and an exam with three assignments. There is also a resit exam with three assignments in August/September if they failed the first exam in January. **Bottom**: Heatmap from Dodona learning analytics page showing distribution per day of all 331 734 solutions submitted during the 2021–2022 edition of the course (442 students). The darker the colour, the more solutions were submitted that day. Weekly lab sessions for different groups on Monday afternoon, Friday morning and Friday afternoon, where we can see darker squares. Weekly deadlines for mandatory assignments on Tuesdays at 22:00. Three exam sessions for different groups in January. Two more exam sessions for different groups in August/September.

In the regular weeks, when a new programming topic is covered, students prepare themselves by reading the textbook chapters covering the topic, following the flipped classroom approach (Akçayır & Akçayır, 2018; Bishop & Verleger, 2013). Lectures are interactive programming sessions that aim at bridging the initial gap between theory and practice, advancing concepts, and engaging in collaborative learning (Tucker, 2012). Along the same lines, the first assignment for each topic is an ISBN-themed programming challenge whose model solution is shared with the students, together with an instructional video that works step-by-step towards the model solution. Students must then try to solve five other programming assignments on that topic before a deadline one week later. That results in 60 mandatory assignments across the semester. Students can work on their programming assignments during weekly computer labs, where they can collaborate in small groups and ask help from teaching assistants. They can also work on their assignments and submit solutions outside lab sessions. In addition to the mandatory assignments, students can further elaborate on their programming skills by tackling additional programming exercises they select from a pool of over 900 exercises linked to the ten programming topics. Submissions for these additional exercises are not taken into account in the final grade.

3.2.2 Assessment, feedback and grading

We use Dodona to promote students' active learning through problem-solving (Prince, 2004). Each course edition has its own dedicated course in Dodona, with a learning path containing all mandatory, test, and exam assignments grouped into series with corresponding deadlines. Mandatory assignments for the first unit are published at the start of the semester, and those for the second unit after the test of the first unit. For each test and exam we organize multiple sessions for different groups of students. Assignments for test and exam sessions are provided in a hidden series that is only accessible for students participating in the session using a shared token link. The test and exam assignments are published afterwards for all students, when grades are announced. Students can see class progress when working on their mandatory assignments, nudging them to avoid procrastination. Only teachers can see class progress for test and exam series so as not to accidentally stress out students. For the same reason, we intentionally organize tests and exams following exactly the same procedure, so that students can take high-stake exams in a familiar context and adjust their approach based on previous experiences. The only difference is that test assignments are not as hard as exam assignments, as students are still in the midst of learning programming skills when tests are taken.

Students are stimulated to use an integrated development environment (IDE) to work on their programming assignments. IDEs bundle a battery of programming tools to support today's generation of software developers in writing, building, running, testing, and debugging software. Working with such tools can be a true blessing for both seasoned and novice programmers, but there is no silver bullet (Brooks & Kugler, 1987). Learning to code remains inherently hard (Kelleher et al., 2002) and consists of challenges that are different to reading and learning natural languages (Fincher, 1999). As an additional aid, students can continuously submit (intermediate) solutions for their programming assignments and immediately receive automatically generated feedback upon each submission, even during tests and exams. Guided by that feedback, they can track potential errors in their code, remedy them and submit updated solutions. There is no restriction on the number of solutions that can be submitted per assignment. All submitted solutions are stored, but for each assignment only the last submission before the deadline is taken into account to grade students. This allows students to update their solutions after the deadline (i.e. after model solutions are published) without impacting their grades, as a way to further practice their programming skills. One effect of active learning, triggered by mandatory assignments with weekly deadlines and intermediate tests, is that most learning happens during the term (Figure 3.4). In contrast to other courses, students do not spend a lot of time practising their coding skills for this course in the days before an exam. We want to explicitly encourage this behaviour, because we strongly believe that one can not learn to code in a few days' time (Norvig, 2001).

For the assessment of tests and exams, we follow the line of thought that human expert feedback through source code annotations is a valuable complement to feedback coming from automated assessment, and that human interpretation is an absolute necessity when it comes to grading (Ala-Mutka, 2005; Jackson & Usher, 1997; Staubitz et al., 2015). We shifted from paper-based to digital code reviews and grading when support for manual assessment was released in version 3.7 of Dodona (summer 2020). Although online reviewing positively impacted our productivity, the biggest gain did not come from an immediate speed-up in the process of generating feedback and grades compared to the paper-based approach. While time-on-task remained about the same, our online source code reviews were much more elaborate than what we produced before on printed copies of student submissions. This was triggered by improved reusability of digital annotations and the foresight of streamlined feedback delivery. Where delivering custom feedback only requires a single click after the assessment of an evaluation has been completed in Dodona, it took us much more effort before to distribute our paper-based feedback. Students were direct beneficiaries from more and richer feedback, as observed from the fact that 75% of our students looked at their personalized feedback within 24 hours after it had been released, before we even published grades in Dodona. What did not change is the fact that we complement personalized feedback with collective feedback sessions in which we discuss model solutions for test and exam assignments, and the low numbers of questions we received from students on their personalized feedback. As a future development, we hope to reduce the time spent on manual assessment through improved computer-assisted reuse of digital source code annotations in Dodona (see Chapter 6).

We primarily rely on automated assessment as a first step in providing formative feedback while students work on their mandatory assignments. After all, a back-of-the-envelope calculation tells us it would take us 72 full-time equivalents (FTE) to generate equivalent amounts of manual feedback for mandatory assignments compared to what we do for tests and exams. In addition to volume, automated assessment also yields the responsiveness needed to establish an interactive feedback loop (Gibbs & Simpson, 2005). Automated assessment thus allows us to motivate students working through enough programming assignments and to stimulate their self-monitoring and self-regulated learning (Pintrich, 1995; Schunk & Zimmerman, 1994). It results in triggering additional questions from students that we manage to respond to with one-to-one personalized human tutoring, either synchronously during hands-on sessions or asynchronously through Dodona's Q&A module. We observe that individual students seem to have a strong bias towards either asking for face-to-face help during hands-on sessions or asking questions online. This could be influenced by the time when they mainly work on their assignments, by their way of collaboration on assignments, or by reservations because of perceived threats to self-esteem or social embarrassment (Karabenick & Knapp, 1991; Newman & Schwager, 1993).

In computing a final score for the course, we try to find an appropriate balance between stimulating students to find solutions for programming assignments themselves and collaborating with and learning from peers, instructors and teachers while working on assignments. The final score is computed as the sum of a score obtained for the exam (80%) and a score for each unit that combines the student's performance on the mandatory and test assignments (10% per unit). We use Dodona's grading module to determine scores for tests and exams based on correctness, programming style, choice made between the use of different programming techniques and the overall quality of the implementation. The score for a unit is calculated as

$$s \times f$$

where s is the score for the two test assignments and f is the fraction of mandatory assignments the student has solved correctly. A solution for a mandatory assignment is considered correct if it passes all unit tests. Evaluating mandatory assignments therefore does not require any human intervention, except for writing unit tests when designing the assignments, and is performed entirely by our Python judge. In our experience, most students traditionally perform much better on mandatory assignments compared to test and exam assignments (Glass & Kang, 2022), given the possibilities for collaboration on mandatory assignments.

3.2.3 Open and collaborative learning environment

We strongly believe that effective collaboration among small groups of students is beneficial for learning (Prince, 2004), and encourage students to collaborate and ask questions to tutors and other students during and outside lab sessions. We also demonstrate how they can embrace collaborative coding and pair programming services provided by modern integrated development environments (Hanks et al., 2011; Williams et al., 2002). However, we do recommend them to collaborate in groups of no more than three students, and to exchange and discuss ideas and strategies for solving assignments rather than sharing literal code with each other. After all, our main reason for working with mandatory assignments is to give students sufficient opportunity to learn topic-oriented programming skills by applying them in practice, and shared solutions spoil the learning experience. The factor f in the score for a unit encourages students to keep fine-tuning their solutions for programming assignments until all test cases succeed before the deadline passes. But maximizing that factor without proper learning of programming skills will likely yield a low test score s and thus an overall low score for the unit, even if many mandatory exercises were solved correctly.

Fostering an open collaboration environment to work on mandatory assignments with strict deadlines and taking them into account for computing the final score is a potential promoter for plagiarism, but using it as a weight factor for the test score rather than as an independent score item should promote learning by avoiding that plagiarism is rewarded. It takes some effort to properly explain this to students. We initially used MOSS (Schleimer et al., 2003) and now use Dolos (Maertens et al., 2022) to monitor submitted solutions for mandatory assignments, both before and at the deadline. The solution space for the first few mandatory assignments is too small for linking high similarity to plagiarism:

submitted solutions only contain a few lines of code and the diversity of implementation strategies is small. But at some point, as the solution space broadens, we start to see highly similar solutions that are reliable signals of code exchange among larger groups of students. Strikingly this usually happens among students enrolled in the same study programme (Figure 3.5). As soon as this happens – typically in week 3 or 4 of the course – plagiarism is discussed during the next lecture. Usually this is a lecture about working with the string data type, so we can introduce plagiarism detection as a possible application of string processing.

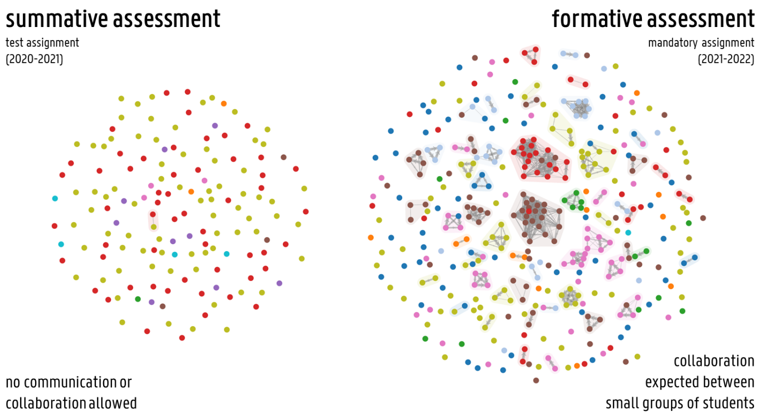


Figure 3.5: Dolos plagiarism graphs for the Python programming assignment "π-ramidal constants" that was created and used for a test of the 2020–2021 edition of the course (left) and reused as a mandatory assignment in the 2021–2022 edition (right). Graphs constructed from the last submission before the deadline of 142 and 382 students respectively. The colour of each node represents the student's study programme. Edges connect highly similar pairs of submissions, with similarity threshold set to 0.8 in both graphs. Edge directions are based on submission timestamps in Dodona. Clusters of connected nodes are highlighted with a distinct background colour and have one node with a solid border that indicates the first correct submission among all submissions in that cluster. All students submitted unique solutions during the test, except for two students who confessed they exchanged a solution during the test. Submissions for the mandatory assignment show that most students work either individually or in groups of two or three students, but we also observe some clusters of four or more students that exchanged solutions and submitted them with hardly any varying types and amounts of modifications.

In an announcement entitled “copy-paste $\neq$ learn to code” we show students some pseudonymized Dolos plagiarism graphs that act as mirrors to make them reflect upon which node in the graph they could be (Figure 3.5). We stress that the learning effect dramatically drops in groups of four or more students. Typically, we notice that in such a group only one or a few students make the effort to learn to code, while the other students usually piggyback by copy-pasting solutions. We make students aware that understanding someone else’s code for programming assignments is a lot easier than trying to find solutions themselves. Over the years, we have experienced that a lot of students are caught in the trap of genuinely believing that being able to understand code is the same as being able to write code that solves a problem until they take a test at the end of a unit. That’s where the s factor of the test score comes into play. After all, the goal of summative tests is to evaluate if individual students have acquired the skills to solve programming challenges on their own.

When talking to students about plagiarism, we also point out that the plagiarism graphs are directed graphs, indicating which student is the potential source of exchanging a solution among a cluster of students. We specifically address these students by pointing out that they are probably good at programming and might want to exchange their solutions with other students in a way to help their peers. Instead of really helping them out though, they actually take away learning opportunities from their fellow students by giving away the solution as a spoiler. Stated differently, they help maximize the factor f but effectively also reduce the s factor of the test score, where both factors need to be high to yield a high score for the unit. After this lecture, we usually notice a stark decline in the number of plagiarized solutions.

The goal of plagiarism detection at this stage is prevention rather than penalization, because we want students to take responsibility over their learning. The combination of realizing that teachers and instructors can easily detect plagiarism and an upcoming test that evaluates if students can solve programming challenges on their own, usually has an immediate and persistent effect on reducing cluster sizes in the plagiarism graphs to at most three students. At the same time, the signal is given that plagiarism detection is one of the tools we have to detect fraud during tests and exams. The entire group of students is only addressed once about plagiarism, without going into detail about how plagiarism detection itself works, because we believe that overemphasizing this topic is not very effective and explaining how it works might drive students towards spending time thinking on how they could bypass the detection process, which is time they'd better spend on learning to code. Every three or four years we see a persistent cluster of students exchanging code for mandatory assignments over multiple weeks. If this is the case, we individually address these students to point them again on their responsibilities, again differentiating between students that share their solution and students that receive solutions from others.

Tests and exams, on the other hand, are taken on-campus under human surveillance and allow no communication with fellow students or other persons (and, more recently, also no generative AI). Students can work on their personal computers and get exactly two hours to solve two programming assignments during a test, and three hours and thirty minutes to solve three programming assignments during an exam.

Tests and exams are "open book/open Internet", so any hard copy and digital resources can be consulted while solving test or exam assignments. Students are instructed that they can only be passive users of the Internet: all information available on the Internet at the start of a test or exam can be consulted, but no new information can be added. When taking over code fragments from the Internet, students have to add a proper citation as a comment in their submitted source code.

After each test and exam, we again use MOSS/Dolos to detect and inspect highly similar code snippets among submitted solutions and to find convincing evidence they result from exchange of code or other forms of interpersonal communication (Figure 3.5). If we catalogue cases as plagiarism beyond reasonable doubt, the examination board is informed to take further action (Maertens et al., 2022).

3.2.4 Workload for running a course edition

To organize "open book/open Internet" tests and exams that are valid and reliable, we always create new assignments and avoid assignments whose solutions or parts thereof are readily available online. At the start of a test or exam, we share a token link that gives students access to the assignments in a hidden series on Dodona.

For each edition of the course, mandatory assignments were initially a combination of selected test and exam exercises reused from the previous edition of the course and newly designed exercises. The former to give students an idea about the level of exercises they can expect during tests and exams, and the latter to avoid solution slippage. As feedback for the students we publish sample solutions for all mandatory exercises after the weekly deadline has passed. This also indicates that students must strictly adhere to deadlines, because sample solutions are available afterwards. As deadlines are very clear and adjusted to timezone settings in Dodona, we never experience discussions with students about deadlines.

After nine editions of the course, we felt we had a large enough portfolio of exercises to start reusing mandatory exercises from four or more years ago instead of designing new exercises for each edition. However, we still continue to design new exercises for each test and exam. After each test and exam, exercises are published and students receive manual reviews on the code they submitted, on top of the automated feedback they already got during the test or exam. But in contrast to mandatory exercises we do not publish sample solutions for test and exam exercises, so that these exercises can be reused during the next edition of the course. When students ask for sample solutions of test or exam exercises, we explain that we want to give the next generation of students the same learning opportunities they had.

So far, we have created more than 900 programming assignments for this introductory Python course alone. All these assignments are publicly shared on Dodona as open educational resources (Caswell et al., 2008; Downes, 2007; Hylén, 2021; Tuomi, 2013; Wiley et al., 2014). They are used in many other courses on Dodona (on average 10.8 courses per assignment) and by many students (on average 503.7 students and 4 801.5 submitted solutions per assignment). We estimate that it takes about 10 person-hours on average to create a new assignment for a test or an exam: 2 hours for coming up with an idea, 30 minutes for implementing and tweaking a sample solution that meets the educational goals of the assignment and can be used to generate a test suite for automated assessment, 4 hours for describing the assignment (including background research), 30 minutes for translating the description from Dutch into English, one hour to configure support for automated assessment, and another 2 hours for reviewing the result by some extra pairs of eyes.

Generating a test suite usually takes 30 to 60 minutes for assignments that can rely on basic test and feedback generation features that are built into the judge. The configuration for automated assessment might take 2 to 3 hours for assignments that require more elaborate test generation or that need to extend the judge with custom components for dedicated forms of assessment (e.g. assessing non-deterministic behaviour) or feedback generation (e.g. generating visual feedback). Keuning et al. (2018) found that publications rarely describe how difficult and time-consuming it is to add assignments to automated assessment platforms, or even if this is possible at all.

The ease of extending Dodona with new programming assignments is reflected by more than 17 500 assignments that have been added to the platform so far. Our experience is that configuring support for automated assessment only takes a fraction of the total time for designing and implementing assignments for our programming course, and in absolute numbers stays far away from the one person-week reported for adding assignments to Bridge (Bonar & Cunningham, 1988). Because the automated assessment infrastructure of Dodona provides common resources and functionality through a Docker container and a judge, the assignment-specific configuration usually remains lightweight. Only around 5% of the assignments need extensions on top of the built-in test and feedback generation features of the judge.

So how much effort does it cost us to run one edition of our programming course? For the most recent 2021–2022 edition we estimate about 34 person-weeks in total (Table 3.3), the bulk of which is spent on on-campus tutoring of students during hands-on sessions (30%), manual assessment and grading (22%), and creating new assignments (21%). About half of the workload (53%) is devoted to summative feedback through tests and exams: creating assignments, supervision, manual assessment and grading. Most of the other work (42%) goes into providing formative feedback through on-campus and online assistance while students work on their mandatory assignments. Out of 2 215 questions that students asked through Dodona's online Q&A module, 1 983 (90%) were answered by teaching assistants and 232 (10%) were marked as answered by the student who originally asked the question. Because automated assessment provides first-line support, the need for human tutoring is already heavily reduced. We have drastically cut the time we initially spent on mandatory assignments by reusing existing assignments and because the Python judge is stable enough to require hardly any maintenance or further development.

3.2.5 Learning analytics and educational data mining

A longitudinal analysis of student submissions across the term shows that most learning happens during the 13 weeks of educational activities and that students do not have to catch up practising their programming skills during the exam period (Figure 3.4). Active learning thus effectively avoids

Task	Estimated workload (hours)
Lectures	60
Mandatory assignments	540
Select assignments	10
Review selected assignments	30
Tips & tricks	10
Automated assessment	0
Hands-on sessions	390
Answering online questions	100
Tests & exams	690
Create new assignments	270
Supervise tests and exams	130
Automated assessment	0
Manual assessment	288
Plagiarism detection	2
Total	1 290

Table 3.3: Estimated workload to run the 2021–2022 edition of the introductory Python programming course for 442 students with 1 lecturer, 7 teaching assistants and 3 undergraduate students who serve as teaching assistants (Gordon et al., 2013).

procrastination. We observe that students submit solutions every day of the week and show increased activity around hands-on sessions and in the run-up to the weekly deadlines (Figure 3.6). Weekends are also used to work further on programming assignments, but students seem to be watching over a good night's sleep.

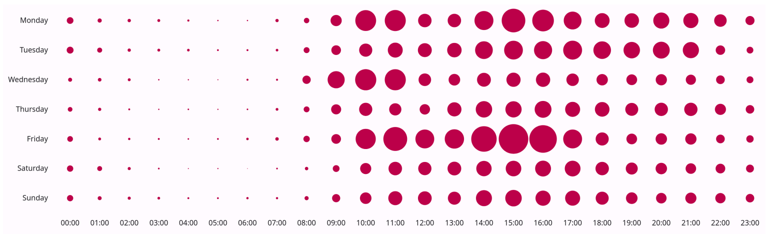


Figure 3.6: Punchcard from the Dodona learning analytics page showing the distribution per weekday and per hour of all 331 734 solutions submitted during the 2021–2022 edition of the course (442 students).

Throughout a course edition, we use Dodona's series analytics to monitor how students perform on our selection of programming assignments (Figures 3.7, 3.8, and 3.9). This allows us to make informed decisions and appropriate interventions, for example when students experience issues with the automated assessment configuration of a particular assignment or if the original order of assignments in a series does not seem to align with our design goal to present them in increasing order of difficulty. The first students that start working on assignments usually are good performers. Seeing these early birds having trouble with solving one of the assignments may give an early warning that action is needed, such as improving the problem specification, adding extra tips & tricks, or better explaining certain programming concepts to all students during lectures or hands-on sessions. Reversely, observing that many students postpone working on their assignments until just before the deadline might indicate that some assignments are simply too hard at this moment in time through the learning pathway of the students or that completing the collection of programming assignments interferes with the workload from other courses. Such "deadline hugging" patterns are also a good breeding ground for students to resort on exchanging solutions among each other.

Using educational data mining techniques on historical data exported from several editions of the course, we further investigated what aspects of practising programming skills promote or inhibit learning, or have no or minor effect on the learning process (see Chapter 5). It will not come as a

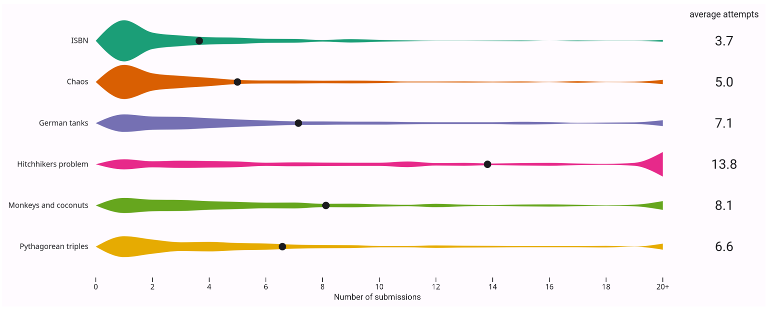


Figure 3.7: Distribution of the number of student submissions per programming assignment. The larger the zone, the more students submitted a particular number of solutions. Black dot indicates the average number of submissions per student.

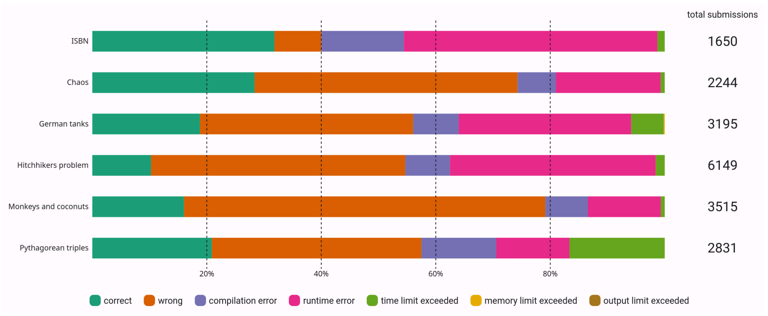


Figure 3.8: Distribution of top-level submission statuses per programming assignment.

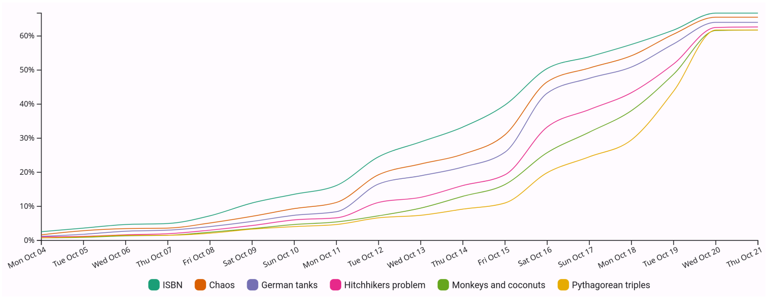


Figure 3.9: Progression over time of the percentage of students that correctly solved each assignment. The visualisation starts two weeks before the deadline, which is on the 19th of October.

surprise that midterm test scores are good predictors for a student's final grade, because tests and exams are both summative assessments that are organized and graded in the same way. However, we found that organizing a final exam end-of-term is still a catalyst of learning, even for courses with a strong focus of active learning during weeks of educational activities.

In evaluating if students gain deeper understanding when learning from their mistakes while working progressively on their programming assignments, we found the old adage that practice makes perfect to depend on what kind of mistakes students make. Learning to code requires mastering two major competences: *i)* getting familiar with the syntax and semantics of a programming language to express the steps for solving a problem in a formal way, so that the algorithm can be executed by a computer, and *ii)* problem-solving itself. It turns out that staying stuck longer on compilation errors (mistakes against the syntax of the programming language) inhibits learning, whereas taking progressively more time to get rid of logical errors (reflective of solving a problem with a wrong algorithm) as assignments get more complex actually promotes learning. After all, time spent in discovering solution strategies while thinking about logical errors can be reclaimed multifold when confronted with similar issues in later assignments (Glass & Kang, 2022).

These findings neatly align with the claim of Edwards et al. (2018) that problem-solving is a higher-order learning task in the Taxonomy by Bloom et al. (1956) (analysis and synthesis) than language syntax (knowledge, comprehension, and application).

Using historical data from previous course editions, we can also make highly accurate predictions about what students will pass or fail the current course edition (see Chapter 5). This can already be done after a few weeks into the course, so remedial actions for at-risk students can be started well in time. The approach is privacy-friendly as we only need to process metadata on student submissions for programming assignments and results from automated and manual assessment extracted from Dodona. Given that cohort sizes are large enough, historical data from a single course edition are already enough to make accurate predictions.

3.3 Conclusion

Dodona has grown into a widely used automated assessment platform. As we have shown in this chapter, both students and teachers alike appreciate its extensive feature set and user-friendliness. By exploiting all Dodona features, it is possible to design and implement a highly activating course. While there is still a lot of time invested in running a course like this, the time Dodona saves can be reinvested in hands-on guidance of students and giving manual feedback on evaluations and examinations.

4 Under the hood: technical architecture and design

Dodona and its ecosystem comprise a lot of code. This chapter answers the question of what technical work goes into building a platform like Dodona. We do this by discussing the technical background of Dodona itself (Van Petegem et al., 2023). As mentioned in Chapter 2, Dodona is the fruit of a collaborative effort of the entire Dodona team.

We also present a stand-alone online code editor, Papyros (`https://papyros.dodona.be`), that was integrated into Dodona (De Ridder et al., 2022). This work was done by Winnie De Ridder in his master's thesis, under supervision and guidance of myself, prof. Bart Mesuere and prof. Peter Dawyndt.

We also discuss two judges that were developed in the context of this dissertation. The R judge, which was written entirely by myself (Nüst et al., 2020).

The TESTed judge was first prototyped in my master's thesis (Van Petegem & Dawyndt, 2018) and was further developed in two other master's theses by Niko Strijbol (Strijbol et al., 2020) and Boris Sels (Sels et al., 2021), which I also supervised and guided along with prof. Peter Dawyndt.

In this chapter we assume the reader is familiar with Dodona's features and how they are used, as detailed in Chapters 2 and 3.

4.1 Dodona

To ensure that Dodona[21] is robust against sudden increases in workload and when serving hundreds of concurrent users, it has a multi-tier service architecture that delegates different parts of the application to different servers, as can be seen on Figure 4.1. More specifically, the web server,

[21] `https://github.com/dodona-edu/dodona`

database (MySQL) and caching system (Memcached) each run on their own machine. In addition, a scalable pool of interchangeable worker servers are available to automatically assess incoming student submissions. In this section, we will highlight a few of these components.

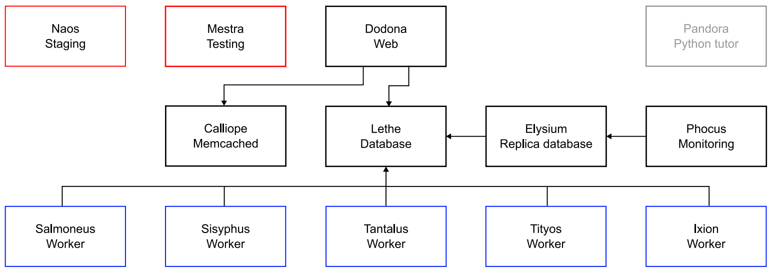


Figure 4.1: Diagram of all the servers involved with running and developing Dodona. The role of each server in the deployment is listed below its name. Worker servers are marked in blue, development servers are marked in red. Servers are connected if they communicate with each other. The direction of the connection signifies which server initiates the connection. Every server also has an implicit connection with Phocus (the monitoring server), since metrics such as load, CPU usage, disk usage, etc. are collected and sent to Phocus on every server. The Pandora server is greyed out because it has been decommissioned (see Section 4.1.3 for more info).

4.1.1 The Dodona web application

The user-facing part of Dodona runs on the main web server, which is also called Dodona (see Figure 4.1). Dodona is a Ruby-on-Rails web application, currently running on Ruby 3.1 and Rails 7.1. We use Apache 2.4.52 to proxy our requests to the actual application. We follow the Rails-standard way of organizing functionality in models, views and controllers. In Rails, requests are sent to the appropriate action in the appropriate controller by the router. In these actions, models are queried and/or edited, after which they are used to construct the data for a response. This data is then rendered by the corresponding view, which can be HTML, JSON, JavaScript, or even a CSV.

The way we handle complex logic in the frontend has seen a number of changes along the years. When Dodona was started, there were only a few places where JavaScript was used. Dodona also used the Rails-standard way of serving dynamically generated JavaScript to replace parts of pages (e.g. for pagination or search). With the introduction of more complex features like evaluations, we switched to using lightweight web components where this made sense. We also eliminated jQuery, because more and more of its functionality was implemented natively by browsers. And lastly, all JavaScript was rewritten to TypeScript.

Security and performance

Another important aspect of running a public web application is its security. Dodona needs to operate in a challenging environment where students simultaneously submit untrusted code to be executed on its servers ("remote code execution as a service") and expect automatically generated feedback, ideally within a few seconds. Many design decisions are therefore aimed at maintaining and improving the performance, reliability, and security of its systems. This includes using Cloudflare as a CDN and common protections such as a Content Security Policy or Cross Site Request Forgery protection, but is also reflected in the implementation of Dodona itself, as we will explain in this section.

Since Dodona grew from being used to teach mostly by people we knew personally to being used in secondary schools all over Flanders, we went from being able to fully trust exercise authors to having this trust reduced (as it is impossible for a team of our size to vet all the people we give teacher's rights in Dodona). This meant that our threat model and therefore the security measures we had to take also changed over the years. Once Dodona was opened up to more and more teachers, we gradually locked down what teachers could do with e.g. their exercise descriptions. Content where teachers can inject raw HTML into Dodona was moved to iframes, to make sure that teachers could still be as creative as they wanted while writing exercises, while simultaneously not allowing them to execute JavaScript in a session where users are logged in. For user content where this creative freedom is not as necessary (e.g. series or course descriptions), but some Markdown/HTML content is still wanted, we sanitize the (generated) HTML so that it can only include HTML elements and attributes that are specifically allowed.

One of the most important components of Dodona is the feedback shown after a submission is evaluated (as seen in Figure 2.6). It has, therefore, seen a lot of security, optimization and UI work over the years. Judge and exercise authors (and even students, through their submissions) can determine a lot of the content that eventually ends up in the feedback. Therefore, the same sanitization that is used for series and course descriptions is used for the messages that are added to the feedback (since these can contain Markdown and arbitrary HTML as well). The increase in teachers that added exercises to Dodona also meant that the variety in feedback given grew, sometimes resulting in a huge volume of testcases and long output.

Optimization work was needed to cope with this volume of feedback. For example, one of the biggest optimizations was in how expected and generated feedback are diffed and how these diffs are rendered. When Dodona was first written, the library used for creating diffs of the generated and expected results (`diffy`[22]) actually shelled out to the GNU `diff` command. This output was parsed and transformed into HTML by the library using find and replace operations. As one can expect, starting a new process and doing a lot of string operations every time outputs had to be diffed resulted in very slow loading times. The library was replaced with a pure Ruby library (`diff-lcs`[23]), and its outputs were built into HTML using Rails' efficient `Builder` class. This change of diffing method also fixed a number of bugs we were experiencing along the way.

Even this was not enough to handle the most extreme of exercises though. Diffing hundreds of lines hundreds of times still takes a long time, even if done in-process while optimized by a JIT. The resulting feedback also contained so much HTML that the browsers on our development machines (which are pretty powerful machines) noticeably slowed down when loading and rendering them. To handle these cases, we needed to do less work and needed to output less HTML. We decided to only diff line-by-line (instead of character-by-character) in most of these cases and to not diff at all in the most extreme cases, reducing the amount of HTML required to render them as well. This was also motivated by usability. If there are lots of small differences between a very long generated and expected output, the diff view in the feedback could also become visually overwhelming for students.

4.1.2 Judging submissions

Student submissions are automatically assessed in background jobs by our worker servers (Salmoneus, Sisyphus, Tantalus, Tityos and Ixion;

[22]`https://github.com/samg/diffy`
[23]`https://github.com/halostatue/diff-lcs`

Figure 4.1). To divide the work over these servers we make use of a job queue, based on `delayed_job`[24]. Each worker server has 6 job runners, which regularly poll the job queue when idle.

For proper virtualization we use Docker containers (Peveler et al., 2019) that use OS-level containerization technologies and define runtime environments in which all data and executable software (e.g. scripts, compilers, interpreters, linters, database systems) are provided and executed. These resources are typically pre-installed in the image of the container. Prior to launching the actual assessment, the container is extended with the submission, the judge and the resources included in the assessment configuration (Figure 4.2). Additional resources can be downloaded and/or installed during the assessment itself, provided that Internet access is granted to the container. When the container is started, limits are placed on the amount of resources it can consume. This includes a limit in runtime, memory usage, disk usage, network access and the number of processes a container can have running at the same time. Some of these limits are (partially) configurable per exercise, but sane upper bounds are always applied. This is also the case for network access, where even if the container is allowed internet access, it can not access other Dodona hosts (such as the database server).

The actual assessment of the student submission is done by a software component called a *judge* (Wasik et al., 2018). The judge must be robust enough to provide feedback on all possible submissions for the assignment, especially submissions that are incorrect or deliberately want to tamper with the automatic assessment procedure (Forisek, 2006). Following the principles of software reuse, the judge is ideally also a generic framework that can be used to assess submissions for multiple assignments. This is enabled by the submission metadata that is passed when calling the judge, which includes the path to the source code of the submission, the path to the assessment resources of the assignment and other metadata such as programming language, natural language, time limit and memory limit.

Rather than providing a fixed set of judges, Dodona adopts a minimalistic interface that allows third parties to create new judges: automatic assessment is bootstrapped by launching the judge's `run` executable that can fetch the JSON formatted submission metadata from standard input and must generate JSON formatted feedback on standard output. The feedback has a standardized hierarchical structure that is specified in a

[24]`https://github.com/collectiveidea/delayed_job`

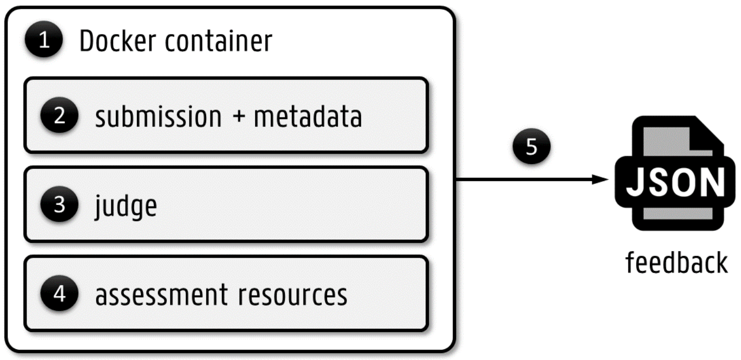


Figure 4.2: Outline of the procedure to automatically assess a student submission for a programming assignment. Dodona instantiates a Docker container (1) from the image linked to the assignment (or from the default image linked to the judge of the assignment) and loads the submission and its metadata (2), the judge linked to the assignment (3) and the assessment resources of the assignment (4) into the container. Dodona then launches the actual assessment, collects and bundles the generated feedback (5), and stores it into a database along with the submission and its metadata.

JSON schema[25]. At the lowest level, *tests* are a form of structured feedback expressed as a pair of generated and expected results. They typically test some behaviour of the submitted code against expected behaviour. Tests can have a brief description and snippets of unstructured feedback called messages. Descriptions and messages can be formatted as plain text, HTML (including images), Markdown, or source code. Tests can be grouped into *test cases*, which in turn can be grouped into *contexts* and eventually into *tabs*. All these hierarchical levels can have descriptions and messages of their own and serve no other purpose than visually grouping tests in the user interface. At the top level, a submission has a fine-grained status that reflects the overall assessment of the submission: `compilation error` (the submitted code did not compile), `runtime error` (executing the submitted code failed during assessment), `memory limit exceeded` (memory limit was exceeded during assessment), `time limit exceeded` (assessment did not complete within the given time), `output limit exceeded` (too much output was generated during assessment), `wrong` (assessment completed but not all strict requirements were fulfilled), or `correct` (assessment completed, and all strict requirements were fulfilled).

Taken together, a Docker image, a judge and a programming assignment configuration (including both a description and an assessment configuration) constitute a *task package* as defined by (Verhoeff, 2008): a unit Dodona uses to render the description of the assignment and to automatically assess its submissions. However, Dodona's layered design embodies the separation of concerns (Laplante, 2007) needed to develop, update and maintain the three modules in isolation and to maximize their reuse: multiple judges can use the same docker image and multiple programming assignments can use the same judge. Related to this, an explicit design goal for judges is to make the assessment configuration for individual assignments as lightweight as possible. After all, minimal configurations reduce the time and effort teachers and instructors need to create programming assignments that support automated assessment. Sharing of data files and multimedia content among the programming assignments in a repository also implements the inheritance mechanism for *bundle packages* as hinted by Verhoeff (2008). Another form of inheritance is specifying default assessment configurations at the directory level, which takes advantage of the hierarchical grouping of learning activities in a repository to share common settings.

[25]`https://github.com/dodona-edu/dodona/tree/main/public/schemas`

4.1.3 Python Tutor

The Python Tutor (Guo, 2013) is a debugger built into Dodona. It provides timeline debugging, where for each step in the timeline, each corresponding to a line being executed, all variables on the stack are visualized.

The deployment of the Python Tutor also saw a number of changes over the years. The Python Tutor itself is written in Python, so could not be part of Dodona itself. It started out as a Docker container on the same server as the main Dodona web application. Because it is used mainly by students who want to figure out their mistakes, the service responsible for running student code could become overwhelmed and in extreme cases even make the entire server unresponsive. After we identified this issue, the Python Tutor was moved to its own server (Pandora in Figure 4.1). This did not fix the Tutor itself becoming overwhelmed however, which meant that students that depended on the Tutor were sometimes unable to use it. This of course happened more during periods where the Tutor was being used a lot, such as evaluations and exams. One can imagine that the experience for students who are already quite stressed out about the exam they are taking when the Tutor suddenly failed was not very good. In the meantime, we had started to experiment with running Python code client-side in the browser (see Section 4.2 for more info). Because these experiments were successful, we migrated the Python Tutor from its own server to being run by students in their own browser using Pyodide. This means that the only student that can be impacted by the Python Tutor failing for a testcase is the student themselves (and because the Tutor is being run on a device that is under a far less heavy load, the Python Tutor fails much less often). In practice, we got no questions or complaints about the Python Tutor's performance after these changes, even during exams where 460 students were submitting simultaneously.

4.1.4 Development process

Development of Dodona is done on GitHub. Over the years, Dodona has seen over 16 500 commits by 26 contributors, and there have been 343 releases. All new features and bug fixes are added to the `main` branch through pull requests, of which there have been about 4 000. These pull requests are reviewed by (at least) two developers of the Dodona team before they are merged. We also treat pull requests as a form of internal documentation by writing an extensive description and adding screenshots for all visual changes or additions. The extensive test suite also runs automatically for every pull request (using GitHub Actions), and developers are encouraged to add new tests for each feature or bug fix. We've also made it very easy to deploy to our testing (Mestra) and staging (Naos) environments so that reviewers can test changes without having to spin up their local development instance of Dodona. These are the two unconnected servers seen in Figure 4.1. Mestra runs a Dodona instance much like the instance developers use locally. There is no production data present and in fact, the database is wiped and reseeded on every deploy. Naos is much closer to the production setup. It runs on a pseudonymized version of the production database, and has all the judges configured.

We also make sure that our dependencies are always up-to-date using Dependabot[26]. By updating our dependencies regularly, we make sure that we are not met by incompatibilities that take a long time to integrate when there is an important security update. Since Dodona is accessible over the public web, it would be problematic if we could not quickly apply security updates.

The way we release Dodona has seen a few changes over the years. We've gone from a few large releases with bugfix point-releases between them, to lots of smaller releases, to now a *release* per pull request. Releasing every pull request immediately after merging makes getting feedback from our users a very quick process. When we did versioned releases we also wrote release notes at the time of release. Because we do not have versioned releases any more, we now bundle the changes into release notes for every month. They are mostly autogenerated from the merged PRs, but bigger features are given more context and explanation.

4.1.5 Deployment process

After a pull request is merged, it is automatically deployed by a GitHub action. This action first runs all the tests again, deploys to the staging server and then deploys to the production servers. Since Naos has a copy of the production database, the deploy would be stopped if there are any migrations that fail in production. This way we can be sure the actual production database is never in an inconsistent migration state. The actual deployment is done by Capistrano[27]. Capistrano allows us to roll back

[26]`https://docs.github.com/en/code-security/dependabot/working-with-dependabot`
[27]`https://capistranorb.com/`

any deploys and makes clever use of symlinking to make sure that deploys happen without any service interruption.

Backups of the database are automatically saved every day and kept for 12 months. The backups are rotated according to a grandfather-father-son scheme (Jessen, 2010). The backups are taken by dumping a replica database. The replica database is used because dumping the main database write-locks it while it is being dumped, which would result in Dodona being unusable for a significant amount of time. We regularly test the backups by restoring them on Naos.

We also have an extensive monitoring and alerting system in place, based on Grafana[28]. This gives us some superficial analytics about Dodona usage, but can also tell us if there are problems with one of our servers. See Figure 4.3 for an example of the data this dashboard gives us. The analytics are also calculated using the replica database to avoid putting unnecessary load on our main production database.

The web server and worker servers also send notifications when an error occurs in their runtime. This is one of the main ways we discover bugs that got through our tests, since our users do not regularly report bugs themselves. We also get notified when there are long-running requests, since we consider our users having to wait a long time to see the page they requested a bug in itself. These notifications were an important driver to optimize some pages or to make certain operations asynchronous.

4.2 Papyros

Papyros[29] is a stand-alone basic online IDE we developed, primarily focused on secondary education (see Figure 4.4 for a screenshot). Recurring feedback we got from secondary education teachers when introducing Dodona to them was that Dodona did not have a simple way for students to run and test their code themselves. Testing their code in this case also means manually typing a response to an input prompt when an `input` statement is run by the interpreter. In the educational practice that Dodona was born out of, this was an explicit design goal. We wanted to guide students to use an IDE locally instead of programming in Dodona directly, since if they needed to program later in life, they would not have Dodona available as their programming environment. This same

[28]`https://grafana.com/`
[29]`https://github.com/dodona-edu/papyros`

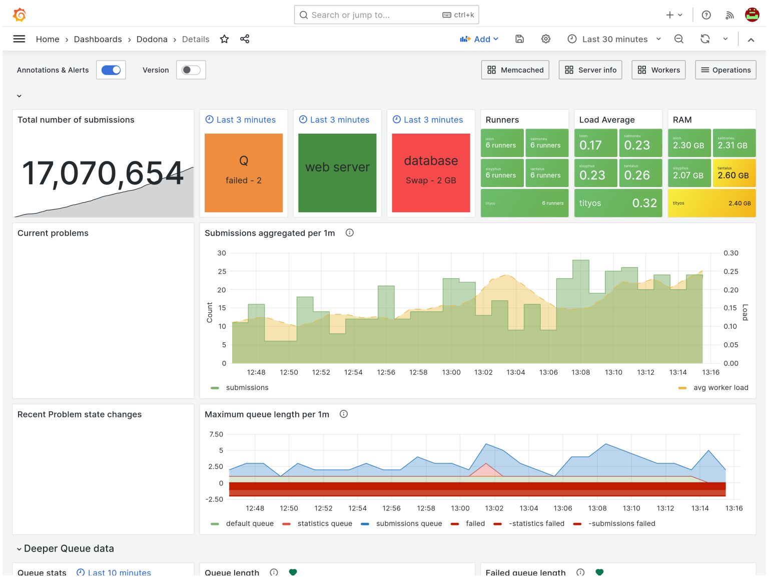
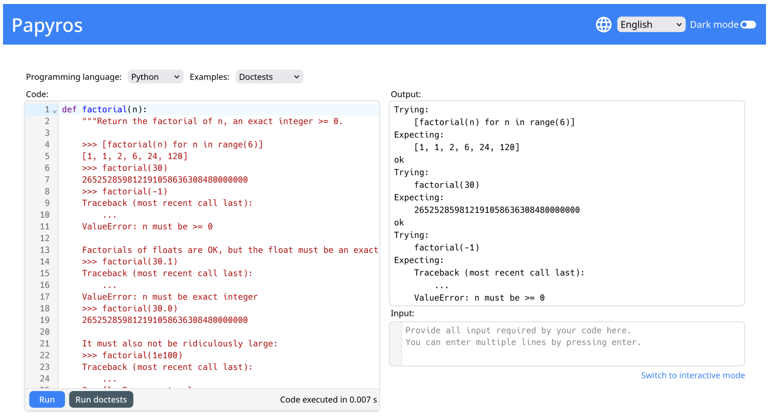


Figure 4.3: Grafana dashboard for Dodona, giving a quick overview of important metrics.

goal is not present in secondary education. In that context, the challenge of programming is already big enough, without complicating things by installing a real IDE with a lot of buttons and menus that students will never use. Students might also be working on devices that they do not own (PCs in the school), where installing an IDE might not even be possible.



```python
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
```

```
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
```

```
Provide all input required by your code here.
You can enter multiple lines by pressing enter.
```

Figure 4.4: User interface of Papyros. The editor can be seen on the left, with the output window to the right of it. The input window is below the output window and is currently in batch mode. All empty text fields have placeholder text that explains how they can be used.

There are a few reasons why we could not initially offer a simple online IDE. Even though we can use a lot of the infrastructure very graciously offered by Ghent University, these resources are not limitless. The extra (interactive) evaluation of student code was something we did not have the resources for, nor did we have any architectural components in place to easily integrate this into Dodona. The main goal of Papyros was thus to provide a client-side Python execution environment we could then include in Dodona. We focused on Python because it is the most widely used programming language in secondary education, at least in Flanders. Note that we do not want to replace Dodona's entire execution model with client-side execution, as the client is an untrusted execution environment where debugging tools could be used to manipulate the results. Because the main idea is integration in Dodona, we primarily wanted users to be able to execute entire programs, and not necessarily offer a REPL at first.

Given that the target audience for Papyros is secondary education students, we identified a number of secondary requirements:

- The editor of our online IDE should have syntax highlighting. Recent literature (Hannebauer et al., 2018) has shown that this does not necessarily have an impact on students' learning, but as the authors point out, it was the prevailing wisdom for a long time that it does help.
- It should also include linting. Linters notify students about syntax errors, but also about style guide violations and anti-patterns.
- Error messages for errors that occur during execution should be user-friendly (Becker et al., 2019).
- Code completion should be available. When starting out with programming, it is hard to remember all the different functions available. Completion frameworks allow students to search for functions, and can show inline documentation for these functions.

4.2.1 Execution

Python can not be executed directly by a browser, since only JavaScript and WebAssembly are natively supported. We investigated a number of solutions for running Python code in the browser.

The first of these is Brython[30]. Brython works by transpiling Python code to JavaScript, where the transpilation is implemented in JavaScript. The project is conceptualized as a way to develop web applications in Python, and not to run arbitrary Python code in the browser, so a lot of its tooling is not directly applicable to our use case, especially concerning interactive input prompts. It also runs on the main thread of the browser, so executing a student's code would freeze the browser until it is done running.

Another solution we looked into is Skulpt[31]. It also transpiles Python code to JavaScript, and supports Python 2 and Python 3.7. After loading Skulpt, a global object is added to the page where Python code can be executed through JavaScript.

[30]`https://brython.info`
[31]`https://skulpt.org`

The final option we looked into was Pyodide[32]. Pyodide was initially developed by Mozilla as part of their Iodide project, aiming to make scientific research shareable and reproducible via the browser. It is now a stand-alone project. Pyodide is a port of the Python interpreter to WebAssembly, allowing it to be executed by the browser. Since the project is focused on scientific research, it has wide support for external libraries such as NumPy. Because Pyodide can be treated as a regular JavaScript library, it can be run in a web worker, making sure that the page stays responsive while the user's code is being executed.

We also looked into integrating other platforms such as Repl.it, but all of them were either not free or did not provide a suitable interface for integration. We chose to base Papyros on Pyodide given its active development, support for recent Python versions and its ability to be executed on a separate thread.

4.2.2 Implementation

There are two aspects to the implementation: the user interface and the technical inner workings. Given that this work will primarily be used by secondary school students, the user interface is an important part of this work that should not be neglected.

User interface

The most important choice in the user interface was the choice of the editor. There were three main options: *i)* Ace[33], *ii)* Monaco[34], and *iii)* CodeMirror[35].

Ace was the editor used by Dodona at the time. It supports syntax highlighting and has some built-in linting. However, it is not very extensible, it does not support mobile devices well, and it's no longer actively developed.

Monaco is the editor extracted from Visual Studio Code and often used by people building full-fledged web IDE's. It also has syntax highlighting and linting and is much more extensible. As with Ace though, support for mobile devices is lacking.

[32]`https://pyodide.org/en/stable`
[33]`https://ace.c9.io/`
[34]`https://microsoft.github.io/monaco-editor/`
[35]`https://codemirror.net/`

CodeMirror is a modern editor made for the web, and not linked to any specific project. It is also extensible and has modular syntax highlighting and linting support. In contrast with Ace and Monaco, it has very good support for mobile devices. Its documentation is also very clear and extensive. Given the clear advantages, we decided to use CodeMirror for Papyros.

The two other main components of Papyros are the output window and the input window. The output window is a simple read-only text area. The input window is a text area that has two modes: interactive mode and batch input. In interactive mode, the user is expected to write the input needed by their program the moment it asks for it (similar to running their program on the command line and answering the prompts when they appear). In batch mode, the user can prefill all the input required by their program.

Inner workings

Since Pyodide does the heavy lifting of executing the actual Python code, most of the implementation work consisted of making Pyodide run in a web worker and hooking up the Python internals to our user interface. The communication between the main UI thread and the web worker happens via message passing. With message passing, all data has to be copied. To avoid having to copy large amounts of data, and to be able to copy actual functions, classes or HTML elements, shared memory can be used. To work correctly with shared memory, synchronization primitives have to be used.

After loading Pyodide, we load a Python script that overwrites certain functions with our versions. For example, base Pyodide will overwrite `input` with a function that calls into JavaScript-land and executes `prompt`. Since we're running Pyodide in a web worker, `prompt` is not available (and we want to implement custom input handling anyway). For `input` we actually run into another problem: `input` is synchronous in Python. In a normal Python environment, `input` will only return a value once the user entered a line of text on the command line. We do not want to edit user code (to make it asynchronous) because that process is error-prone and fragile. So we need a way to make our overwritten version of `input` synchronous as well.

The best way to do this is by using the synchronization primitives of shared memory. We can block on some other thread writing to a certain memory location, and since blocking is synchronous, this makes our `input` synchronous as well. Unfortunately, not all browsers supported shared memory at the time. Other browsers also severely constrain the environment in which shared memory can be used, since a number of CPU side channel attacks related to it were discovered.

Luckily, there is another way we can make the browser perform indefinite synchronous operations from a web worker. Web workers can perform synchronous HTTP requests. We can then intercept these HTTP requests from a service worker. Service workers were originally conceived to allow web applications to continue functioning even when devices go offline. In that case, a service worker could respond to network requests with data it has in its cache. So, putting this together, the web worker tells the main thread that it needs input and then fires off a synchronous HTTP request to some non-existent endpoint. The service worker intercepts this request, and responds to the request once it receives some input from the main thread.

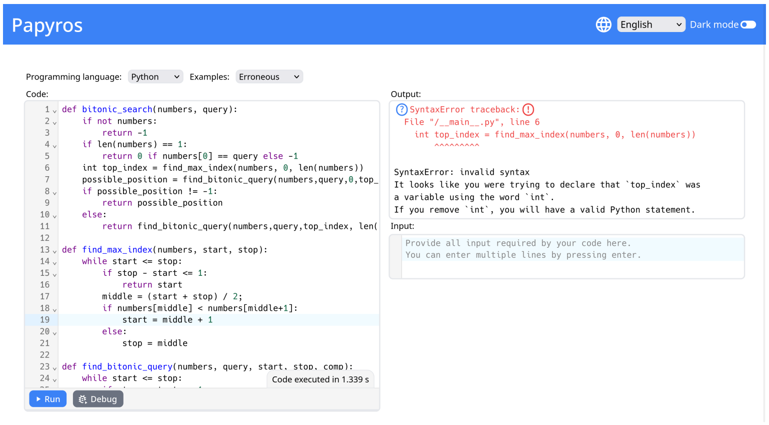
The functionality for performing synchronous communication with the main thread from a web worker was parcelled off into its own library (`sync-message`[36]). This library could then decide which of these two methods to use, depending on the available environment. Another package, `python_runner`[37], bundles all required modifications to the Python environment in Pyodide. This work was done in collaboration with Alex Hall.

Extensions

CodeMirror already has a number of functionalities it supports out of the box such as linting and code completion. It is, however, a pure JavaScript library. This means that these functionalities had to be newly implemented, since the standard tooling for Python is almost entirely implemented in Python. Fortunately CodeMirror also supports supplying one's own linting message and code completion. Since we have a working Python environment, we can also use it to run the standard Python tools for linting (Pylint) and code completion (Jedi) and hook up their results to CodeMirror. For code completion this has the added benefit of also showing the documentation for the autocompleted items, which is especially useful for people new to programming (which is exactly our target audience).

[36] `https://github.com/alexmojaki/sync-message`
[37] `https://github.com/alexmojaki/python_runner`

Usability was further improved by adding the `FriendlyTraceback` library. `FriendlyTraceback` is a Python library that changes error messages in Python to be clearer to beginners, by explicitly answering questions such as where and why an error occurred. An example of what this looks like can be seen in Figure 4.5



English

Programming language: Python   Examples: Erroneous

```
def bitonic_search(numbers, query):
    if not numbers:
        return -1
    if len(numbers) == 1:
        return 0 if numbers[0] == query else -1
    int top_index = find_max_index(numbers, 0, len(numbers))
    possible_position = find_bitonic_query(numbers,query,0,top_
    if possible_position != -1:
        return possible_position
    else:
        return find_bitonic_query(numbers,query,top_index, len(

def find_max_index(numbers, start, stop):
    while start <= stop:
        if stop - start <= 1:
            return start
        middle = (start + stop) / 2;
        if numbers[middle] < numbers[middle+1]:
            start = middle + 1
        else:
            stop = middle

def find_bitonic_query(numbers, query, start, stop, comp):
    while start <= stop:
```

Code executed in 1.339 s

```
SyntaxError traceback:
  File "/__main__.py", line 6
    int top_index = find_max_index(numbers, 0, len(numbers))
        ^^^^^^^^^


SyntaxError: invalid syntax
It looks like you were trying to declare that `top_index` was
a variable using the word `int`.
If you remove `int`, you will have a valid Python statement.
```

```
Provide all input required by your code here.
You can enter multiple lines by pressing enter.
```

Figure 4.5: Papyros execution where a student tried to add a type declaration to a variable, which `FriendlyTraceback` shows a fitting error message for.

4.3 R judge

Because Dodona had proven itself as a useful tool for teaching Python and Java to students, colleagues teaching statistics started asking if we could build R support into Dodona. We started working on an R judge[38] soon after. By now, more than 1 250 R exercises have been added, and almost 1 million submissions have been made to an R exercise.

Because R is the *lingua franca* of statistics, there are a few extra features that come to mind that are not typically handled by judges, such as handling of data frames and outputting visual graphs (or even evaluating that a graph was built correctly). Another feature that teachers wanted that we had not built into a judge previously was support for inspecting

[38]`https://github.com/dodona-edu/judge-r`

the student's source code, e.g. for making sure that certain functions were or were not used.

4.3.1 Exercise API

The API for the R judge was designed to follow the visual structure of the feedback shown as closely as possible, as can be seen in the sample evaluation code in Listing 4.1. Tabs are represented by different evaluation files. In addition to the `testEqual` function demonstrated in Listing 4.1 there are some other functions to specifically support the requested functionality. `testImage` will set up some handlers in the R environment so that generated plots (or other images) are sent as feedback (in a base-64 encoded string) instead of the filesystem. It will also by default make the test fail if no image was generated (but does not do any verification of the image contents). An example of what the feedback looks like when an image is generated can be seen in Figure 4.6. `testDF` has some extra functionality for testing the equality of data frames, where it is possible to ignore row and column order. The generated feedback is also limited to 5 lines of output, to avoid overwhelming students (and their browsers) with the entire table. `testGGPlot` can be used to introspect plots generated with GGPlot (Wickham et al., 2023). To test whether students use certain functions, `testFunctionUsed` and `testFunctionUsedInVar` can be used. The latter tests whether the specific function is used when initializing a specific variable.

If some code needs to be executed in the student's environment before the student's code is run (e.g. to make some dataset available, or to fix a random seed), the `preExec` argument of the `context` function can be used to do so.

4.3.2 Security

Other than the API for teachers creating exercises, encapsulation of student code is also an important part of a judge. Students should not be able to access functions defined by the judge, or be able to find the correct solution or the evaluating code. The R judge makes sure of this by making extensive use of environments. This is also reflected in the teacher API: they can access variables or execute functions in the student environment, but this environment has to be explicitly passed to the function generating the student result. In R, all environments except the root environment

```r
context({
  testcase('The correct method was used', {
    testEqual("test$alternative",
              function(studentEnv) {
                studentEnv$test$alternative
              },
              'two.sided')
    testEqual("test$method",
              function(studentEnv) {
                studentEnv$test$method
              },
              ' Two Sample t-test')
  })
  testcase('p value is correct', {
    testEqual("test$p.value",
              function(studentEnv) {
                studentEnv$test$p.value
              },
              0.175)
  })
}, preExec = {
  set.seed(20190322)
})
```

Listing 4.1: Sample evaluation code for a simple R exercise. The feedback will contain one context with two test cases in it. The first test case checks whether some t-test was performed correctly, and does this by performing two equality checks. The second test case checks that the p-value calculated by the t-test is correct. The `preExec` is executed in the student's environment and here fixes a random seed for the student's execution.

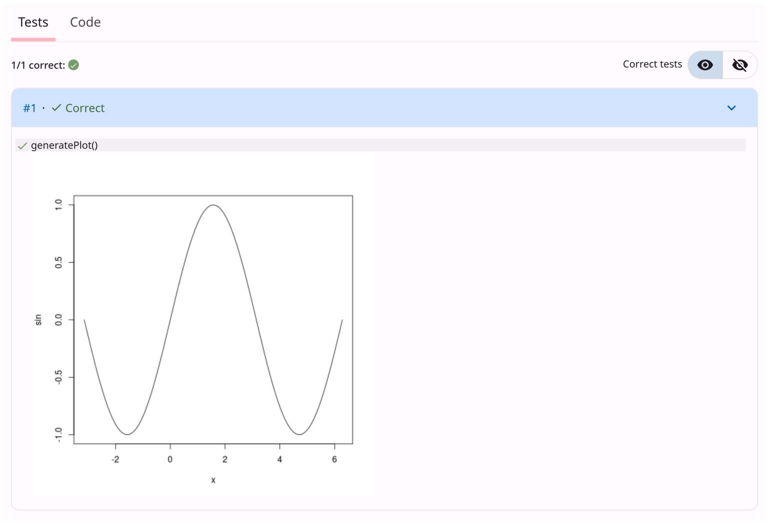


Figure 4.6: Feedback for an R exercise where the goal is to generate a plot. The code generates a plot showing a simple sine function, which is reflected in the feedback.

have a parent, essentially creating a tree structure of environments. In most cases, this tree will actually be a path, but in the R judge, the student environment is explicitly attached to the base environment. This even makes sure that libraries loaded by the judge are not initially available to the student code (thus allowing teachers to test that students can correctly load libraries). The judge itself runs in an anonymous environment, so that even students with intimate knowledge of the inner workings of R and the judge itself would not be able to find it.

The judge is also programmed very defensively. Every time execution is handed off to student code (or even teacher code), appropriate error handlers and output redirections are installed. This prevents the student and teacher code from e.g. writing to standard output (and thus messing up the JSON expected by Dodona).

4.4 TESTed

TESTed[39] is a universal judge for Dodona. TESTed was developed to solve two major drawbacks with the current judge system of Dodona:

- When creating the same exercise in multiple programming languages, the exercise description and test cases need to be redone for every programming language. This is especially relevant for very simple exercises that students almost always start with, and for exercises in algorithms courses, where the programming language a student solves an exercise in is of lesser importance than the way they solve it. Mistakes in exercises also have to be fixed in all instances of the exercise when there are multiple instances of the exercise.
- The judges themselves have to be created from scratch every time. Most judges offer the same basic concepts and features, most of which are independent of programming language (communication with Dodona, checking correctness, I/O, …).

The goal of TESTed was to implement a judge so that programming exercises only have to be created once to be available in all programming languages TESTed supports. TESTed currently supports Bash, C, C#, Haskell, Java, JavaScript, Kotlin, and Python. An exercise should also not have to be changed when support for a new programming language is added. Adding a new programming language itself should also be easier than creating a new judge from scratch. As a secondary goal, we also wanted to make it as easy as possible to create new exercises. Teachers who have not used Dodona before should be able to create a new basic exercise without too many issues.

TESTed was first developed as a proof of concept in my master's thesis (Van Petegem & Dawyndt, 2018), which presented a method for estimating the time and memory complexity of solutions for programming exercises. One of the goals was to make this method work over many programming languages. To do this, we wrote a framework based on Jupyter kernels[40] where the interaction with each programming language was abstracted away behind a common interface. We realized this framework could be useful in itself, but it was only developed as far as we needed for the thesis. Further work then developed this proof of concept into the full judge we will present in the following sections.

[39] `https://github.com/dodona-edu/universal-judge`
[40] `https://jupyter.org`

We will expand on TESTed using an example exercise. In this exercise, students need to rotate a list. For example, in Python, `rotate([0, 1, 2, 3, 4], 2)` should return the list `[3, 4, 0, 1, 2]`. The goal is that teachers can write their exercises as in Listing 4.2.

```yaml
- tab: "Feedback"
  contexts:
    - testcases:
        - statement: "numbers01 = [0, 1, 2, 3, 4]"
        - expression: "rotate(numbers01, 2)"
          return: [3, 4, 0, 1, 2]
        - expression: "rotate(numbers01, 1)"
          return: [4, 0, 1, 2, 3]
    - testcases:
        - statement: "numbers02 = [0, 1, 2, 3, 4, 5]"
        - expression: "rotate(numbers02, 2)"
          return: [4, 5, 0, 1, 2, 3]
        - expression: "rotate(numbers02, 1)"
          return: [5, 0, 1, 2, 3, 4]
```

Listing 4.2: Example of a TESTed test plan, showing statements and expressions. Statements and expressions are in a custom Python-like language.

4.4.1 Overview

TESTed generally works using the following steps:

1. Receive the submission, exercise test plan, and any auxiliary files from Dodona.
2. Validate the test plan and making sure the submission's programming language is supported for the given exercise.
3. Generate test code for each context in the test plan.
4. Optionally compile the test code, either in batch mode or per context. This step is skipped if evaluation a submission written in an interpreted language.
5. Execute the test code.
6. Evaluate the results, either with programming language-specific evaluation, programmed evaluation, or generic evaluation.

7. Send the evaluation results to Dodona.

4.4.2 Test plan

One of the most important elements that is needed to perform these steps is the test plan. This test plan is a hierarchical structure, which closely resembles the underlying structure of Dodona's feedback. There are, however, a few important differences. The first of these is the *context testcase*. This is a special testcase per context that executes the main function (or the entire program in case this is more appropriate for the language being executed). The only possible inputs for this testcase are text for the standard input stream, command-line arguments and files in the working directory. The exit status code can only be checked in this testcase as well.

Like the communication with Dodona, this test plan is a JSON document under the hood. In the following sections, we will use the JSON representation of the test plan to discuss how TESTed works. Exercise authors use the DSL to write their tests, which we discuss in Section 4.4.9. This DSL is internally converted by TESTed to the more extensive underlying structure before execution. A test plan of the example exercise can be seen in Listing 4.3.

4.4.3 Data serialization

As part of the test plan, we also need a way to generically describe values and their types. This is what we will call the *serialization format*. The serialization format should be able to represent all the basic data types we want to support in the programming language independent part of the test plan. These data types are basic primitives like integers, reals (floating point numbers), booleans, and strings, but also more complex collection types like arrays (or lists), sets and mapping types (maps, dictionaries, and objects). Note that the serialization format is also used on the side of the programming language, to receive (function) arguments and send back execution results.

Of course, a number of data serialization formats already exist, like `MessagePack`[41], `ProtoBuf`[42], … Binary formats were excluded from the start, because they can not easily be embedded in our JSON test plan,

[41]`https://msgpack.org/`
[42]`https://protobuf.dev/`

```
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "testcases": [
            {
              "input": {
                "type": "function",
                "name": "rotate",
                "arguments": [
                  ...
                ]
              },
              "output": {
                "result": {
                  "value": {
                    ...
                  }
                }
              }
            },
            ...
          ]
        }
      ]
    }
  ]
}
```

Listing 4.3: Basic structure of a test plan. The structure of Dodona's feedback is followed closely. The function arguments have been left out, as they are explained in Section 4.4.3.

but more importantly, they can neither be written nor read by humans. Other formats did not support all the types we wanted to support and could not be extended to do so. Because of our goal in supporting many programming languages, the format also had to be either widely implemented or be easily implementable. None of the formats we investigated met all these requirements. We opted to make the serialization format in JSON as well. Values are represented by objects containing the encoded value and the accompanying type. Note that this is a recursive format: the values in a collection are also serialized according to this specification.

The types of values are split in three categories. The first category are the basic types listed above. The second category are the advanced types. These are specialized versions of the basic types, for example to specify the number of bits that a number should be, or whether a collection should be a tuple or a list. The final category of types can only be used to specify an expected type. In addition to the other categories, `any` can be specified. Like the name says, `any` signifies that the expected type is unknown, and the student can therefore return any type.

The encoded expected return value of our example exercise can be seen in Listing 4.4.

```json
{
  "type": "sequence",
  "data": [
    { "type": "integer", "data": 3 },
    { "type": "integer", "data": 4 },
    { "type": "integer", "data": 0 },
    { "type": "integer", "data": 1 },
    { "type": "integer", "data": 2 }
  ]
}
```

Listing 4.4: A list encoded using TESTed's data serialization format. The corresponding Python list would be `[3, 4, 0, 1, 2]`.

4.4.4 Statements

There is more complexity hidden in the idea of creating a variable of a custom type. It implies that we need to be able to create variables, instead of just capturing the result of function calls or other expressions.

To support this, specific structures were added to the test plan JSON schema. Listing 4.5 shows what it would look like if we wanted to assign the function argument of our example exercise to a variable.

```json
"testcases": [
  {
    "input": {
      "type": "sequence",
      "variable": "numbers01",
      "expression": {
        "type": "sequence",
        "data": [
          { "type": "integer", "data": 0 },
          { "type": "integer", "data": 1 },
          { "type": "integer", "data": 2 },
          { "type": "integer", "data": 3 },
          { "type": "integer", "data": 4 }
        ],
      }
    }
  }
]
```

Listing 4.5: A TESTed testcase containing a statement. The corresponding Python statement would be `numbers01 = [0, 1, 2, 3, 4]`.

4.4.5 Checking programming language support

We also need to make sure that the programming language of the submission under test is supported by the test plan of its exercise. The two things that are checked are whether a programming language supports all the types that are used and whether the language has all the necessary language constructs. For example, if the test plan uses a `tuple`, but the language does not support it, it's obviously not possible to evaluate a submission in that language. The same is true for overloaded functions: if it is necessary that a function can be called with a string and with a number, a language like C will not be able to support this. Collections also are not yet supported for C, since the way arrays and their lengths work in C is quite different from other languages. Our example exercise will not work in C for this reason.

4.4.6 Execution

To go from the generic test plan to something that can actually be executed in the given language, we need to generate test code. This is done by way of a templating system. For each programming language supported by TESTed, a few templates need to be defined. The serialization format also needs to be implemented in the given programming language. Because the serialization format is based on JSON and JSON is a widely used format, this requirement is usually pretty easy to fulfil.

For some languages, the code needs to be compiled as well. All test code is usually compiled into one executable, since this only results in one call to the compiler (which is usually a pretty slow process). There is one big drawback to this way of compiling code: if there is a compilation error (for example because a student has not yet implemented all requested functions) the compilation will fail for all contexts. Because of this, TESTed will fall back to separate compilations for each context if a compilation error occurs. Subsequently, the test code is executed and its results collected.

4.4.7 Evaluation

The generated output is usually evaluated by TESTed itself. TESTed can however only evaluate the output as far as it is programmed to do so. There are two other ways the results can be evaluated: programmed evaluation and programming-language specific evaluation. With programmed evaluation, the results are passed to code written by a teacher. For efficiency's sake, this code has to be written in Python (which means TESTed does not need to launch a new process for the evaluation). This code will then check the results, and generate appropriate feedback. Programming-language specific evaluation is executed immediately after the test code in the process of the test code. This can be used to evaluate programming-language specific concepts, for example the correct use of pointers in C.

4.4.8 Linting

Next to correctness, style is also an important aspect of source code. In a lot of contexts, linters are used to perform basic style checks. Linting was also implemented in TESTed. For each supported programming language, both the linter to be used and how its output should be interpreted are specified.

4.4.9 DSL

As mentioned in Section 4.4.2, exercise authors are not expected to write their test plans in JSON. It is very verbose and error-prone when writing (trailing commas are not allowed, all object keys are strings and need to be written as such, etc.). This aspect of usability was not the initial focus of TESTed, since most Dodona power users already use code to generate their test plans. Because code is very good at outputting an exact and verbose format like JSON, this avoids its main drawback. However, we wanted teachers in secondary education to be able to work with TESTed, and they mostly do not have enough experience with programming themselves to generate a test plan. To solve this problem we wanted to integrate a domain-specific language (DSL) to describe TESTed test plans.

We first investigated whether we could use an existing format to do so. The best option of these was PEML: the Programming Exercise Markup Language (Mishra & Edwards, 2023). Envisioned as a universal format for programming exercise descriptions, their goals seemed to align with ours. Unfortunately, they did not base themselves on any existing formats. This means that there is little tooling around PEML. Parsing it as part of TESTed would require a lot of implementation work, and IDEs or other editors do not do syntax highlighting for it. The format itself is also quite error-prone when writing. Because of these reasons, we discarded PEML and started working on our own DSL.

Our own DSL is based on YAML[43]. YAML is a superset of JSON and describes itself as "a human-friendly data serialization language for all programming languages". The DSL structure is quite similar to the actual test plan, though it does limit the amount of repetition required for common operations. YAML's concise nature also contributes to the read- and writability of its test plans.

For the actual statements, expressions and values, we added an abstract programming language, made to look somewhat like Python 3. Note that this is not a full programming language, but only supports language constructs as far as they are needed by TESTed. Values are interpreted as basic types, but can be cast explicitly to one of the more advanced types. The DSL version of the test plan for the example exercise can be seen in Listing 4.2.

[43]`https://yaml.org`

4.5 Conclusion

This chapter shows that building a platform like Dodona takes a lot of work. No software is ever free of bugs, and we have to keep up with dependency updates. Expanding into secondary education also meant catering to their needs, by building a basic in-browser IDE. The infrastructure and tooling required for supporting the assessment of many submissions in a lot of different programming languages is also considerable.

5 Pass/fail prediction in programming courses

We now shift to the chapters where we make use of the data provided by Dodona to perform educational data mining research.

This chapter is based on **Van Petegem, C.**, Deconinck, L., Mourisse, D., Maertens, R., Strijbol, N., Dhoedt, B., De Wever, B., Dawyndt, P., Mesuere, B., 2022. Pass/Fail Prediction in Programming Courses. *Journal of Educational Computing Research*, 68–95. It also briefly discusses the work reproduction of this research performed in Zhidkikh, D., Heilala, V., **Van Petegem, C.**, Dawyndt, P., Järvinen, M., Viitanen, S., De Wever, B., Mesuere, B., Lappalainen, V., Kettunen, L., & Hämäläinen, R., 2024. Reproducing Predictive Learning Analytics in CS1: Toward Generalizable and Explainable Models for Enhancing Student Retention. *Journal of Learning Analytics*, 1-21.

The work presented in this chapter was part of the master thesis by Louise Deconinck, with the reproduction being led by Denis Zhidkikh.

5.1 Introduction

A lot of educational opportunities are missed by keeping assessment separate from learning (Black & Wiliam, 1998; Wiliam, 2011). Educational technology can bridge this divide by providing real-time data and feedback to help students learn better, teachers teach better, and educational systems become more effective (OECD, 2021). Earlier research demonstrated that the adoption of interactive platforms may lead to better learning outcomes (Khalifa & Lam, 2002) and allows collecting rich data on student behaviour throughout the learning process in non-evasive ways. Effectively using such data to extract knowledge and further improve the underlying processes, which is called educational data mining (Baker & Yacef, 2009), is increasingly explored as a way to enhance learning and educational processes (Dutt et al., 2017).

About one third of the students enrolled in introductory programming courses fail (Bennedsen & Caspersen, 2007; Watson & Li, 2014). Such high failure rates are problematic in light of low enrolment numbers and high industrial demand for software engineering and data science profiles (Watson & Li, 2014). To remedy this situation, it is important to have detection systems for monitoring at-risk students, understand why they are failing, and develop preventive strategies. Ideally, detection happens early on in the learning process to leave room for timely feedback and interventions that can help students increase their chances of passing a course.

Previous approaches for predicting performance on examinations either take into account prior knowledge such as educational history and socio-economic background of students or require extensive tracking of student behaviour. Extensive behaviour tracking may directly impact the learning process itself. Rountree et al. (2004) used decision trees to find that the chance of failure strongly correlates with a combination of academic background, mathematical background, age, year of study, and expectation of a grade other than “A”. They conclude that students with a skewed view on workload and content are more likely to fail. Kovacic (2012) used data mining techniques and logistic regression on enrolment data to conclude that ethnicity and curriculum are the most important factors for predicting student success. They were able to predict success with 60% accuracy. Asif et al. (2017) combine examination results from the last two years in high school and the first two years in higher education to predict student performance in the remaining two years of their academic study program. They used data from one cohort to train models and from another cohort to test that the accuracy of their predictions is about 80%. This evaluates their models in a similar scenario in which they could be applied in practice.

A downside of the previous studies is that collecting uniform and complete data on student enrolment, educational history and socio-economic background is impractical for use in educational practice. Data collection is time-consuming and the data itself can be considered privacy-sensitive. Usability of predictive models therefore not only depends on their accuracy, but also on their dependency on findable, accessible, interoperable and reusable data (Wilkinson et al., 2016). Predictions based on educational history and socio-economic background also raise ethical concerns. Such background information definitely does not explain everything and lowers the perceived fairness of predictions (Binns et al., 2018; Grgić-Hlača et al., 2018). Students also can not change their background, so these items are not actionable for any corrective intervention.

It might be more convenient and acceptable if predictive models are restricted to data collected on student behaviour during the learning process of a single course. An example of such an approach comes from Vihavainen (2013), using snapshots of source code written by students to capture their work attitude. Students are actively monitored while writing source code and a snapshot is taken automatically each time they edit a document. These snapshots undergo static and dynamic analysis to detect good practices and code smells, which are fed as features to a non-parametric Bayesian network classifier whose pass/fail predictions are 78% accurate by the end of the semester. In a follow-up study they applied the same data and classifier to accurately predict learning outcomes for the same student cohort in another course (Vihavainen et al., 2013). In this case, their predictions were 98.1% accurate, although the sample size was rather small. While this procedure does not rely on external background information, it has the drawback that data collection is more invasive and directly intervenes with the learning process. Students can not work in their preferred programming environment and have to agree with extensive behaviour tracking.

Approaches that are not using machine learning also exist. Feldman et al. (2019) try to answer the question "Am I on the right track?" on the level of individual exercises, by checking if the student's current progress can be used as a base to synthesize a correct program. However, there is no clear way to transform this type of approach into an estimation of success on examinations. Werth (1986) found significant ($p < 0.05$) correlations between students' college grades, the number of hours worked, the number of high school mathematics classes and the students' grades for an introductory programming course. Goold & Rimmer (2000) also looked at learning style (surveyed using LSI2) as a factor in addition to demographics, academic ability, problem-solving ability and indicators of personal motivation. The regressions in their study account for 42 to 65 percent of the variation in cohort performances.

In this chapter, we present an alternative framework (Figure 5.1) to predict if students will pass or fail a course within the same context of learning to code. The method only relies on submission behaviour for programming exercises to make accurate predictions and does not require any prior knowledge or intrusive behaviour tracking. Interpretability of the resulting models was an important design goal to enable further investigation on learning habits. We also focused on early detection of at-risk students, because predictive models are only effective for the cohort under investigation if remedial actions can be started long before students take their final exam.

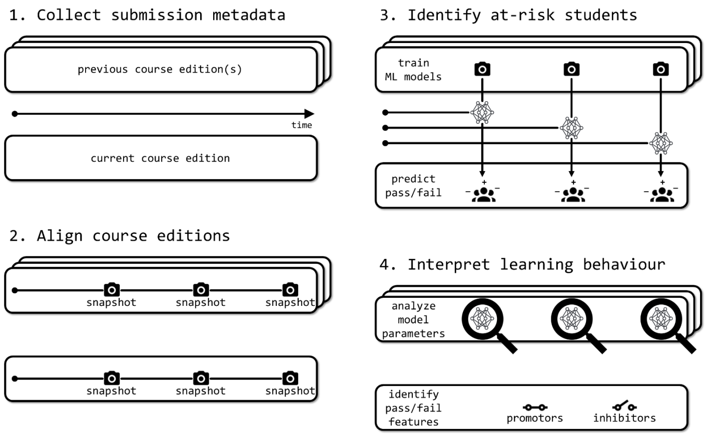


Figure 5.1: Step-by-step process of the proposed pass/fail prediction framework for programming courses: 1) Collect metadata from student submissions during successive course editions. 2) Align course editions by identifying corresponding time points and calculating snapshots at these time points. A snapshot measures student performance only from metadata available in the course edition at the time the snapshot was taken. 3) Train a machine learning model on snapshot data from previous course editions and predict which students will likely pass or fail the current course edition by applying the model on a snapshot of the current edition. 4) Infer what learning behaviour has a positive or negative learning effect by interpreting feature weights of the machine learning model. Teachers can use insights from both steps 3 and 4 to take actions in their teaching practice.

The chapter starts with a description of how data is collected, what data is used and which machine learning methods have been evaluated to make pass/fail predictions. We evaluated the same models and features in multiple courses to test their robustness against differences in teaching styles and student backgrounds. The results are discussed from a methodological and educational perspective with a focus on *i)* accuracy (What machine learning algorithms yield the best predictions?), *ii)* early detection (Can we already make accurate predictions early on in the semester?), and *iii)* interpretability (Are resulting models clear about which features are important? Can we explain why certain features are identified as important? How self-evident are important features?).

5.2 Materials and methods

5.2.1 Course structures

This study uses data from two introductory programming courses, referenced as course A and course B, collected during 3 editions of each course in academic years 2016–2017, 2017–2018, and 2018–2019. Course A is the course described in Section 3.2. Course B is the introductory programming course taught at the Faculty of Engineering at Ghent University. Both courses run once per academic year across a 12-week semester (September–December). They have separate lecturers and teaching assistants, and are taken by students of different faculties. The courses have their own structure, but each edition of a course follows the same structure. Table 5.1 summarizes some statistics on the course editions included in this study.

	year	students	# ex.	solutions	tries	pass rate
A	2016–2017	322	60	167 675	9.56	60.86%
A	2017–2018	249	60	125 920	9.19	61.44%
A	2018–2019	307	60	176 535	10.29	65.14%
B	2016–2017	372	138	371 891	9.10	56.72%
B	2017–2018	393	187	407 696	7.31	60.81%
B	2018–2019	437	201	421 461	6.26	62.47%

Table 5.1: Statistics for course editions included in this study. The courses are taken by different student cohorts at different faculties and differ in structure, lecturers and teaching assistants. The number of tries is the average number of solutions submitted by a student per exercise they worked on (i.e. for which the student submitted at least one solution in the course edition).

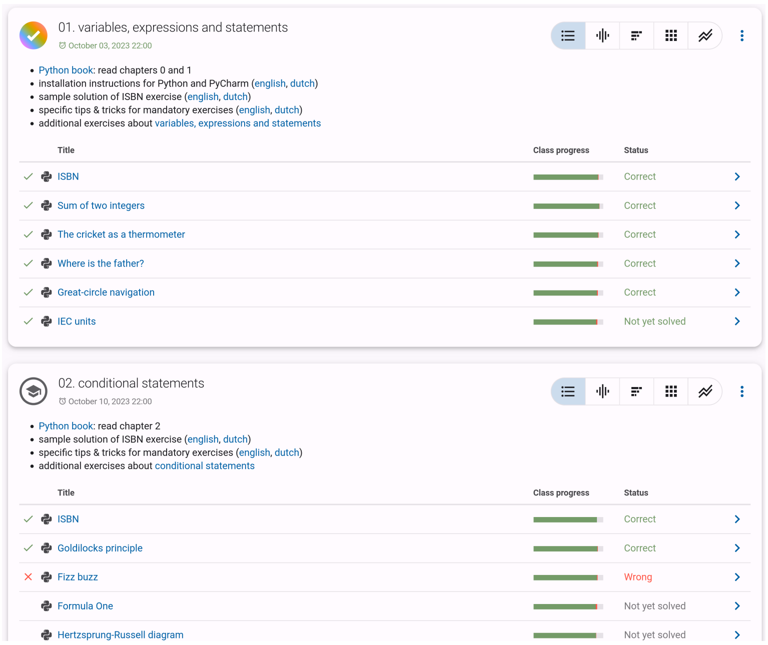
Course A is subdivided into two successive instructional units that each cover five programming topics – one topic per week – followed by an evaluation about all topics covered in the unit. Students must solve six programming exercises on each topic before a deadline one week later. Submitted solutions for these mandatory exercises are automatically evaluated and considered correct if they pass all unit tests for the exercise. Failing to submit a correct solution for a mandatory exercise has a small impact on the score for the evaluation at the end of the unit. The final exam at the end of the semester evaluates all topics covered in the entire course. Students need to solve new programming exercises during evaluations (2 exercises) and exams (3 exercises), where reviewers manually evaluate and grade submitted solutions based on correctness, programming style used, choice made between the use of different programming techniques, and the overall quality of the solution. Each edition of the course is taken by about 300 students.

Course B has 20 lab sessions across the semester, with evaluations after the 10th and 17th lab session and a final exam at the end of the semester. Each lab session comes with a set of exercises and has an indicative deadline for submitting solutions. However, these exercises are not taken into account when computing the final score for the course, so students are completely free to work on exercises as a way to practice their coding skills. Students need to solve new programming exercises during evaluations (3 exercises) and exams (4 exercises). Solutions submitted during evaluations are automatically graded based on the number of passed unit tests for the exercise. Solutions submitted during exams are manually graded in the same way as for course A. Each edition of the course is taken by about 400 students.

We opted to use two different courses that are structured quite differently to make sure our framework is generally applicable in other courses where the same behavioural data can be collected.

5.2.2 Learning environment

Both courses use Dodona as their online learning environment (Van Petegem et al., 2023). Dodona promotes active learning through problem-solving (Prince, 2004). Each course edition has its own Dodona course, with a learning path that groups exercises in separate series (Figure 5.2). Course A has one series per covered programming topic (10 series in total) and course B has one series per lab session (20 series in total). A submission deadline is set for each series. Dodona is also used to take tests and exams, within series that are only accessible for participating students.


01. variables, expressions and statements
October 03, 2023 22:00

- Python book: read chapters 0 and 1
- installation instructions for Python and PyCharm (english, dutch)
- sample solution of ISBN exercise (english, dutch)
- specific tips & tricks for mandatory exercises (english, dutch)
- additional exercises about variables, expressions and statements

	Title	Class progress	Status
✓	ISBN		Correct
✓	Sum of two integers		Correct
✓	The cricket as a thermometer		Correct
✓	Where is the father?		Correct
✓	Great-circle navigation		Correct
✓	IEC units		Not yet solved

02. conditional statements
October 10, 2023 22:00

- Python book: read chapter 2
- sample solution of ISBN exercise (english, dutch)
- specific tips & tricks for mandatory exercises (english, dutch)
- additional exercises about conditional statements

	Title	Class progress	Status
✓	ISBN		Correct
✓	Goldilocks principle		Correct
✗	Fizz buzz		Wrong
	Formula One		Not yet solved
	Hertzsprung-Russell diagram		Not yet solved

Figure 5.2: Student view of a course in Dodona, showing two series of six exercises in the learning path of course A. Each series has its own deadline. The status column shows a global status for each exercise based on the last solution submitted. The class progress column visualizes global status for each exercise for all students subscribed in the course. Icons on the left show a global status for each exercise based on the last submission submitted before the series deadline.

Throughout an edition of a course, students can continuously submit solutions for programming exercises and immediately receive feedback upon each submission, even during tests and exams. This rich feedback is automatically generated by an online judge and unit tests linked to each exercise (Wasik et al., 2018). Guided by that feedback, students can track potential errors in their code, remedy them and submit an updated solution. There is no restriction on the number of solutions that can be submitted per exercise, and students can continue to submit solutions after a series deadline. All submitted solutions are stored, but only the last submission before the deadline is taken into account to determine the status (and grade) of an exercise for a student. One of the effects of active learning, triggered by exercises with deadlines and automated feedback, is that most learning happens during the semester as can be seen on the heatmap in Figure 5.3.

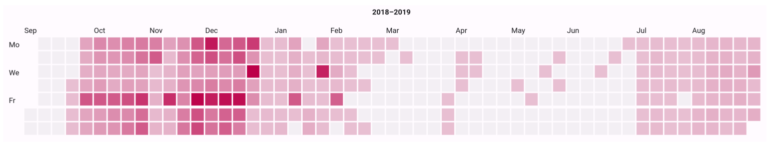


Figure 5.3: Heatmap showing the distribution per day of all 176 535 solutions submitted during the 2018–2019 edition of course A. The darker the colour, the more submissions were made on that day. A lighter red means there are few submissions on that day. A light grey square means that no submissions were made that day. Weekly lab sessions for different groups were organized on Monday afternoon, Friday morning and Friday afternoon. Weekly deadlines for mandatory exercises were on Tuesdays at 22:00. There were four exam sessions for different groups in January. There is little activity in the exam periods, except for days on which there was an exam. The course is not taught in the second semester, so there is very little activity there. Two exam sessions were organized in August/September granting an extra chance to students who failed on their exam in January/February.

5.2.3 Submission data

We exported data from Dodona on all solutions submitted by students during each course edition included in the study. Each solution has a submission timestamp with precision down to the second and is linked to a course edition, series in the learning path, exercise and student. We did not use the actual source code submitted by students, but did use the status describing the global assessment made by the learning environment: correct, wrong, compilation error, runtime error, time limit exceeded, memory limit exceeded, or output limit exceeded.

Comparison of student behaviour between different editions of the same course is enabled by computing snapshots for each edition at series deadlines. Because course editions follow the same structure, we can align their series and compare snapshots for corresponding series. Corresponding snapshots represent student performance at intermediate points during the semester and their chronology also allows longitudinal analysis within the semester. Course A has snapshots for the five series of the first unit (labelled S1–S5), a snapshot for the evaluation of the first unit (labelled E1), snapshots for the five series of the second unit (labelled S6–S10), a snapshot for the evaluation of the second unit (labelled E2) and a snapshot for the exam (labelled E3). Course B has snapshots for the first ten lab sessions (labelled S1–S10), a snapshot for the first evaluation (labelled E1), snapshots for the next series of seven lab sessions (labelled S11–S17), a snapshot for the second evaluation (labelled E2), snapshots for the last three lab sessions (S18–S20) and a snapshot for the exam (labelled E3).

It is important to stress that a snapshot of a course edition measures student performance only using the information available at the time of the snapshot. As a result, the snapshot does not take into account submissions after its timestamp. The behaviour of a student can then be expressed as a set of features extracted from the raw submission data. We identified different types of features (see Appendix B) that indirectly quantify certain behavioural aspects of students practising their programming skills. When and how long do students work on their exercises? Can students correctly solve an exercise and how much feedback do they need to accomplish this? What kinds of mistakes do students make while solving programming exercises? Do students further optimize the quality of their solution after it passes all unit tests, based on automated feedback or publication of sample solutions? Note that there is no one-on-one relationship between these behavioural aspects and feature types. Some aspects will be covered by multiple feature types, and some feature types incorporate multiple behavioural aspects. We will therefore need to take into account possible dependencies between feature types while making predictions.

A feature type essentially makes one observation per student per series. Each feature type thus results in multiple features: one for each series in the course (excluding series for evaluations and exams). In addition, the snapshot also contains a feature for the average of each feature type across all series. We do not use observations per individual exercise, as the actual exercises might differ between course editions. Snapshots taken at the deadline of an evaluation or later, also contain the score a student obtained for the evaluation. These features of the snapshot can be used to predict whether a student will finally pass/fail the course. In addition, the snapshot also contains a label indicating whether the student passed or failed that is used during training and testing of classification algorithms. Students that did not take part in the final examination, automatically fail the course.

Since course B has no hard deadlines, we left out deadline-related features from its snapshots (`first_dl`, `last_dl` and `nr_dl`; see Appendix B). To investigate the impact of deadline-related features, we also made predictions for course A that ignore these features.

5.2.4 Classification algorithms

We evaluated four classification algorithms to make pass/fail predictions from student behaviour: stochastic gradient descent (Ferguson, 1982), logistic regression (Kleinbaum, 1994), support vector machines (Cortes & Vapnik, 1995), and random forests (Svetnik et al., 2003). We used implementations of these algorithms from `scikit-learn` (Pedregosa et al., 2011) and optimized model parameters for each algorithm by cross-validated grid-search over a parameter grid.

Readers unfamiliar with machine learning can think of these specific algorithms as black boxes, but we briefly explain the basic principles of classification for their understanding. Supervised learning algorithms use a dataset that contains both inputs and desired outputs to build a model that can be used to predict the output associated with new inputs. The dataset used to build the model is called the training set and consists of training examples, with each example represented as an array of input values (feature vector). Classification is a specific case of supervised learning where the outputs are restricted to a limited set of values (labels), in contrast to for example all possible numerical values with a range. Classification algorithms are validated by splitting a dataset of labelled feature vectors into a training set and a test set, building a model from the training set, and evaluating the accuracy of its predictions on the test set. Keeping training and test data separate is crucial to avoid bias during validation. A standard method to make unbiased predictions for all examples in a dataset is k-fold cross-validation: partition the dataset in k subsets and then perform k experiments that each take one subset for evaluation and the other $k-1$ subsets for training the model.

Pass/fail prediction is a binary classification problem with two possible outputs: passing or failing a course. We evaluated the accuracy of the predictions for each snapshot and each classification algorithm with three different types of training sets. As we have data from three editions of each course, the largest possible training set to make predictions for the snapshot of a course edition combines the corresponding snapshots from the two remaining course editions. We also made predictions for a snapshot using each of its corresponding snapshots as individual training sets to see if we can still make accurate predictions based on data from only one other course edition. Finally, we also made predictions for a snapshot using 5-fold cross-validation to compare the quality of predictions based on data from the same or another cohort of students. Note that the latter strategy is not applicable to make predictions in practice, because we will not have pass/fail results as training labels while taking snapshots during the semester. In practice, to make predictions for a snapshot, we can rely only on corresponding snapshots from previous course editions. However, because we can assume that different editions of the same course yield independent data, we also used snapshots from future course editions in our experiments.

There are many metrics that can be used to evaluate how accurately a classifier predicted which students will pass or fail the course from the data in a given snapshot. Predicting a student will pass the course is called a positive prediction, and predicting they will fail the course is called a negative prediction. Predictions that correspond with the actual outcome are called true predictions, and predictions that differ from the actual outcome are called false predictions. This results in four possible combinations of predictions: true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). Two standard accuracy metrics used in information retrieval are precision (Equation 5.1) and recall (Equation 5.2). The latter is also called sensitivity if used in combination with specificity (Equation 5.3).

$$\frac{TP}{TP+FP} \tag{5.1}$$

$$\frac{TP}{TP+FN} \tag{5.2}$$

$$\frac{TN}{TN + FP} \tag{5.3}$$

Many studies for pass/fail prediction use accuracy (Equation 5.4) as a single performance metric. However, this can yield misleading results. For example, let's take a dummy classifier that always "predicts" students will pass, no matter what. This is clearly a bad classifier, but it will nonetheless have an accuracy of 75% for a course where 75% of the students pass.

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{5.4}$$

In our study, we will therefore use two more complex metrics that take these effects into account: balanced accuracy and F_1-score. Balanced accuracy is the average of sensitivity and specificity. The F_1-score is the harmonic mean of precision and recall. If we go back to our example, the optimistic classifier that consistently predicts that all students will pass the course and thus fails to identify any failing student will have a balanced accuracy of 50% and an F_1-score of 75%. Under the same circumstances, a pessimistic classifier that consistently predicts that all students will fail the course has a balanced accuracy of 50% and an F_1-score of 0%.

5.2.5 Pass/fail predictions

In summary, Figure 5.1 outlines the entire flow of the proposed pass/fail prediction framework. It starts by extracting metadata for all submissions students made so far within a course (timestamp, status, student, exercise, series) and collecting their marks on intermediate tests and final exams (step 1). In practice, applying the framework on a student cohort in the current course edition only requires submission metadata and pass/fail outcomes from student cohorts in previous course editions. Successive course editions are then aligned by identifying fixed time points throughout the course where predictions are made, for example at submission deadlines, intermediate tests or final exams (step 2). We conducted a longitudinal study to evaluate the accuracy of pass/fail predictions at successive stages of a course (step 3). This is done by extracting features from the raw submission metadata of one or more course editions and training machine learning models that can identify at-risk students during other course editions. Our scripts that implement this framework are provided as supplementary material.[44] Teachers can also interpret the behaviour of students in their class by analysing the feature weights of the machine learning models (step 4).

5.3 Results and discussion

We evaluated the performance of four classification algorithms for pass/fail predictions in a longitudinal sequence of snapshots from course A and B: stochastic gradient descent (Figure 5.4), logistic regression (Figure 5.5), support vector machines (Figure 5.6), and random forests (Figure 5.7). For each classifier, course and snapshot, we evaluated 12 predictions for the following combinations of training and test sets: train on one edition and test on another edition; train on two editions and test on the other edition; train and test on one edition using 5-fold cross validation. In addition, we made predictions for course A using both the full set of features and a reduced feature set that ignores deadline-related features. We discuss the results in terms of accuracy, potential for early detection, and interpretability.

5.3.1 Accuracy

The overall conclusion from the longitudinal analysis is that indirectly measuring how students practice their coding skills by solving programming exercises (formative assessments) in combination with directly measuring how they perform on intermediate evaluations (summative assessments), allows us to predict with high accuracy if students will pass or fail a programming course. The signals to make such predictions seem to be present in the data, as we come to the same conclusions irrespective of the course, classification algorithm, or performance metric evaluated in our study. Overall, logistic regression was the best performing classifier for both courses, but the difference compared to the other classifiers is small.

When we compare the longitudinal trends of balanced accuracy for the predictions of both courses, we see that course A starts with a lower balanced accuracy at the first snapshot, but its accuracy increases faster and is slightly higher at the end of the semester. At the start of the semester at snapshot S1, course A has an average balanced accuracy between 60% and 65% and course B around 70%. Nearly halfway through the semester,

[44]`https://github.com/dodona-edu/pass-fail-article`

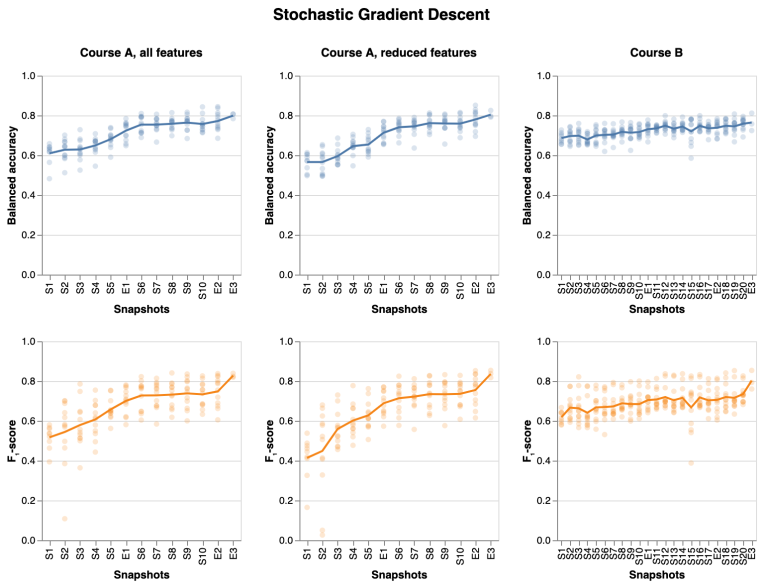


Figure 5.4: Performance of stochastic gradient descent classifiers for pass/fail predictions in a longitudinal sequence of snapshots from courses A (all features and reduced set of features) and B, measured by balanced accuracy and F_1-score. Dots represent performance of a single prediction, with 12 predictions for each group of corresponding snapshots (columns). Solid line connects averages of the performances for each group of corresponding snapshots.

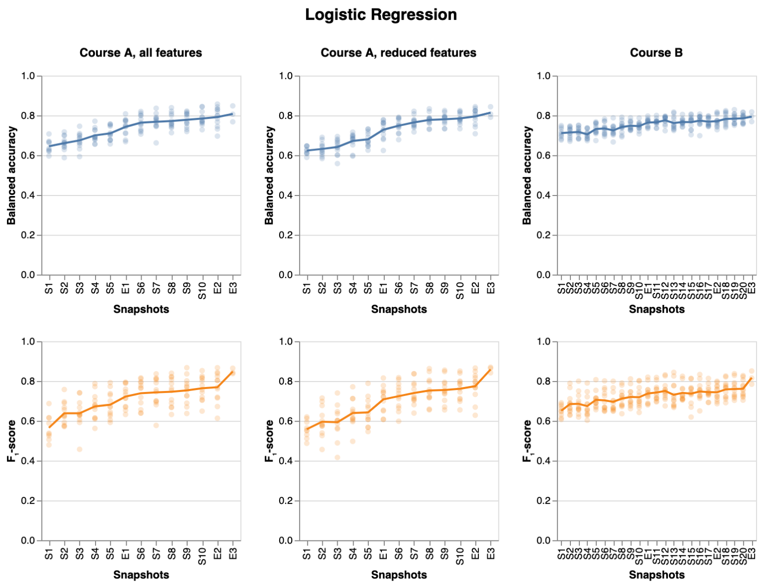


Figure 5.5: Performance of logistic regression classifiers for pass/fail predictions in a longitudinal sequence of snapshots from courses A (all features and reduced set of features) and B, measured by balanced accuracy and F$_1$-score. Dots represent performance of a single prediction, with 12 predictions for each group of corresponding snapshots (columns). Solid line connects averages of the performances for each group of corresponding snapshots.

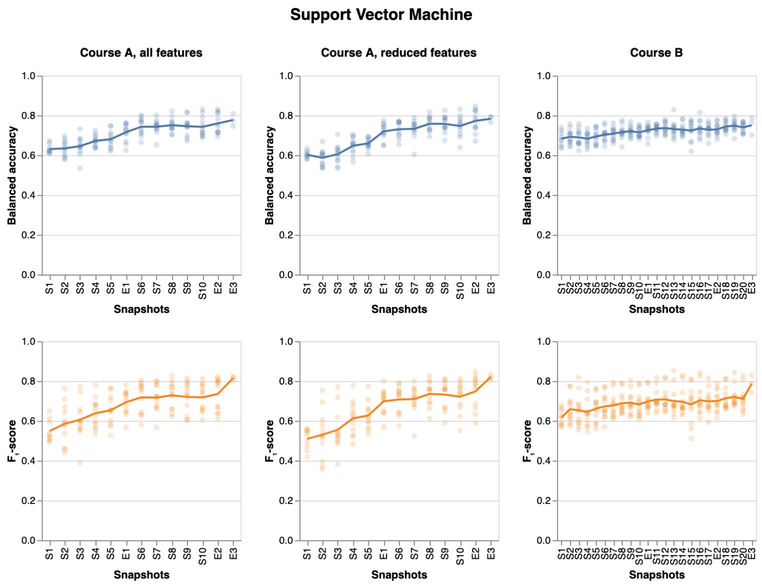


Figure 5.6: Performance of support vector machine classifiers for pass/fail predictions in a longitudinal sequence of snapshots from courses A (all features and reduced set of features) and B, measured by balanced accuracy and F_1-score. Dots represent performance of a single prediction, with 12 predictions for each group of corresponding snapshots (columns). Solid line connects averages of the performances for each group of corresponding snapshots.

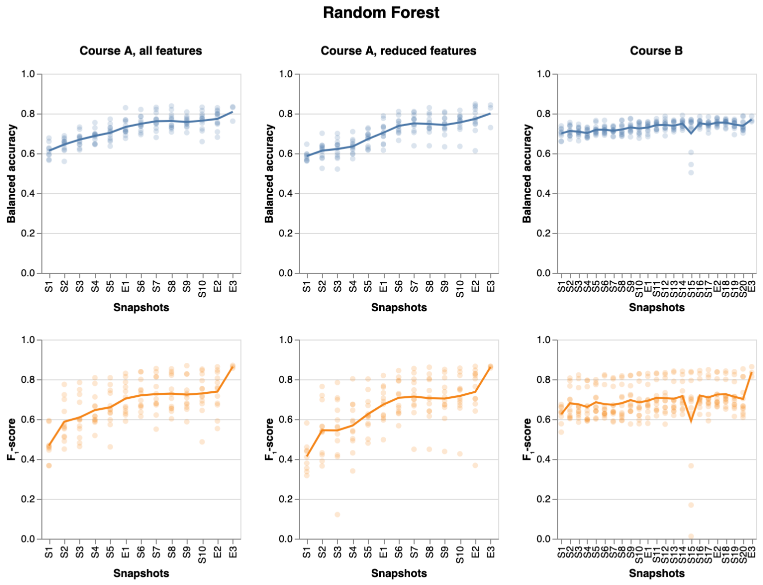


Figure 5.7: Performance of random forest classifiers for pass/fail predictions in a longitudinal sequence of snapshots from courses A (all features and reduced set of features) and B, measured by balanced accuracy and F_1-score. Dots represent performance of a single prediction, with 12 predictions for each group of corresponding snapshots (columns). Solid line connects averages of the performances for each group of corresponding snapshots.

before the first evaluation, we see an average balanced accuracy around 70% for course A at snapshot S5 and between 70% and 75% for course B at snapshot S8. After the first evaluation, we can make predictions with a balanced accuracy between 75% and 80% for both courses. The predictions for course B stay within this range for the rest of the semester, but for course A we can consistently make predictions with an average balanced accuracy of 80% near the end of the semester.

Compared to the accuracy results of Kovacic (2012), we see a 15-20% increase for our balanced accuracy results. Our balanced accuracy results are similar to the accuracy results of Livieris et al. (2019), who used semi-supervised machine learning. Asif et al. (2017) achieve an accuracy of about 80% when using one cohort of training and another cohort for testing, which is again similar to our balanced accuracy results. All of these studies used prior academic history as the basis for their methods, which we do not use in our framework. We also see similar results as compared to Vihavainen (2013) where we do not have to rely on data collection that interferes with the learning process. Note that we are comparing the basic accuracy results of prior studies with the more reliable balanced accuracy results of our framework.

F_1-scores follow the same trend as balanced accuracy, but the inclination is even more pronounced because it starts lower and ends higher. It shows another sharp improvement of predictive performance for both courses when students practice their programming skills in preparation of the final exam (snapshot E3). This underscores the need to keep organizing final summative assessments as catalysts of learning, even for courses with a strong focus on active learning.

The variation in predictive accuracy for a group of corresponding snapshots is higher for course A than for course B. This might be explained by the fact that successive editions of course B use the same set of exercises, supplemented with evaluation and exam exercises from the previous edition, whereas each edition of course A uses a different selection of exercises.

Predictions made with training sets from the same student cohort (5-fold cross-validation) perform better than those with training sets from different cohorts (see supplementary material for details[45]). This is more pronounced for F_1-scores than for balanced accuracy, but the differences are small enough so that nothing prevents us from building classification models with historical data from previous student cohorts to make pass/fail

[45]`https://github.com/dodona-edu/pass-fail-article`

predictions for the current cohort, which is something that can not be done in practice with data from the same cohort as pass/fail information is needed during the training phase. In addition, we found no significant performance differences for classification models using data from a single course edition or combining data from two course editions. Given that cohort sizes are large enough, this tells us that accurate predictions can already be made in practice with historical data from a single course edition. This is also relevant when the structure of a course changes, because we can only make predictions from historical data for course editions whose snapshots align.

The need to align snapshots is also the reason why we had to build separate models for courses A and B since both have differences in course structure. The models, however, were built using the same set of feature types. Because course B does not work with hard deadlines, deadline-related feature types could not be computed for its snapshots. This missing data and associated features had no impact on the performance of the predictions. Deliberately dropping the same feature types for course A also had no significant effect on the performance of predictions, illustrating that the training phase is where classification algorithms decide themselves how the individual features will contribute to the predictions. This frees us from having to determine the importance of features beforehand, allows us to add new features that might contribute to predictions even if they correlate with other features, and makes it possible to investigate afterwards how important individual features are for a given classifier (see Section 5.3.3).

Even though the structure of the courses is quite different, our method achieves high accuracy results for both courses. The results for course A with reduced features also still gives accurate results. This indicates that the method should be generalizable to other courses where similar data can be collected, even if the structure is quite different or when some features are impossible to calculate due to the course structure.

5.3.2 Early detection

Accuracy of predictions systematically increases as we capture more of student behaviour during the semester. But surprisingly we can already make quite accurate predictions early on in the semester, long before students take their first evaluation. Because of the steady trend, predictions for course B at the start of the semester are already reliable enough to serve as input for student feedback or teacher interventions. It takes some more time to identify at-risk students for course A, but from week four or five onwards the predictions may also become an instrument to design remedial actions for this course. Hard deadlines and graded exercises are a strong enforcement of active learning behaviour on the students of course A, and might disguise somewhat more the motivation of students to work on their programming skills. This might explain why it takes a bit longer to properly measure student motivation for course A than for course B.

5.3.3 Interpretability

So far, we have considered classification models as black boxes in our longitudinal analysis of pass/fail predictions. However, many machine learning techniques can tell us something about the contribution of individual features to make the predictions. In the case of our pass/fail predictions, looking at the importance of feature types and linking them to aspects of practising programming skills, might give us insights into what kind of behaviour promotes or inhibits learning, or has no or a minor effect on the learning process. Temporal information can tell us what behaviour makes a steady contribution to learning or where we see shifts throughout the semester.

This interpretability was a considerable factor in our choice of the classification algorithms we investigated in this study. Since we identified logistic regression as the best-performing classifier, we will have a closer look at feature contributions in its models. These models are explained by the feature weights in the logistic regression equation, so we will express the importance of a feature as its actual weight in the model. We use a temperature scale when plotting importances: white for zero importance, a red gradient for positive importance values and a blue gradient for negative importance values. A feature importance w can be interpreted as follows for logistic regression models: an increase of the feature value by one standard deviation increases the odds of passing the course by a factor of e^w when all other feature values remain the same (Molnar, 2019). The absolute value of the importance determines the impact the feature has on predictions. Features with zero importance have no impact because $e^0 = 1$. Features represented with a light colour have a weak impact and features represented with a dark colour have a strong impact. As a reference, an importance of 0.7 doubles the odds for passing the course with each standard deviation increase of the feature value, because $e^{0.7} \sim 2$. The sign of the importance determines whether the feature promotes or inhibits the odds of passing the course. Features with a positive importance (red colour) will increase the odds with increasing feature values, and features with a negative importance (blue colour) will decrease the odds with increasing feature values.

To simulate that we want to make predictions for each course edition included in this study, we trained logistic regression models with data from the remaining two editions of the same course. A label “edition 18–19” therefore means that we want to make predictions for the 2018–2019 edition of a course with a model built from the 2016–2017 and 2017–2018 editions of the course. However, in this case we are not interested in the predictions themselves, but in the importance of the features in the models. The importance of all features for each course edition can be found in the supplementary material.[46] We will restrict our discussion by highlighting the importance of a selection of feature types for the two courses.

For course A, we look into the evaluation scores (Figure 5.8) and the feature types `correct_after_15m` (Figure 5.9) and `wrong` (Figure 5.10). Evaluation scores have a very strong impact on predictions, and high evaluation scores increase the odds of passing the course. This comes as no surprise, as both the evaluations and exams are summative assessments that are organized and graded in the same way. Although the difficulty of evaluation exercises is lower than those of exam exercises, evaluation scores already are good predictors for exam scores. Also note that these features only show up in snapshots taken at or after the corresponding evaluation. They have zero impact on predictions for earlier snapshots, as the information is not available at the time these snapshots are taken.

The second feature type we want to highlight is `correct_after_15m`: the number of exercises in a series where the first correct submission was made within fifteen minutes after the first submission (Figure 5.9). Note that we can not directly measure how long students work on an exercise, as they may write, run and test their solutions on their local machine before their first submission to Dodona. Rather, this feature type measures how long it takes students to find and remedy errors in their code (debugging), after they start getting automatic feedback from Dodona.

For exercise series in the first unit of course A (series 1–5), we generally see that the features have a positive impact (red). This means that students will more likely pass the course if they are able to quickly remedy errors in their solutions for these exercises. The first and fourth series are an exception here. The fact that students need more time for the first series

[46] `https://github.com/dodona-edu/pass-fail-article`

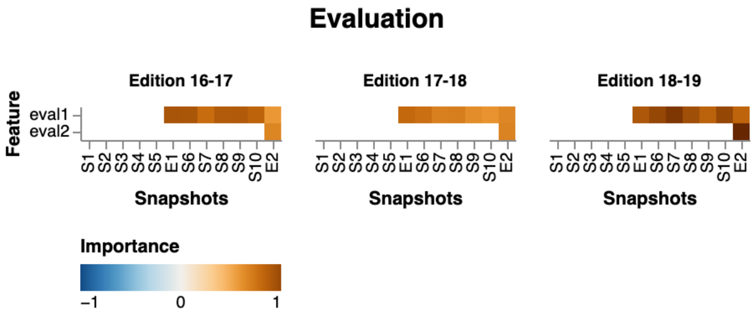


Figure 5.8: Importance of evaluation scores in the logistic regression models for course A (full feature set). Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be.

might reflect that learning something new is hard at the beginning, even if the exercises are still relatively easy. Series 4 of course A covers strings as the first compound data type of Python in combination with nested loops, where (non-nested) loops themselves are covered in series 3. This complex combination might mean that students generally need more time to debug the exercises in series 4.

For the series of the second unit (series 6–10), we observe two different effects. The impact of these features is zero for the first few snapshots (grey bottom left corner). This is because the exercises from these series were not yet published at the time of those snapshots, where all series of the first unit were available from the start of the semester. For the later snapshots, we generally see a negative (blue) weight associated with the features. It might seem counterintuitive and contradicts the explanation given for the series of the first unit. However, the exercises of the second unit are a lot more complex than those of the first unit. This up to a point that even for good students it is hard to debug and correctly solve an exercise in only 15 minutes. Students that need less than 15 minutes at this stage might be bypassing learning by copying solutions from fellow students instead of solving the exercises themselves. An exception to this pattern are the few red squares forming a diagonal in the middle of the bottom half. These squares correspond with exercises that are solved as soon as they become available as opposed to waiting for the deadline. A possible explanation for these few slightly positive weights is that these exercises are solved by highly-motivated, top students.

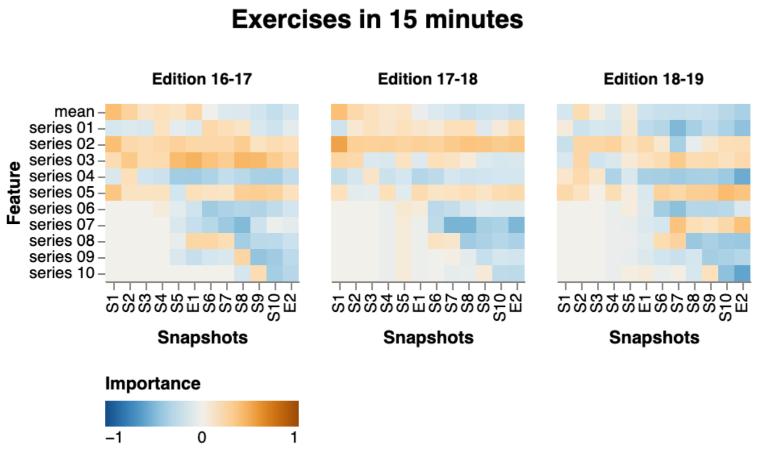


Figure 5.9: Importance of feature type `correct_after_15m` (the number of exercises in a series where the first correct submission was made within fifteen minutes after the first submission) in logistic regression models for course A (full feature set). Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be. Blues mean that a growth in the feature value decreases the odds. The darker the colour the larger this decrease will be.

Finally, if we look at the feature type `wrong` (Figure 5.10), submitting a lot of submissions with logical errors mostly has a positive impact on the odds of passing the course. This underscores the old adage that practice makes perfect, as real learning happens where students learn from their mistakes. Exceptions to this rule are found for series 2, 3, 9 and 10. The lecturer and teaching assistants identify the topics covered in series 2 and 9 by far as the easiest topics covered in the course, and identify the topics covered in series 3, 6 and 10 as the hardest. However, it does not feel very intuitive that being stuck with logical errors longer than other students either inhibits the odds for passing on topics that are extremely hard or easy or promotes the odds on topics with moderate difficulty. This shows that interpreting the importance of feature types is not always straightforward.

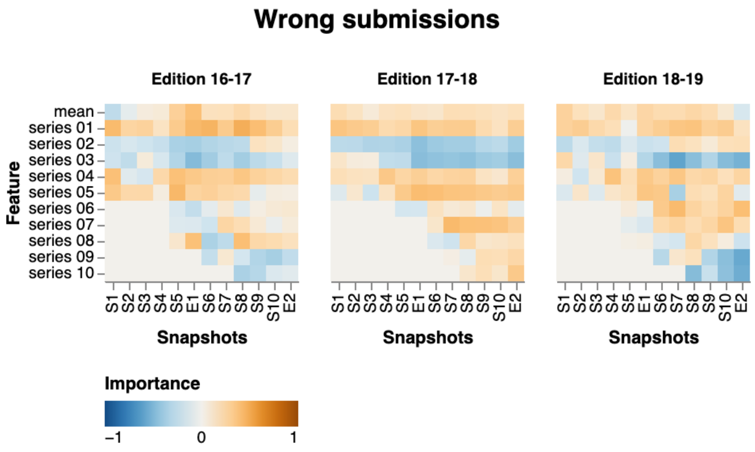


Figure 5.10: Importance of feature type `wrong` (the number of wrong submissions in a series) in logistic regression models for course A (full feature set). Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be. Blues mean that a growth in the feature value decreases the odds. The darker the colour the larger this decrease will be.

For course B, we look into the evaluation scores (Figure 5.11) and the feature types `comp_error` (Figure 5.12) and `wrong` (Figure 5.13). The importance of evaluation scores is similar as for course A, although their relative impact on the predictions is slightly lower. The latter might be caused by automatic grading of evaluation exercises, where exam exercises are graded by hand. The fact that the second evaluation is scheduled a little bit earlier in the semester than for course A, makes that pass/fail predictions for course B can also rely earlier on this important feature. However, we do not see a similar increase of the global performance metrics around the second evaluation of course B, as we see for the first evaluation.

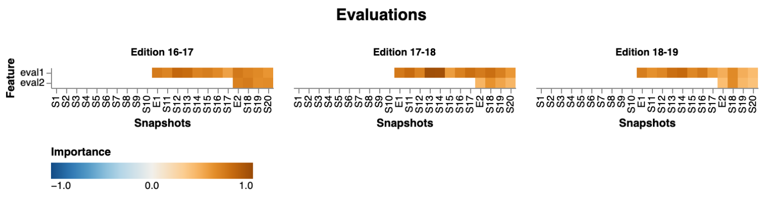


Figure 5.11: Importance of evaluation scores in the logistic regression models for course B. Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be.

Learning to code requires mastering two major competences: *i)* getting familiar with the syntax rules of a programming language to express the steps for solving a problem in a formal way, so that the algorithm can be executed by a computer, and *ii)* problem-solving itself. As a result, we can make a distinction between different kinds of errors in source code. Compilation errors are mistakes against the syntax of the programming language, whereas logical errors result from solving a problem with a wrong algorithm. When comparing the importance of the number of compilation (Figure 5.12) and logical errors (Figure 5.13) students make while practising their coding skills, we see a clear difference. Making a lot of compilation errors has a negative impact on the odds for passing the course (blue colour dominates in Figure 5.12), whereas making a lot of logical errors makes a positive contribution (red colour dominates in Figure 5.13). This aligns with the claim of Edwards et al. (2018) that problem-solving is a higher-order learning task in the Taxonomy by Bloom et al. (1956) (analysis and synthesis) than language syntax (knowledge, comprehension, and application). Students that get stuck longer in the mechanics of a programming language will more likely fail the course, whereas students that make a lot of logical errors and properly learn from them will more likely pass the course. So making mistakes is beneficial for learning, but it depends on what kind of mistakes. We also looked at the number of solutions with logical errors while interpreting feature types for course A. Although we hinted there towards the same conclusions as for course B, the signals were less consistent. This shows that interpreting feature importances always needs to take the educational context into account. This can also be seen in Figure 5.9, where some weeks contribute positively and some negatively. The reasons for these differences depend on the content of the course, which requires knowledge of the course contents to interpret correctly.

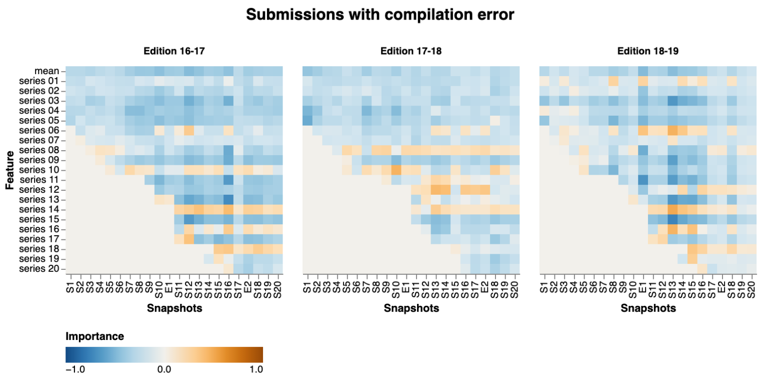


Figure 5.12: Importance of feature type `comp_error` (the number of submissions with compilation errors in a series) in logistic regression models for course B. Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be. Blues mean that a growth in the feature value decreases the odds. The darker the colour the larger this decrease will be.

5.4 Replication study at Jyväskylä University in Finland

After our original study, we collaborated with researchers from Jyväskylä University (JYU) in Finland on replicating the study in their introductory programming course (Zhidkikh et al., 2024). There are however, some notable differences to the study performed at Ghent University. In the new study, self-reported data was added to the model to test whether this enhances its predictions. Also, the focus shifted from pass/fail prediction to dropout prediction. This happened because of the different way the course at JYU is taught. By performing well enough in all weekly exercises and a project, students can already receive a passing grade. This is impossible in

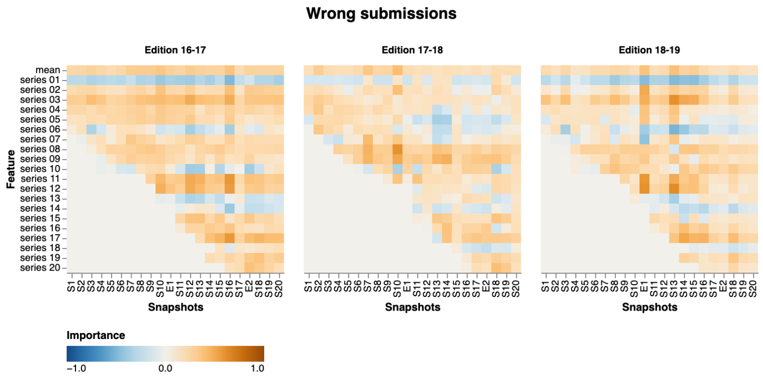


Figure 5.13: Importance of feature type `wrong` (the number of wrong submissions in a series) in logistic regression models for course B. Reds mean that a growth in the feature value for a student increases the odds of passing the course for that student. The darker the colour the larger this increase will be. Blues mean that a growth in the feature value decreases the odds. The darker the colour the larger this decrease will be.

the courses at Ghent University, where most of the final marks are earned at the exam at the end of the semester.

Another important difference in the two studies is the data that was available to feed into the machine learning model. Dodona keeps rich data about the evaluation results of a student's submission. In TIM (the learning environment used at JYU), only a score is kept for each submission. This score represents the underlying evaluation results (compilation error/mistakes in the output/…). While it is possible to reverse engineer the score into some underlying status, for some statuses that Dodona can make a distinction between this is not possible with TIM. This means that a different set of features had to be used in the study at JYU than the feature set used in the study at Ghent University. The specific feature types left out of the study at JYU are `comp_error` and `runtime_error`.

The course at JYU had been taught in the same way since 2015, resulting in behavioural and survey data from 2 615 students from the 2015–2021 academic years. The snapshots were made weekly as well, since the course also works with weekly assignments and deadlines. The self-reported data consists of pre-course and midterm surveys that inquire about aptitudes towards learning programming and motivation, including expectation about grades, prior programming experience, study year, attendance and number of concurrent courses.

In the analysis, the same four classifiers as the original study were tested. In addition to this, the dropout analysis was done for three datasets: *i)* behavioural data only, *ii)* self-reported data only, and *iii)* combined behavioural and self-reported data.

The results obtained in the study at JYU are very similar to the results obtained at Ghent University. Again, logistic regression was found to yield the best and most stable results. Even though no data about midterm evaluations or examinations was used (since this data was not available) a similar jump in accuracy around the midterm of the course was also observed. The jump in accuracy can be explained here by the fact that the period around the middle of the term is when most students drop out. It was also observed that the first weeks of the course play an important role in reducing dropout.

The addition of the self-reported data to the snapshots resulted in a statistically significant improvement of predictions in the first four weeks of the course. For the remaining weeks, the change in prediction performance was not statistically significant. This again points to the conclusion that the first few weeks of a CS1 course play a significant role in student success. The models trained only on self-reported data performed significantly worse than the other models.

The replication done at JYU showed that our prediction strategy can be used in significantly different educational contexts. Of course, adaptations to the method have to be made sometimes given differences in course structure and learning environment used, but these adaptations do not result in different prediction results.

5.5 Conclusions and future work

In this chapter, we presented a classification framework for predicting if students will likely pass or fail introductory programming courses. The framework already yields high-accuracy predictions early on in the semester and is privacy-friendly because it only works with metadata from programming challenges solved by students while working on their programming skills. Being able to identify at-risk students early on in the semester opens windows for remedial actions to improve the overall success rate of students.

We validated the framework by building separate classifiers for three courses because of differences in course structure, institute and learning platform, but using similar sets of features for training models. The results showed that submission metadata from previous student cohorts can be used to make predictions about the current cohort of students, even if course editions use different sets of exercises, or the courses are structured differently. Making predictions requires aligning snapshots between successive editions of a course, where students have the same expected progress at corresponding snapshots. Historical metadata from a single course edition suffices if group sizes are large enough. Different classification algorithms can be plugged into the framework, but logistic regression resulted in the best-performing classifiers.

Apart from their application to make pass/fail predictions, an interesting side effect of classification models that map indirect measurements of learning behaviour onto mastery of programming skills is that they allow us to interpret what behavioural aspects contribute to learning to code. Visualization of feature importance turned out to be a useful instrument for linking individual feature types with student behaviour that promotes or inhibits learning. We applied this interpretability to some important feature types that popped up for the three courses included in this study.

Our study has several strengths and promising implications for future practice and research. First, we were able to predict success based on historical metadata from earlier cohorts, and we are already able to do that early on in the semester. In addition to that, the accuracy of our predictions is similar to those of earlier efforts (Asif et al., 2017; Kovacic, 2012; Vihavainen, 2013) while we are not using prior academic history or interfering with the students' usual learning workflows. However, there are also some limitations and work for the future. While our visualizations of the features (Figures 5.8 through 5.13) are helpful to indicate which features are important at which stage of the course in view of increasing versus decreasing the odds of passing the course, they may not be oversimplified and need to be carefully interpreted and placed into context. This is where the expertise and experience of teachers comes in. These visualizations can be interpreted by teachers and further contextualized towards the specific course objectives. For example, teachers know the content and goals of every series of exercises, and they can use the information presented in our visualizations in order to investigate why certain series of exercises are more or less important in view of passing the course. In addition, they may use the information to further redesign their course.

We can thus conclude that the proposed framework achieves the objectives set for accuracy, early prediction and interpretability. Having this new framework at hand immediately raises some follow-up research questions that urge for further exploration: *i)* Do we inform students about their odds of passing a course? How and when do we inform students about their performance in an educationally responsible way? What learning analytics do we use to present predictions to students, and do we only show results or also explain how the data led to the results? How do students react to the announcement of their chance at passing the course? How do we ensure that students are not demotivated? *ii)* What actions could teachers take upon early insights which students will likely fail the course? What recommendations could they make to increase the odds that more students will pass the course? How could interpretations of important behavioural features be translated into learning analytics that give teachers more insight into how students learn to code? *iii)* Can we combine student progress (what programming skills does a student already have and at what level of mastery), student preferences (which skills does a student want to improve on), and intrinsic properties of programming exercises (what skills are needed to solve an exercise and how difficult is it) into dynamic learning paths that recommend exercises to optimize the learning effect for individual students?

6 Automating manual feedback

This chapter discusses the history of manual feedback in the programming course taught at the Faculty of Sciences at Ghent University (as described in the case study in Section 3.2) and how it influenced the development of evaluation and grading features within Dodona. We will then expand on some further experiments using data mining techniques we did to try to further reduce the time spent adding manual feedback. Section 6.5 is based on an article that is currently being prepared for submission.

Comments and evaluations were added to Dodona by myself. Niko Strijbol implemented the addition of grades to evaluations. Jorg Van Renterghem finalized the addition of feedback reuse. The work on feedback prediction was started by myself and further developed in collaboration with Kasper Demeyere during his master's thesis.

6.1 Phase 0: Paper-based assessment

Since the academic year 2015–2016 the programming course has started taking two open-book/open-internet evaluations in addition to the regular exam.[47] The first is a midterm and the other happens at the end of the semester (but before the exam period). The organization of these evaluations has been a learning process for everyone involved. Although the basic idea has remained the same (solve two Python programming exercises in two hours, or three in 3.5 hours for the exam), almost every aspect surrounding this basic premise has changed.

To be able to give feedback, student solutions were initially printed at the end of the evaluation. At first this happened by giving each student a USB stick on which they could find some initial files and which they had to copy their solution to. Later, this was replaced by a submission platform

[47]Before this, sessions were organized where students had to explain the code they submitted for an exercise. This was found not to be a great system, since it's far easier to explain code than to write it.

developed at Ghent University (Indianio) that had support for printing in the evaluation rooms. Indianio and its printing support was developed specifically to support courses in this format. Students were then allowed to check their printed solutions to make sure that the correct code was graded. This however means that the end of an evaluation takes a lot of time, since printing all these papers is a slow and badly parallelizable process (not the mention the environmental impact!).[48]

It also has some important drawbacks while grading. SPOJ (and later Dodona) was used to generate automated feedback on correctness. This automated feedback was not available when assessing a student's source code on paper. It therefore takes either more mental energy to work out whether the student's code would behave correctly with all inputs or it takes some hassle to look up a student's automated assessment results every time. Another important drawback is that students have a much harder time seeing their feedback. While their numerical grades were posted online or emailed to them, to see the comments graders wrote alongside their code they had to come to a hands-on session and ask the assistant there to be able to view the annotated version of their code (which could sometimes be hard to read, depending on the handwriting of the grader).[49] Very few students did so. There are a few possible explanations for this. They might experience social barriers for asking feedback on an evaluation they performed poorly on. For students who performed well, it might not be worth the hassle of going to ask about feedback. But maybe more importantly, a vicious cycle started to appear: because few students look at their feedback, graders did not spend much effort in writing out clear and useful feedback. Code that was too complex or plain wrong usually received little more than a strikethrough, instead of an explanation on why the student's method did not work.

6.2 Phase 1: Adding comments via Dodona

Seeing the amount of hassle that assessing these evaluations brought with them, we decided to build support for manual feedback and grading into Dodona. The first step of this was the functionality of adding comments

[48]The assignments themselves were also printed out and given to all students, which increased the amount of paper even more.

[49]For the second evaluation, the feedback was also scanned and emailed, since there were no more hands-on sessions. This was even the basis for a Dodona exercise: `https://dodona.be/en/activities/235452497/`.

to code. This work was started in the academic year 2019–2020, so the onset of the COVID-19 pandemic brought a lot of momentum to this work. Suddenly, the idea of printing student submissions became impossible, since the evaluations had to be taken remotely by students and the graders were working from home as well. Graders could now add comments to a student's code, which would allow the student to view the feedback remotely as well. An example of such a comment can be seen on Figure 6.1. There were still a few drawbacks to this system for assessing and grading though:

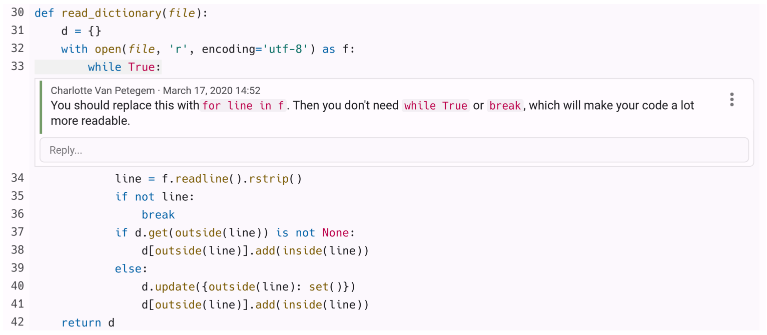
- Knowing which submissions to grade was not always trivial. For most students, the existing deadline system worked, since the solution they submitted right before the deadline was the submission taken into account when grading. There are however also students who receive extra time based on a special status granted to them by Ghent University (due to e.g. a learning disability). For these students, graders had to manually search for the submission made right before the extended deadline these students receive. This means that students could not be graded anonymously. It also makes the process a lot more error-prone.

- Comment visibility could not yet be time-gated towards students. This meant that graders had to write their comments in a local file with some extra metadata about the assessment. Afterwards this local file could be processed using some home-grown scripts to automatically add all comments at (nearly) the same time.

- Grades were added in external files, which was quite error-prone, since this involves manually looking up the correct student and entering their scores in a global spreadsheet. It is also less transparent towards students. While rubrics were made for every exercise that had to be graded, every grader had their preferred way of aggregating and entering these scores. This means that even though the rubrics exist, students had no option of seeing the different marks they received for different rubrics.

This was obviously not a great user experience, and not something we could recommend to anyone using Dodona who was not part of the Dodona development team.

We could already do some anecdotal analysis of this new system. One first observation that might seem counterintuitive is that graders did not feel like they spent less time grading. If anything, they reported spending



```
def read_dictionary(file):
    d = {}
    with open(file, 'r', encoding='utf-8') as f:
        while True:
```

Charlotte Van Petegem · March 17, 2020 14:52
You should replace this with `for line in f`. Then you don't need `while True` or `break`, which will make your code a lot more readable.

```
            line = f.readline().rstrip()
            if not line:
                break
            if d.get(outside(line)) is not None:
                d[outside(line)].add(inside(line))
            else:
                d.update({outside(line): set()})
                d[outside(line)].add(inside(line))
    return d
```

Figure 6.1: The first comment ever left on Dodona as part of a grading session.

more time grading. Another observation however is that graders gave more feedback and felt that the feedback they gave was of higher quality than before. In the first trial of this system, the feedback was viewed by over 80% of students within 24 hours, which is something that we had never observed when grading on paper.

6.3 Phase 2: Evaluations

To streamline and automate the process of grading even more, the concept of an evaluation was added to Dodona.[50] Evaluations address two of the drawbacks identified above:

- Comments made within an evaluation are linked to this evaluation. They are only made visible to students once the feedback of the evaluation is released.
- Evaluations also add an overview of the submissions that need to receive feedback. Since the submissions are explicitly linked to the evaluation, changing the submissions for students who receive extra time is also a lot less error-prone, since it can be done before actually starting out with the assessment. Evaluations also have specific UI to do this, where the timestamps are shown to teachers as accurately as Dodona saves them.

[50] See `https://docs.dodona.be/en/guides/teachers/grading/` for the actual process of creating an evaluation.

The addition of evaluations resulted in a subjective feeling of time being saved by the graders, at least in comparison with the previous system of adding comments.

To address the third concern mentioned above, another feature was implemented in Dodona. We added rubrics and a user-friendly way of entering scores. This means that students can view the scores they received for each rubric, and can do so right next to the feedback that was added manually.

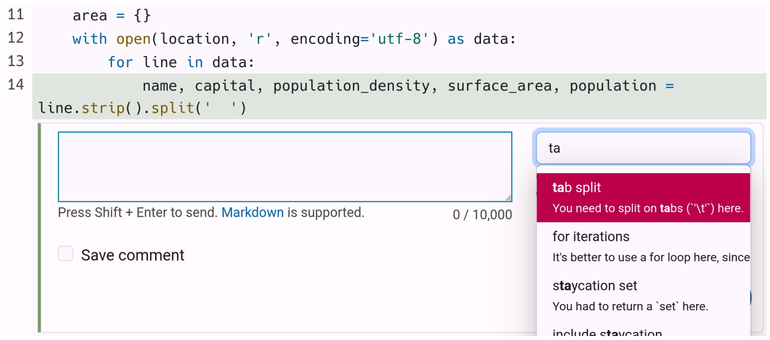
6.4 Phase 3: Feedback reuse

Grading and giving feedback has always been a time-consuming process, and the move to digital grading did not improve this compared to grading on paper. Even though the process itself was optimized, this optimization was used by graders to write out more and more comprehensive feedback.

Since evaluations are done with a few exercises solved by lots of students, there are usually a lot of mistakes that are common to a lot of students. This leads to graders giving the same feedback a lot of times. In fact, most graders maintained a list of commonly given feedback in a separate program or document.

We implemented the concept of feedback reuse to streamline giving commonly reused feedback. When giving feedback, the grader has the option to save the annotation they are currently writing. When they later encounter a situation where they want to give that same feedback, the only thing they have to do is write a few letters of the annotation in the saved annotation search box, and they can quickly insert the text written earlier. An example of this can be seen in Figure 6.2.

While originally conceptualized mainly for the benefit of graders, students can actually benefit from this feature as well. Graders only need to write out a detailed and clear message once and can then reuse that message over a lot of submissions instead of writing a shorter message each time. Because feedback is also added to a specific section of code, graders naturally write atomic feedback that is easier to reuse than monolothic sections of feedback (Moons et al., 2022).



```
    area = {}
    with open(location, 'r', encoding='utf-8') as data:
        for line in data:
            name, capital, population_density, surface_area, population =
line.strip().split('  ')
```

ta
tab split
You need to split on tabs (`\t`) here.
for iterations
It's better to use a for loop here, since
staycation set
You had to return a `set` here.
include staycation
Press Shift + Enter to send. Markdown is supported.
0 / 10,000
Save comment

Figure 6.2: An example of searching for a previously saved annotation.

6.5 Phase 4: Feedback prediction

Given that we now have a system for reusing earlier feedback, we can ask ourselves if we can do this in a smarter way. Instead of teachers having to search for the annotation they want to use, what if we could predict which annotation they want to use? This is exactly what we will explore in this section.

6.5.1 Introduction

Feedback is a key factor in student learning (Black & Wiliam, 1998; Hattie & Timperley, 2007). In programming education, many steps have been taken to give feedback using automated assessment systems (Ala-Mutka, 2005; Ihantola et al., 2010; Paiva, Leal, et al., 2022). These automated assessment systems give feedback on correctness, and can give some feedback on style and best practices through the use of linters. However, they are generally unable to give the same high-level feedback on program design that an experienced programmer can give. In many educational practices, automated assessment is therefore supplemented with manual feedback, especially when grading evaluations or exams (Debuse et al., 2008). This requires a significant time investment of teachers (Tuck, 2012). Reducing the time spent on giving feedback also benefits students since it generally means they will receive more timely feedback, which has long been considered important (Poulos & Mahony, 2008).

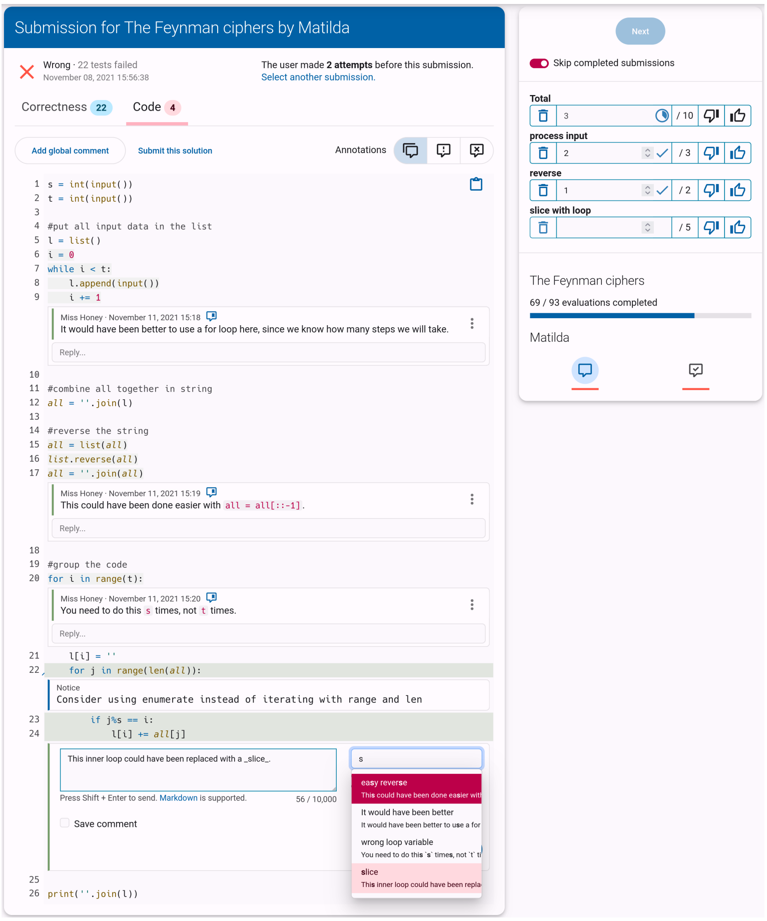
As a result, many researchers have explored the use of AI to enhance giving feedback. Vittorini et al. (2021) used natural language processing to automate grading, and found that students who used the system during the semester were more likely to pass the course at the end of the semester. Lee (2023) has used supervised learning with ensemble learning to enable students to perform peer and self-assessment. In addition, Bernius et al. (2022) introduced a framework based on clustering text segments in free-form textual exercises to reduce the grading workload. Strickroth & Holzinger (2023) attempt to solve this problem specifically for programming exercises by clustering submissions based on failed tests cases and compiler error messages.

The context of our work is the Dodona learning environment, developed at Ghent University (Van Petegem et al., 2023). Dodona gives automated feedback on each submitted solution to programming exercises, but also has a module that allows teachers to give manual feedback on student submissions and assign scores. The process of giving manual feedback on a submission to a programming exercise in Dodona is very similar to a code review, where errors or suggestions for improvements are annotated on the relevant line(s) (Figure 6.3). In 2023 alone, 3 663 749 solutions were submitted to Dodona, of which 44 012 were manually assessed. During manual assessment, 22 888 annotations were added to specific lines of code.

However, there is a crucial difference between traditional code reviews and those in an educational context: teachers often give feedback on numerous submissions to the same exercise. Since students often make similar mistakes in their submissions to an exercise, it logically follows that teachers will repeatedly give the same or similar feedback on multiple student submissions. To facilitate the reuse of feedback, Dodona allows teachers to save specific annotations for later search and retrieval. In 2023, 777 annotations were saved by teachers on Dodona, which were reused a total of 7 180 times. The usage of this functionality has generated data that we can use in this study: annotations that are shared between code submissions and that occur on specific lines of those submissions.

In this section we answer the following research questions: (RQ1) Can previously added annotations be used to predict what annotations a reviewer is likely to add to a specific line of code during manual assessment of student-written code? (RQ2) Additionally, can this be done so that both training of and predictions by the method are fast enough to use in live reviewing situations with human reviewers?

We present ECHO (Efficient Critique Harvesting and Ordering), a machine learning approach that aims to facilitate the reuse of previously given feedback. We begin with a detailed explanation of the design of ECHO. We



Submission for The Feynman ciphers by Matilda

Wrong · 22 tests failed
November 08, 2021 15:56:38

The user made **2 attempts** before this submission.
Select another submission.

Correctness 22 Code 4

Add global comment Submit this solution Annotations

```
 1 s = int(input())
 2 t = int(input())
 3
 4 #put all input data in the list
 5 l = list()
 6 i = 0
 7 while i < t:
 8     l.append(input())
 9     i += 1
```

Miss Honey · November 11, 2021 15:18
It would have been better to use a for loop here, since we know how many steps we will take.
Reply...

```
10
11 #combine all together in string
12 all = ''.join(l)
13
14 #reverse the string
15 all = list(all)
16 list.reverse(all)
17 all = ''.join(all)
```

Miss Honey · November 11, 2021 15:19
This could have been done easier with `all = all[::-1]`.
Reply...

```
18
19 #group the code
20 for i in range(t):
```

Miss Honey · November 11, 2021 15:20
You need to do this `s` times, not `t` times.
Reply...

```
21     l[i] = ''
22     for j in range(len(all)):
```

Notice
Consider using enumerate instead of iterating with range and len

```
23         if j%s == i:
24             l[i] += all[j]
```

This inner loop could have been replaced with a _slice_.

Press Shift + Enter to send. Markdown is supported. 56 / 10,000

Save comment

s

easy reverse
This could have been done easier wit

It would have been better
It would have been better to use a for

wrong loop variable
You need to do this `s` times, not `t` ti

slice
This inner loop could have been repla

```
25
26 print(''.join(l))
```

Next

Skip completed submissions

Total
3 / 10

process input
2 / 3

reverse
1 / 2

slice with loop
/ 5

The Feynman ciphers

69 / 93 evaluations completed

Matilda

Figure 6.3: Assessment of a submitted solution in Dodona. An automated assessment has already been performed, with 22 failed test cases, as can be seen from the badge on the "Correctness" tab. An automated annotation left by Pylint can be seen on line 22. A teacher gives feedback on the code by adding inline annotations and scores the submission by filling out the exercise-specific scoring rubric. The teacher has just searched for a previously saved annotation so that they could reuse it. After manually assessing this submission, the teacher still has another 23 submissions to assess, as shown in the progress bar on the right.

then present and discuss the experimental results we obtained by testing ECHO on student submissions. The dataset we used for this experiment is based on real Python code written by students during exams. First, we test ECHO by predicting Pylint machine annotations. Next, we use annotations left by human reviewers during manual assessment.

6.5.2 Methodology

We consider predicting relevant annotations to be a ranking problem, which we solve by determining similarity between the lines of code where annotations are added. The approach to determine this similarity is based on tree mining. This is a data mining technique for extracting frequently occurring patterns from data that can be represented as trees (Asai et al., 2004; Zaki, 2005). Program code can be represented as an abstract syntax tree (AST), where the nodes of the tree represent the language constructs used in the program. Recent work has demonstrated the efficacy of this approach in efficiently identifying frequent patterns in source code (Pham et al., 2019). In an educational context, these techniques have already been used to find patterns common to solutions that failed a given exercise (Mens et al., 2021). Other work has demonstrated the potential of automatically generating unit tests from mined patterns (Lienard et al., 2023). We use tree mining to find commonalities between the lines of code where the same annotation has been added.

We begin with a general overview of ECHO (Figure 6.4). The first step is to use the tree-sitter library (Brunsfeld et al., 2024) to generate ASTs for each submission. Using tree-sitter makes ECHO independent of the programming language used, since it presents an interface for generating syntax trees independent of the programming language. The syntax trees are post-processed to include identifier names. For each annotation, we identify all occurrences and extract a constrained AST context around the annotated line for each instance. The resulting subtrees are then aggregated for each annotation. If there are three or more subtrees, they are processed by the `TreeminerD` algorithm (Zaki, 2005). This yields a set of frequently occurring patterns specific to that annotation. We then assign weights to these patterns based on their length and their frequency across the entire dataset of patterns for all annotations. In addition to pattern mining, we also determine a set of unique nodes per forest of subtrees. The result of these operations is our trained model.

The model can then be used to score how well an annotation matches a given code fragment. In practice, the reviewer first selects a line of code in a given student's submission. Next, the AST of the selected line and its surrounding context is generated. For each annotation, each of its patterns is matched to the line, and a similarity score is calculated, given the previously determined weights. The percentage of unique nodes which match in the current subtree is also taken into account. These scores are used to rank the annotations, which are then displayed to the reviewer. It is important to note that the reviewer remains in control of which annotation is used, if any.

We will now provide a more in-depth explanation of this process, with a particular emphasis on operational efficiency. Speed is of the utmost importance throughout the model's lifecycle, from training to deployment in real-time reviewing contexts. Given the continuous generation of training data during the review process, the model's training time must be optimized to avoid significant delays, ensuring that the model remains practical for live review situations.

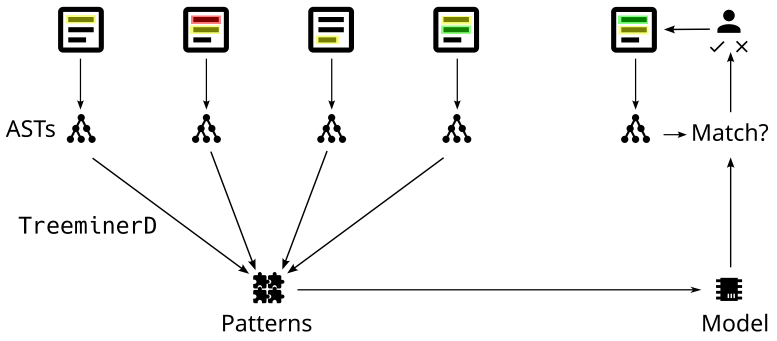


Figure 6.4: Overview of ECHO. Code of previously reviewed submissions is converted to its abstract syntax tree (AST) form. Instances of the same annotation have the same colour. For each annotation, the context of each instance is extracted and mined for patterns using the `TreeminerD` algorithm. These patterns are then weighted to form our model. When a reviewer wants to place an annotation on a line of the submissions they are currently reviewing, all previously given annotations are ranked based on the similarity determined for that line. The reviewer can then choose which annotation they want to place, with the aim of having the selected annotation at the top of in the ranking.

Training

The first step of ECHO is to extract a subtree for each instance of an annotation and then aggregate them per annotation. Currently, the context around a line is extracted by taking all the AST nodes from that line. For example, Figure 6.5 shows that the subtree extracted for the code on line 3 of Listing 6.1. Note that the context we extract here is very limited. Previous iterations of ECHO considered all nodes that contained the relevant line (e.g. the function node for a line in a function), but these contexts proved too large to process in an acceptable time.

```
1 def jump(alpha, n):
2     alpha_number = ord(alpha)
3     adjusted = alpha_number + n
4     return chr(adjusted)
```

Listing 6.1: Example code that simply adds a number to the ASCII value of a character and converts it back to a character.

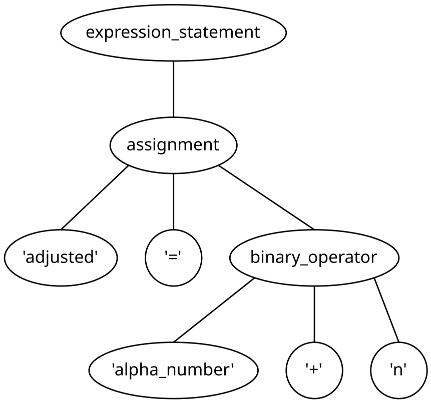


Figure 6.5: AST subtree corresponding to line 3 in Listing 6.1 as generated by tree-sitter.

After collecting subtrees for each annotation, ECHO mines patterns from these subtrees using `TreeminerD` (Zaki, 2005): an algorithm for discovering frequently occurring patterns in datasets of rooted, ordered and labelled trees. `TreeminerD` starts with a list of frequently occurring nodes, and then iteratively expands the frequently occurring patterns. Patterns are embedded subtrees: the nodes in a pattern are a subset in the nodes of the tree, preserving the ancestor-descendant relationships and the left-to-right order of the nodes. An example of a valid pattern for the tree in Figure 6.5 is shown in Figure 6.6.

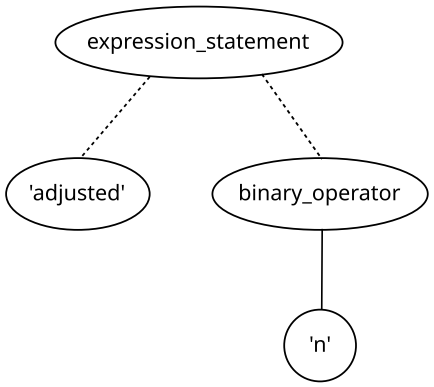


Figure 6.6: Valid pattern for the tree in Figure 6.5. Indirect ancestor-descendant relationships are marked with dashed lines.

In the `TreeminerD` algorithm, frequent means that the number of times the pattern occurs in all trees divided by the number of trees is greater than some predefined threshold. This is called the `minimum support` parameter of the algorithm.

We use a custom implementation of the `TreeminerD` algorithm, to find patterns in the AST subtrees for each annotation. Due to the exponential nature of the number of possible patterns in a tree, we only mine for patterns when there are at least three trees.

ECHO now has a set of patterns corresponding to each annotation. However, some patterns are more informative that others. So it assigns weights to the patterns it gets from `TreeminerD`.

Weights are assigned using two criteria. The first criterion is the size of the pattern (i.e., the number of nodes in the pattern), since a pattern with twenty nodes is much more specific than a pattern with only one node. The second criterion is the number of occurrences of a pattern across all annotations. If the pattern sets for all annotations contain a particular pattern, it can not be used reliably to determine which annotation should be predicted and is therefore given a lower weight. Weights are calculated using the formula below.

$$\text{weight}(\textit{pattern}) = \frac{\text{len}(\textit{pattern})}{\#\text{occurences}(\textit{pattern})}$$

In addition to mining and weighting patterns, ECHO also determines a set of nodes that are unique to the subtrees of each annotation. This is done by taking the union of the nodes of all subtrees for that annotation, and then removing from that set any nodes that occur in the subtrees of at least three other annotations. This step does not require a minimum number of instances per annotation.

Ranking

Having completed the above steps, ECHO has trained its model. To use the model, ECHO needs to know how to match patterns to subtrees.

To check whether a given pattern matches a given subtree, we iterate over all the nodes in the subtree. At the same time, we also iterate over the nodes in the pattern. During the iteration, we also store the current depth, both in the pattern and the subtree. We also keep a stack to store (some of) the depths of the subtree. If the current label in the subtree and the pattern are the same, we store the current subtree depth on the stack and move to the next node in the pattern. Moving up in the tree is more complicated. If the current depth and the depth of the last match (stored on the stack) are the same, we can move forwards in the pattern (and the subtree). If not, we need to check that we are still in the embedded subtree, otherwise we need to reset our position in the pattern to the start. Since subtrees can contain multiple instances of the same label, we also need to make sure that we can backtrack. Listings 6.2 and 6.3 give the full pseudocode for this algorithm.

Checking whether a pattern matches a subtree is an operation that ECHO has to perform many times. For some annotations there are many patterns, and all patterns of all annotations are checked. An important optimization we added was to run the algorithm in Listings 6.2 and 6.3 only if the set of labels in the pattern is a subset of the labels in the subtree.

Given a model where we have weighted patterns for each annotation, and a method for matching patterns to subtrees, we can now put the two together to make a final ranking of the available annotations for a given

```
start, p_i, pattern_depth, depth = 0
depth_stack, history = []

subtree_matches(subtree, pattern):
  result = find_in_subtree(subtree, subtree)
  while not result and history is not empty:
    to_explore, to_explore_subtree = history.pop()
    while not result and to_explore is not empty:
      start, depth, depth_stack, p_i = to_explore.pop()
      new_subtree = to_explore_subtree[start:]
      start = 0
      if pattern_length - p_i <= len(new_subtree) and
       ↪  new_subtree is fully contained in pattern[p_i:]:
        result = find_in_subtree(subtree, new_subtree)
   return result
```

Listing 6.2: Pseudocode for checking whether a pattern matches a subtree. Note that both the pattern and the subtree are stored in the encoding described by Zaki (2005). The implementation of **`find_in_subtree`** can be found in Listing 6.2.

line of code. We calculate a match score for each annotation using the formula below.

$$\text{score}(\mathit{annotation}) = \frac{\displaystyle\sum_{\substack{\mathit{pattern} \\ \in \mathit{patterns}}} \begin{cases} \text{weight}(\mathit{pattern}) & \mathit{pattern} \text{ matches} \\ 0 & \text{otherwise} \end{cases}}{\text{len}(\mathit{patterns})}$$

ECHO then ranks the annotations by combining the score and the percentage of nodes in the set of unique nodes for that annotation.

6.5.3 Results and discussion

As a dataset to validate ECHO, we used Python code written by students for programming exercises from (different) exams. The dataset contains between 135 and 214 submissions per exercise. Each submission for a particular exercise is by a different student. We first split the datasets equally into a training set and a test set. This simulates the midpoint of an assessment session for the exercise. During testing, we let our model suggest annotations for each of the lines that had an actual annotation

```
find_in_subtree(subtree, current_subtree):
  local_history = []
  for item in subtree:
    if item == -1:
      if depth_stack is not empty and depth - 1 == depth_stack.last:
        depth_stack.pop()
        if pattern [p_i] != -1:
          p_i = 0
          if depth_stack is empty:
            history.append((local_history, current_subtree[:i + 1])
            local_history = []
        else:
          p_i += 1
      depth -= 1
    else:
      if pattern[p_i] == item:
        local_history.append((start + i + 1, depth + 1, depth_stack, p_i))
        depth_stack.append(depth)
        p_i += 1
      depth += 1
    if p_i == pattern_length:
      return True
  if local_history is not empty:
    history.append((local_history, current_subtree))
  return False
```

Listing 6.3: Continuation of Listing 6.2.

associated with them in the test set. We evaluate where the correct annotation is ranked. We only look at the top five to get a good idea of how useful the suggested ranking would be in practice: if an annotation is not in the top five, we would expect the reviewer to have to search for it manually, rather than selecting it directly from the suggested ranking.

We first ran Pylint[51] (version 3.1.0) on the students' submissions. Pylint is a static code analyser for Python that checks for errors and code smells, and enforces a standard programming style. We used Pylint's machine annotations as our training and test data. We test per exercise because that's our main use case for ECHO, but we also run a test that combines all submissions from all exercises. An overview of some annotation statistics for the data generated by Pylint can be found in Table 6.1.

Exercise	subm.	ann.	inst.	max	avg
A last goodbye	135	25	189	29	7.56
Symbolic	141	28	277	66	9.89
Narcissus cipher	144	29	148	24	5.10
Cocktail bar	211	31	162	29	5.23
Anthropomorphic emoji	214	24	144	40	6.00
Hermit	194	82	388	59	6.80
Combined	1039	82	1479	196	18.04

Table 6.1: Statistics of Pylint annotations for the programming exercises used in the benchmark.

In a second experiment, we used the manual annotations left by human reviewers on student code in Dodona. Exercises were reviewed by different people, but all submissions for a specific exercise were reviewed by the same person. The reviewers were not aware of ECHO at the time they reviewed the submissions. In this case there is no combined test as the set of annotations used is different for each exercise.

We distinguish between these two sources of annotations because we expect Pylint to be more consistent in both when it places an instance of an annotation and also where it places the instance. Most linting annotations are detected by explicit pattern matching in the AST, so we expect the implicit pattern matching to work fairly well. However, we want to skip this explicit pattern matching for manual annotations because of the time it takes to compile them and the fact that annotations are often specific to a particular exercise and reviewer. Therefore, we also test on manual

[51] `https://www.pylint.org/`

annotations. Manual annotations are expected to be more inconsistent because reviewers may miss a problem in one student's code that they have annotated in another student's code, or they may not place instances of a particular annotation in consistent locations. The method by which human reviewers place an annotation is also much more implicit than Pylint's pattern matching.

The reviewed programming exercises have between 55 and 469 instances of manual annotations. The number of distinct annotations varies between 7 and 34 per exercise. Table 6.2 gives an overview of some of characteristics of the dataset. Timings mentioned in this section were measured on a 2022 Dell laptop with a 3GHz Intel quad-core processor and 32 GB of RAM.

Exercise	subm.	ann.	inst.	max	avg
A last goodbye	135	34	334	92	9.82
Symbolic	141	7	55	25	7.85
Narcissus cipher	144	17	193	55	11.35
Cocktail bar	211	15	469	231	31.27
Anthropomorphic emoji	214	27	322	39	11.93
Hermit	194	32	215	27	6.71

Table 6.2: Statistics of manually added annotations for the programming exercises used in the benchmark.

Machine annotations (Pylint)

We will first discuss the results for the Pylint annotations. During the experiment, a few Pylint annotations that are not related to the structure of the code were omitted to avoid distorting the results. These are "line too long", "trailing whitespace", "trailing newlines", "missing module docstring", "missing class docstring", and "missing function docstring". Depending on the exercise, the actual annotation is ranked among the top five annotations in 45% to 77% of all test instances (Figure 6.7). The annotation is even ranked first for 23% to 52% of all test instances. Interestingly, the method performs worse when the instances for all exercises are combined. This highlights the fact that ECHO is most useful in the context of reviewing similar code many times. For the submissions and instances in the training set, training took between 70 and 245 milliseconds to process all submissions and instances for an exercise. The entire test phase took between 30 and 180 milliseconds per exercise. Individual predictions never exceed 15 milliseconds.

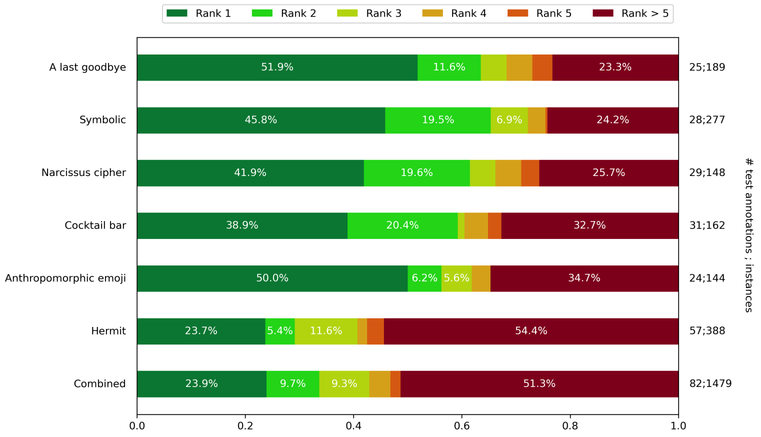


Figure 6.7: Prediction accuracy for suggesting instances of Pylint annotations. The numbers on the right are the total number of annotations and instances respectively. The "Combined" test evaluated ECHO on the entire set of submissions for all exercises.

We have selected some interesting annotations for further inspection, some of which perform very well, and some of which perform less well (Figure 6.8). We chose these specific annotations to demonstrate interesting behaviours exhibited by ECHO. The differences in performance can be explained by the content of the annotation and the underlying patterns that Pylint is looking for. For example, the "unused variable"[52] annotation performs poorly. This can be explained by the fact that we do not feed `TreeminerD` with enough context to find predictive patterns for this Pylint annotation. There are also annotations that can not be predicted at all, because no patterns are found.

Other annotations, such as "consider using with"[53], work very well. For these annotations, `TreeminerD` does have enough context to automatically determine the underlying patterns. The number of instances of an annotation in the training set also has an effect. Annotations with few instances are generally predicted worse than those with many instances.

[52]`https://pylint.pycqa.org/en/latest/user_guide/messages/warning/unused-variable.html`

[53]`https://pylint.pycqa.org/en/latest/user_guide/messages/refactor/consider-using-with.html`

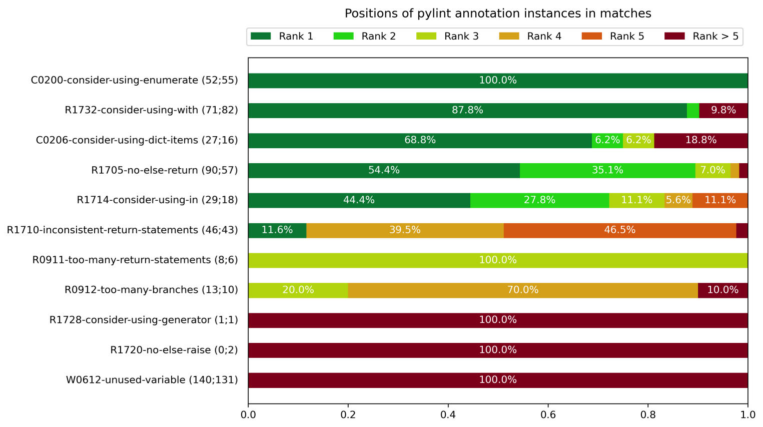


Figure 6.8: Prediction accuracy for a selection of Pylint machine annotations. Each line corresponds to a Pylint annotation, with the number of instances in the training and test sets given in parentheses after the annotation name.

Human annotations

For the annotations added by human reviewers, we used two different scenarios to evaluate ECHO. In addition to using the same 50/50 split between training and test data as for the Pylint data, we also simulated how a human reviewer would use ECHO in practice by gradually increasing the training set and decreasing the test set as the reviewer progresses through the submissions during the assessment. At the start of the assessment, no annotations are available and the first instance of an annotation that applies to a reviewed submission can not be predicted. As more submissions are reviewed and more instances of annotations are placed on those submissions, the training set for modelling predictions on the next submission under review gradually grows.

If we split the submissions and the corresponding annotations of a human reviewer equally into a training and a test set, the prediction accuracy is similar or even slightly better compared to the Pylint annotations (Figure 6.9). The number of instances where the true annotation is ranked first is generally higher (between 29% and 63% depending on the exercise), and the number of instances where it is ranked in the top five is between 63% and 93% depending on the exercise.

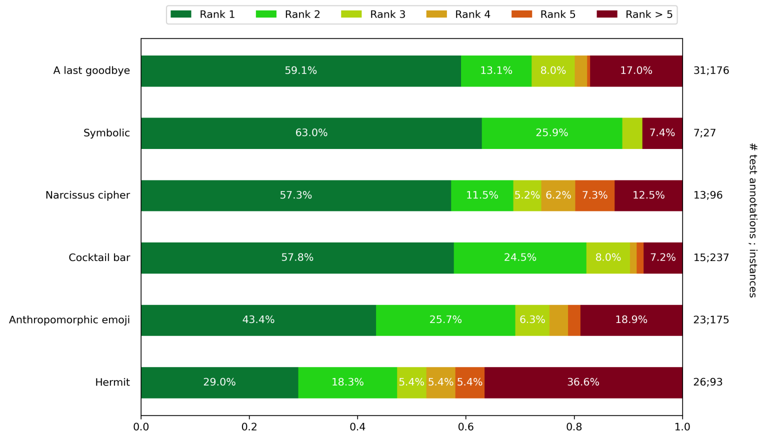


Figure 6.9: Prediction accuracy for suggesting instances of annotations by human reviewers. The numbers on the right are the total number of annotations and instances respectively.

In this experiment, training took between 67 milliseconds and 22.4 seconds per exercise. The entire test phase took between 49 milliseconds and 27 seconds, depending on the exercise. These evaluations were run on the same hardware as those for the machine annotations. For one prediction, the average time ranged from 0.1 milliseconds to 150 milliseconds and the maxima from 0.5 milliseconds to 2.8 seconds. The explanation for these wide ranges remains the same as for the Pylint predictions: it all depends on the number of patterns found.

These results show that we can predict reuse with a fairly high accuracy at the midpoint of a review session for a programming exercise. The accuracy depends on the number of instances per annotation and the consistency of the reviewer. Looking at the underlying data, we can also see that short, atomic messages can be predicted very well, as suggested by Moons et al. (2022). We will now look at the longitudinal prediction accuracy of ECHO, to test how accuracy evolves over the course of a review session.

For the next experiment, we introduce two specific categories of negative prediction results, namely "No training instances" and "No patterns". "No training instances" means that the annotation corresponding to the true instance had no instances in the training set, and therefore could never have been predicted. "No patterns" means that `TreeminerD` was unable to find any frequent patterns for the set of subtrees extracted from the annotation instances and there were also no nodes unique to this set of subtrees in the entire set of subtrees. This could be because the collection of subtrees is too diverse to have common patterns.

Figures 6.10, 6.11, 6.12 and 6.13 show the results of this experiment for four of the programming exercises used in the previous experiments. The "Symbolic" exercise was excluded due to its low number of unique annotations, while the "Hermit" exercise was excluded due to its poor performance in the previous experiment. We also excluded submissions that received no annotations during the human review process, which explains the lower number of submissions compared to the numbers in Table 6.2. This experiment shows that while the review process requires some time to build up before sufficient training instances are available, once a critical mass of training instances is reached, the accuracy for suggesting new instances of annotations reaches its maximum predictive power. This critical mass is reached after about 20 to 30 submissions reviewed, which is quite early in the review process (Figure 6.14). This means that a lot of time could be saved during the review process when ECHO is integrated into an online learning environment. The point at which the critical mass is reached will of course depend on the nature of the exercises and the consistency of the reviewer.

As mentioned above, we are working with a slightly inconsistent dataset when using annotations from human reviewers. They will sometimes miss an instance of an annotation, place it inconsistently, or unnecessarily create duplicate annotations. If ECHO is used in practice, the predictions may be even better, as the knowledge of its existence may further motivate reviewers to be more consistent in their reviews. The programming exercises were also reviewed by different people, which may also explain the differences in prediction accuracy between the exercises.

To evaluate the performance of ECHO for these experiments, we measure the training times, and the times required for each prediction. This corresponds to a reviewer wanting to add an annotation to a line in practice. Figures 6.15, 6.16, 6.17, and 6.18 show the performance of running these experiments. As in the previous experiments, we can see that there is a considerable difference between the exercises. However, the training time only exceeds one second in a few cases and remains well below that in most cases. The prediction times are mostly below 50 milliseconds, except for a few outliers. The average prediction time never exceeds 500 milliseconds.

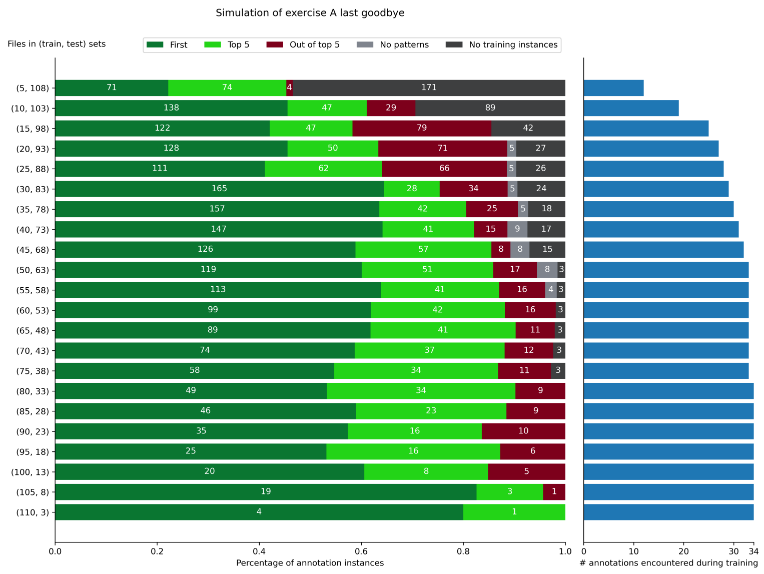


Figure 6.10: Progression of the prediction accuracy for the “A last goodbye” exercise over the course of the review process. Predictions for instances whose annotation had no instances in the training set are classified as “No training instances”. Predictions for instances whose annotation had no corresponding patterns in the model learned from the training set are classified as “No patterns”. The graph on the right shows the number of annotations present with at least one instance in the training set.

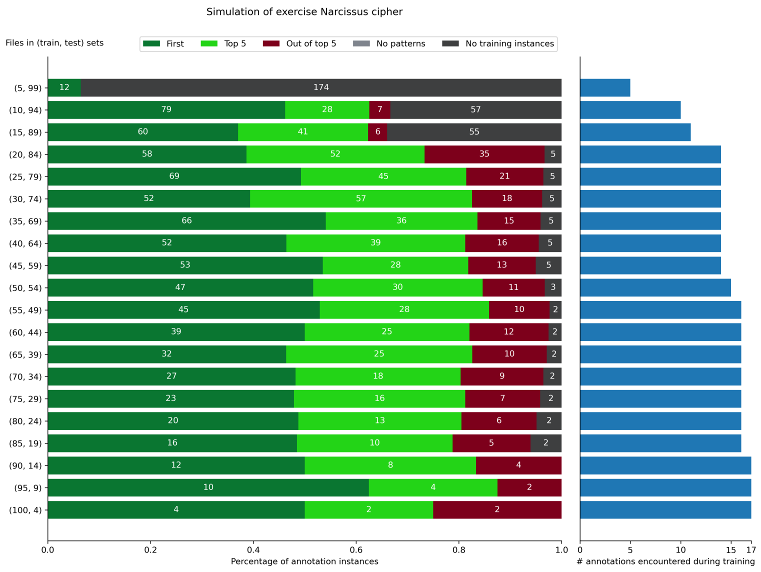


Figure 6.11: Progression of the prediction accuracy for the “Narcissus cipher” exercise over the course of the review process. Predictions for instances whose annotation had no instances in the training set are classified as “No training instances”. Predictions for instances whose annotation had no corresponding patterns in the model learned from the training set are classified as “No patterns”. The graph on the right shows the number of annotations present with at least one instance in the training set.

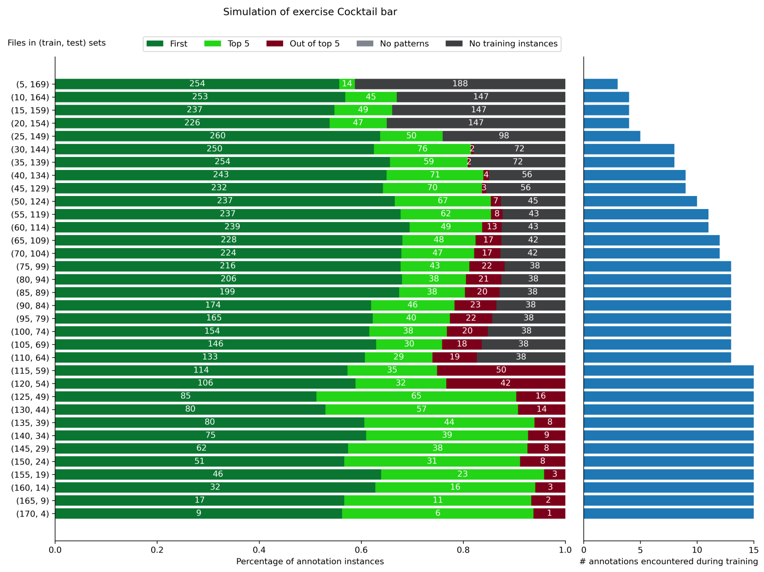


Figure 6.12: Progression of the prediction accuracy for the “Cocktail bar” exercise over the course of the review process. Predictions for instances whose annotation had no instances in the training set are classified as “No training instances”. Predictions for instances whose annotation had no corresponding patterns in the model learned from the training set are classified as “No patterns”. The graph on the right shows the number of annotations present with at least one instance in the training set.

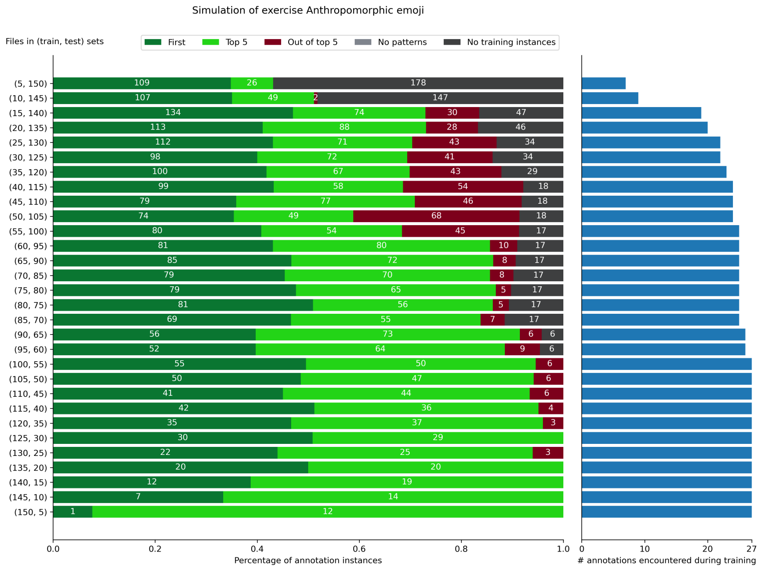


Figure 6.13: Progression of the prediction accuracy for the “Anthropomorphic emoji” exercise over the course of the review process. Predictions for instances whose annotation had no instances in the training set are classified as “No training instances”. Predictions for instances whose annotation had no corresponding patterns in the model learned from the training set are classified as “No patterns”. The graph on the right shows the number of annotations present with at least one instance in the training set.

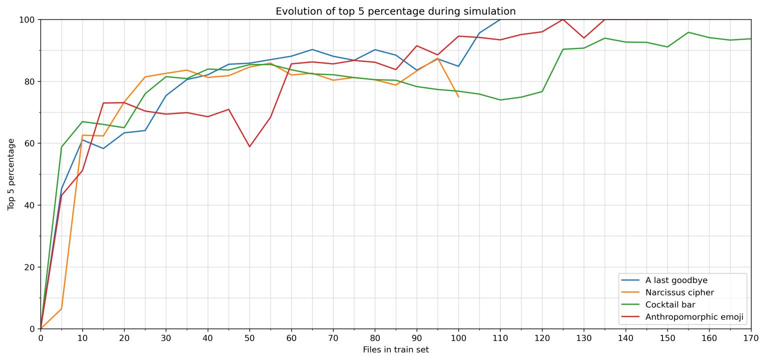


Figure 6.14: Evolution of the percentage of suggestions that are ranked in the top 5. The percentages are fairly stable after 20 to 30 submissions have been reviewed.

The timings show that although there are some outliers, predictions can be made fast enough to make this an interactive system. The outliers also correspond to higher training times, indicating that this is mainly caused by a high number of underlying patterns for some annotations. Currently this process is also parallelized over the files, but in practice, the process could be parallelized over the patterns, which would speed up the prediction even more. Note that the training time may also decrease with more training data. If there are more instances per annotation, the diversity in the related subtrees will usually increase, which reduces the number of patterns that can be found and thus reduces the training time.

6.5.4 Conclusions and future work

We presented ECHO as a predictive method to assist human reviewers in giving feedback when reviewing students submissions to a programming exercise by reusing annotations. Improving the reuse of annotations can both save time and improve the consistency with which feedback is given. The latter in itself might further improve the accuracy of predictions if the strategy is applied during the review process.

ECHO has already shown promising results. We have validated the framework both by predicting automated linting annotations to establish a baseline, and by predicting annotations from human reviewers. The method has about the same prediction accuracy for machine (Pylint) and

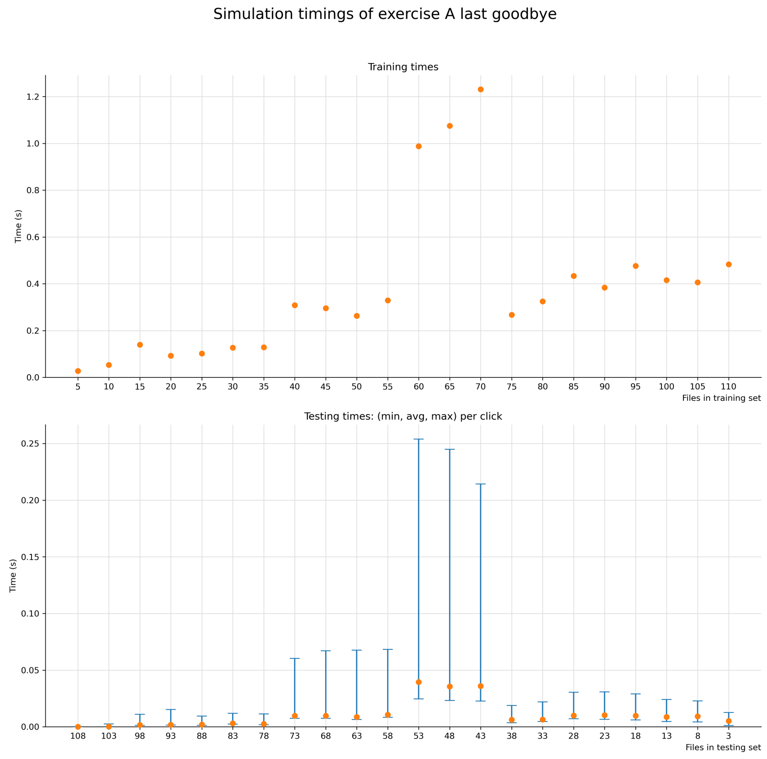


Figure 6.15: Time needed for training and testing during the entire review process for the exercise “A last goodbye”. Top: training time. Bottom: average (orange dot) and range (blue line) of time needed to predict a single instance.

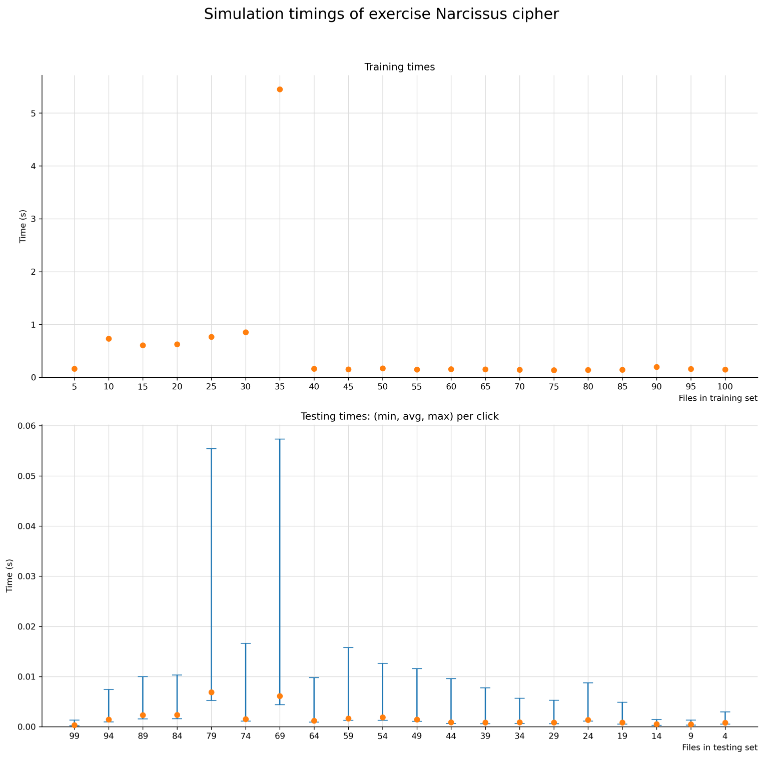


Figure 6.16: Time needed for training and testing during the entire review process for the exercise “Narcissus cipher”. Top: training time. Bottom: average (orange dot) and range (blue line) of time needed to predict a single instance.

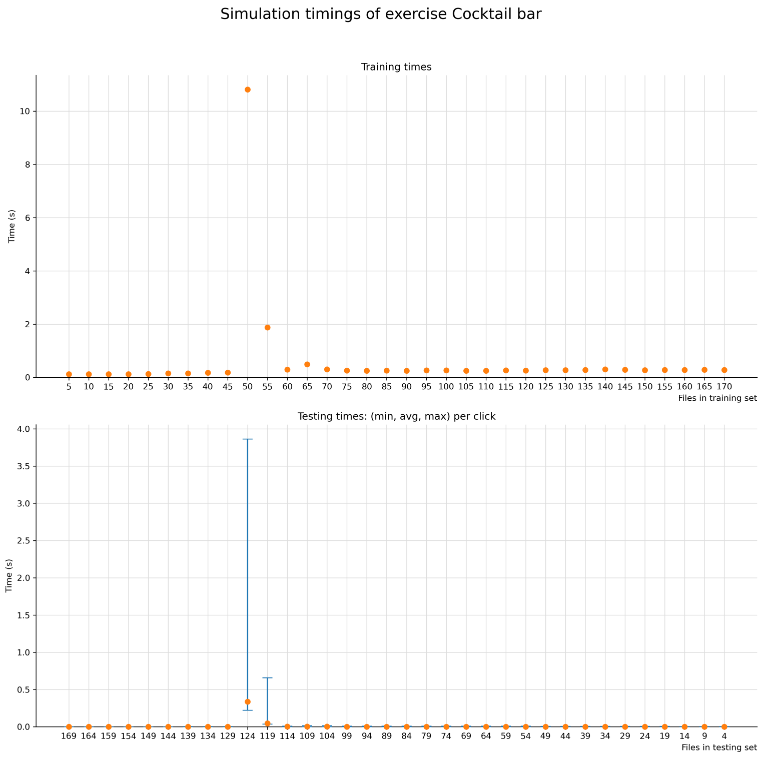


Figure 6.17: Time needed for training and testing during the entire review process for the exercise "Cocktail bar". Top: training time. Bottom: average (orange dot) and range (blue line) of time needed to predict a single instance.

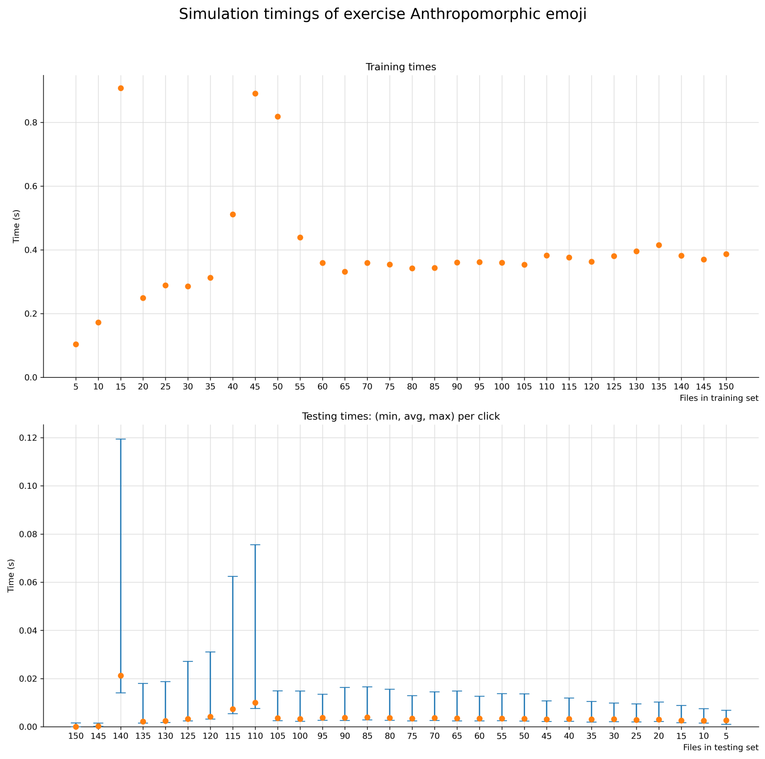


Figure 6.18: Time needed for training and testing during the entire review process for the exercise “Anthropomorphic emoji”. Top: training time. Bottom: average (orange dot) and range (blue line) of time needed to predict a single instance.

human annotations. Thus, we can answer both our research questions in an affirmative way, meaning that the reuse of feedback previously given by a human reviewer on a particular line of a new submission can be predicted with high accuracy (RQ1), and that this can be done fast enough to assist human reviewers in future reviews (RQ2).

Having ECHO at hand immediately raises some opportunities fo follow-up work. Currently, the proposed model is reactive: we suggest a ranking of the most likely annotations when a reviewer wants to add an annotation to a particular line of a submission. By introducing a confidence score, we could check beforehand whether we have a confident match for each line, and then immediately propose these suggestions to the reviewer. Whether or not a reviewer accepts these suggestions could then also be used as an input to the model. This could also have an additional benefit by helping reviewers to be more consistent in where and when they place annotations.

Annotations that do not lend themselves well to prediction also need further investigation. The context used could be expanded, although the important caveat here is that the method still needs to maintain sufficient performance. We could also consider applying some of the source code pattern mining techniques proposed by Pham et al. (2019) to achieve further speed improvements. This could help with the outliers seen in the timing data. Another important aspect that was explicitly outside of the scope of this chapter was the integration of ECHO into a learning platform and user testing.

Of course, alternative methods could also be considered. One can not overlook the rise of Large Language Models (LLMs) and the way in which they could contribute to this problem. LLMs can generate feedback for students based on their submitted solution and a well-chosen system prompt. Fine-tuning of an LLM with feedback already given is another possibility. Future applications could also combine user generated and LLM generated feedback, showing human reviewers the source of the feedback during their reviews.

7 Looking ahead: opportunities and challenges

It feels safe to say that Dodona is a successful automated assessment platform with a big societal impact. 70 000 users is quite a lot, and the fact that it is being actively used in a lot of higher education institutions and secondary schools in Flanders is a feat that not many other similar platforms have achieved.

As we have tried to show in this dissertation, its development has also led to interesting opportunities for new research. Dodona generates a lot of data, and we have shown that educational data mining can be used on this data. It can even be used to develop new educational data mining techniques that are applicable elsewhere. The work is, however, never finished. There are still possibilities for interesting computer science and educational research.

7.1 Research opportunities

A big question, left open in this work, is what to do with the results we obtained in Chapter 5. Teachers can use the results to figure out which aspects of their course students are struggling with, and take general measures to deal with this. But should we, and if so, *how* should we communicate predictions to individual students, or what other interventions with students should we take?

Chapter 6 also suggests a number of improvements that could still be worked on. It gives us a framework for suggesting the feedback a teacher probably wants to give when selecting a line, but we could also try to come up with a confidence score and use that to suggest feedback before the teacher has even done that. Another interesting (more educational) line of research that this work suggests is building the method into an actual assessment platform, and looking at its effects on feedback consistency and quality, time saved by teachers, …

A new idea for research using Dodona's data would be skill estimation. There are a few ways we could try to infer what skills are being tested by exercises: we could try to use the model solution, or the labels assigned to the exercise in Dodona. Using those skills, we could try to estimate a student's mastery of those skills, using their submissions. This would probably be done similarly to the research presented in Chapter 5 (using metrics like time-on-task). A skill profile would be more complicated though, since we would want some kind of vector to represent a student's progress in each estimated skill.

This leads right into another possibility for future research: exercise recommendation. Right now, learning paths in Dodona are static, determined by the teacher of the course the student is following. Dodona has a rich library of extra exercises, which some courses point to as opportunities for extra practice, but it is not always easy for students to know what exercises would be good for them. Using a skill profile, we could recommend exercises that only contain one skill the student has not fully attained, allowing them to focus their practice on that skill specifically. We would again need to infer what skills are tested by exercises, but this was already required for the skill estimation itself.

The research from Chapter 5 could also be used to help solve this problem in another way. If we know a student has a higher chance of failing the course, we might want to recommend some easier exercises. The other way around, if a student has a higher chance of passing, we could suggest more difficult exercises, so they can keep up their good progress in their course. Estimating the difficulty of an exercise is a problem unto itself though (and how difficult an exercise is, is also dependent on the student themselves).

The use of LLMs in Dodona could also be an opportunity. As mentioned in Section 6.5.4, a possibility for using LLMs could be to generate feedback while grading. By feeding an LLM with the student's code, an indication of the failed test cases (although doing this in a good format is an issue to solve in itself) and the type of issues that the teacher wants to address, it should be able to give a good starting point for the feedback. This could also kickstart the process explained in Section 6.5.4. By making generated feedback reusable, the given feedback can still remain consistent and fair.

Another option is to integrate an LLM as an AI tutor (as, for example, Khan Academy has done with Khanmigo[54]). This way, it could interactively help students while they are learning. Instead of tools like ChatGPT or Bard which are typically used to get a correct answer immediately, an AI tutor can guide students to find the correct answer to an exercise gradually by giving hints.

The final possibility we will present here is to prepare suggestions for answers to student questions on Dodona. At first glance, LLMs should be quite good at this. If we use the LLM output as a suggestion for what the teacher could answer, this should be a big time-saver. However, there are some issues around data quality. Questions are sometimes asked on a specific line, but the question does not necessarily have anything to do with that line. Sometimes the question also needs context that is hard to pass on to the LLM. For example, if the question is just “I don’t know what’s wrong.”, a human might look at the failed test cases and be able to answer the “question” in that way. As mentioned previously, passing on the failed test cases to the LLM is a harder problem to solve. The actual assignment also needs to be passed on, but depending on its size this might also present a problem given token limitations/cost per token of some models. Another important aspect of this research would be figuring out how to evaluate the quality of the suggestions.

7.2 Challenges for the future

Even though Dodona is a successful project with some exciting possibilities for research that can still be done, the project also faces some challenges.

The most important of these challenges is the sustainability of the project. Dodona was started in the spare time of some researchers. After a few years, there was somebody working on it full-time. However, the funding for a full-time developer was always, and still is, temporary. PhD students who can devote some of their time to it are attracted, grants are applied for (and sometimes granted), but there is no stable source of funding. We have the advantage that we can kindly make use of Ghent University’s data centre, resulting in very few operational costs. A full-time developer, which Dodona is big enough to need, is expensive though. This puts Dodona’s future in a precarious situation, where there is a constant need to look for new funding opportunities.

[54]`https://www.khanmigo.ai/`

As much as generative AI can be an asset for Dodona, it is also a threat. Most exercises in Dodona can be solved by LLMs without issues.[55] This has some troubling implications for Dodona. Students using ChatGPT or GitHub Copilot when solving their exercises, might not learn as much as students who do the work fully on their own (just like students who plagiarize have a lower chance of passing their courses, as seen in Chapter 5). Another aspect is the fairness and integrity of evaluations using Dodona. The case study in Chapter 3 details the use of open-book/open-internet evaluations. If students can use generative AI during these evaluations (either locally or via a webservice), and knowing that LLMs can solve most exercises on Dodona, these evaluations will test the students' abilities less and less, if students can use LLMs. The way to solve these issues is not clear. It seems like LLMs are here to stay, and just like the calculator is a commonplace tool these days, the same could be true for LLMs in the future.[56] Instead of banning the use of LLMs, teachers could integrate the use of them in their courses. On the other hand, when children first learn to count and add, they do not use calculators. The same might be necessary when learning to program: to learn the basics, students might need to do a lot of things themselves, to really get a feel for what they are doing.

[55]Or at least with some nudging.

[56]The IMEC digimeter (a yearly survey on technology use in Flanders) showed that 18% of Flemish people used generative AI at least monthly in 2023.

A Overview of Dodona releases

In this appendix, we give an overview of the most important Dodona releases, and the changes they introduced, organized per academic year. This is not a full overview of all Dodona releases, and does not mention all changes in a particular release.[57]

2015–2016

0.1 (2016-04-07) Minimal Rails app, where a list of exercises is shown based on files in the filesystem. This was only for JavaScript, and the code was executed locally in the browser.

0.2 (2016-04-14) Addition of a webhook to automatically update exercises. Assignments are rendered from Markdown, and can include media and formulas. Ace was introduced as the editor.

0.5 (2016-08-10) This is the first release supporting Python through server-side judging.

0.6 (2016-08-16) Judges can now be auto-updated through a webhook.

0.7 (2016-09-07) The concept of a series was introduced.

2016–2017

1.0 (2016-09-23) Dodona now runs on multiple servers, and series have gained a deadline.

1.1 (2016-09-28) Teachers can now configure boilerplate code per exercise.

1.2 (2016-10-10) The Python Tutor was added to Dodona.

[57] A full overview of all Dodona releases, with their full changelog, can be found at `https://github.com/dodona-edu/dodona/releases/`.

1.3 (2016-11-02) Hidden series using a token link were added. Users could now also download their solutions for a series.

1.4.6 (2017-03-17) Use the student's latest submission instead of their best submission in most places.

2017–2018

2.0 (2017-09-15) Introduction of the concept of a course administrator. Courses could also be set to hidden, and options for managing registration were added.

2.3 (2018-07-26) OAuth sign in support was added, allowing users from other institutions to use Dodona.[58]

2018–2019

2.4 (2018-09-17) Add management and ownership of exercises and repositories by users. Users with teacher rights could no longer see and edit all users.

2.5 (2018-10-26) Improved search functionality. Courses were now also linked to an institution for improved searchability.

2.6 (2018-11-21) Diffing in the feedback view was fully reworked (see Chapter 4 for more details).

2.7 (2018-12-04) The punchcard was added to the course page. Labels could now also be added to course members.

2.8 (2019-03-05) Submissions and their feedback were moved from the database to the filesystem.

2.9 (2019-03-27) Large UI rework of Dodona, adding the class progress visualization. This release also adds a page with Dodona's privacy policy.

2.10 (2019-05-06) Allow courses to be copied. Anonymous mode (called demo mode at the time) was added.

2.11 (2019-06-27) Introduction of dark mode. This release also adds the heatmap visualization.

[58]This is also the first release where I was personally involved with Dodona's development.

2019–2020

3.0 (2019-09-12) Dodona was made open source. Support for the R programming language was added around this time.

3.1 (2019-10-17) Exercise descriptions were moved to iframes. Diffing was further improved.

3.2 (2019-11-28) Fully reworks the exporting of submissions.

3.3 (2020-02-26) The exercise info page was added.

3.4 (2020-04-19) Allow adding annotations on a submission.

3.6 (2020-04-27) Add reading activities as a new assignment type.

3.7 (2020-06-03) This release adds evaluations to Dodona.

2020–2021

4.0 (2020-09-16) Q&A support got added in this release, along with LTI support.

4.3 (2021-04-26) Add the teacher rights request form.

4.4 (2021-05-05) Add grading to evaluations (as a private beta).

4.6 (2021-06-18) Featured courses were added in this release.

2021–2022

5.0 (2021-09-13) New learning analytics were added to each series. This release also includes the full release of grading after an extensive private beta.

5.3 (2022-02-04) A new heatmap graph was added to the series analytics.

5.5 (2022-04-25) Introduction of Papyros, our own online code editor.

5.6 (2022-07-04) Another visual refresh of Dodona, this time to follow the Material Design 3 spec.

2022–2023

6.0 (2022-08-18) Allow users to sign in with a personal Google or Microsoft account.

6.1 (2022-09-19) Allow reuse of annotations in evaluations.

6.8 (2023-05-17) Threading of questions was added.

2023.07 (2023-07-04) Introduction of monthly releases, whose contents are continuously deployed.

2023.08 (2023-08-01) Switch from dodona.ugent.be to dodona.be

2023–2024

2023.10 (2023-10-01) Annotation reuse is rolled out to all users.

2023.11 (2023-11-01) The Python Tutor is moved client-side.

2023.12 (2023-12-01) The feedback view was reworked, moving every context to its own card.

2024.02 (2024-02-01) Papyros now also has an integrated debugger based on the Python Tutor.

B Pass/fail prediction feature types

`subm` numbers of submissions by student in series

`nosubm` number of exercises student did not submit any solutions for in series

`first_dl` time difference in seconds between student's first submission in series and deadline of series

`last_dl` time difference in seconds between student's last submission in series before deadline and deadline of series

`nr_dl` number of correct submissions in series by student before series' deadline

`correct` number of correct submissions in series by student

`after_correct` number of submissions by student after their first correct submission in the series

`before_correct` number of submissions by student before their first correct submission in the series

`time_series` time difference in seconds between the student's first and last submission in the series

`time_correct` time difference in seconds between the student's first submission in the series and their first correct submission in the series

`wrong` number of submissions by student in series with logical errors

`comp_error` number of submissions by student in series with compilation errors

`runtime_error` number of submissions by student in series with runtime errors

`correct_after_5m` number of exercises where first correct submission by student was made within five minutes after first submission

`correct_after_15m` number of exercises where first correct submission by student was made within fifteen minutes after first submission

`correct_after_2h` number of exercises where first correct submission by student was made within two hours after first submission

`correct_after_24h` number of exercises where first correct submission by student was made within twenty-four hours after first submission

Bibliography

Akçapınar, G., Altun, A., & Aşkar, P. (2019). Using learning analytics to develop early-warning system for at-risk students. *International Journal of Educational Technology in Higher Education*, *16*(1), 40. https://doi.org/10.1186/s41239-019-0172-z

Akçayır, G., & Akçayır, M. (2018). The flipped classroom: A review of its advantages and challenges. *Computers & Education*, *126*, 334–345. https://doi.org/10.1016/j.compedu.2018.07.021

Ala-Mutka, K. M. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, *15*(2), 83–102. https://doi.org/10.1080/08993400500150747

Asai, T., Abe, K., Kawasoe, S., Sakamoto, H., Arimura, H., & Arikawa, S. (2004). Efficient Substructure Discovery from Large Semi-Structured Data. *IEICE TRANSACTIONS on Information and Systems*, *E87-D*(12), 2754–2763.

Asif, R., Merceron, A., Ali, S. A., & Haider, N. G. (2017). Analyzing undergraduate students' performance using educational data mining. *Computers & Education*, *113*, 177–194. https://doi.org/10.1016/j.compedu.2017.05.007

Baker, R. S., & Yacef, K. (2009). The State of Educational Data Mining in 2009: A Review and Future Visions. *Journal of Educational Data Mining*, *1*(1), 3–17. https://doi.org/10.5281/zenodo.3554657

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.-M., Pearce, J. L., & Prather, J. (2019). Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 177–210. https://doi.org/10.1145/3344429.3372508

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36. `https://doi.org/10.1145/1272848.1272879`

Berners-Lee, T., Cailliau, R., Groff, J.-F., & Pollermann, B. (1992). World-Wide Web: The Information Universe. *Internet Research*, *2*(1), 52–58. `https://doi.org/10.1108/eb047254`

Bernius, J. P., Krusche, S., & Bruegge, B. (2022). Machine learning based feedback on textual student answers in large courses. *Computers and Education: Artificial Intelligence*, *3*, 100081. `https://doi.org/10.1016/j.caeai.2022.100081`

Berry, R. E. (1966). Grader Programs. *The Computer Journal*, *9*(3), 252–256. `https://doi.org/10.1093/comjnl/9.3.252`

Binns, R., Van Kleek, M., Veale, M., Lyngs, U., Zhao, J., & Shadbolt, N. (2018). 'It's Reducing a Human Being to a Percentage': Perceptions of Justice in Algorithmic Decisions. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–14. `https://doi.org/10.1145/3173574.3173951`

Bishop, J., & Verleger, M. A. (2013). The Flipped Classroom: A Survey of the Research. *2013 ASEE Annual Conference & Exposition*, 23.1200.1–23.1200.18.

Black, P., & Wiliam, D. (1998). Assessment and Classroom Learning. *Assessment in Education: Principles, Policy & Practice*, *5*(1), 7–74. `https://doi.org/10.1080/0969595980050102`

Bloom, B. S., Engelhart, M. D., Furst, E., Hill, W. H., & Krathwohl, D. R. (1956). Handbook I: Cognitive domain. *New York: David McKay.*

Bonar, J. G., & Cunningham, R. (1988). *Bridge: Intelligent tutoring with intermediate representations.* University of Pittsburgh, Artificial Intelligence and Psychology.

Braden, R. T., & Perlis, A. J. (1965). *An introductory course in computer programming.* Clearinghouse for Federal Scientific and Technical Information, US.

Brooks, F., & Kugler, H. (1987). *No silver bullet.* April.

Brunsfeld, M., Hlynskyi, A., Qureshi, A., Thomson, A., Vera, J., Turnbull, P., Clem, T., Creager, D., Helwer, A., Rix, R., Hendrik van

Antwerpen, Kavolis, D., Davis, M., Ika, Tuấn-Anh Nguyễn, Massicotte, M., Brunk, S., Yahyaabadi, A., Hasabnis, N., … Edgar. (2024). *Tree-sitter/tree-sitter: V0.20.9*. Zenodo. `https://doi.org/10.5281/ZENODO.4619183`

Brusilovsky, P., & Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. *Journal on Educational Resources in Computing*, *5*(3), 6–es. `https://doi.org/10.1145/1163405.1163411`

Caswell, T., Henson, S., Jensen, M., & Wiley, D. (2008). Open Educational Resources: Enabling universal education. *International Review of Research in Open and Distributed Learning*, *9*(1), 1–11. `https://doi.org/10.19173/irrodl.v9i1.469`

Cervone, D. (2012). MathJax: A platform for mathematics on the Web. *Notices of the AMS*, *59*(2), 312–316.

Chatti, M. A., Dyckhoff, A. L., Schroeder, U., & Thüs, H. (2012). A reference model for learning analytics. *International Journal of Technology Enhanced Learning*, *4*(5-6), 318–331. `https://doi.org/10.1504/IJTEL.2012.051815`

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*, *41*(2), 121–131. `https://doi.org/10.1016/S0360-1315(03)00030-7`

Cortes, C., & Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, 273–297.

Daly, C., & Horgan, J. (2005). Patterns of plagiarism. *ACM SIGCSE Bulletin*, *37*(1), 383–387. `https://doi.org/10.1145/1047124.1047473`

Daud, A., Aljohani, N. R., Abbasi, R. A., Lytras, M. D., Abbas, F., & Alowibdi, J. S. (2017). Predicting Student Performance using Advanced Learning Analytics. *Proceedings of the 26th International Conference on World Wide Web Companion*, 415–421. `https://doi.org/10.1145/3041021.3054164`

Dawson, P. (2017). Assessment rubrics: Towards clearer and more replicable design, research and practice. *Assessment & Evaluation in Higher Education*, *42*(3), 347–360. `https://doi.org/10.1080/02602938.2015.1111294`

Debuse, J. C. W., Lawley, M., & Shibl, R. (2008). Educators' perceptions of automated feedback systems. *Australasian Journal of Educational Technology*, *24*(4). `https://doi.org/10.14742/ajet.1198`

De Ridder, W., Van Petegem, C., Dawyndt, P., & Mesuere, B. (2022). *Papyros: schrijven, uitvoeren en testen van Python-code in de browser*. Ghent University.

Dooley, J. (2011). *Software Development and Professional Practice*. Apress. `https://doi.org/10.1007/978-1-4302-3802-7`

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, *5*(3), 4–es. `https://doi.org/10.1145/1163405.1163409`

Downes, S. (2007). Models for Sustainable Open Educational Resources. *Interdisciplinary Journal of E-Learning and Learning Objects*, *3*(1), 29–44.

Dutt, A., Ismail, M. A., & Herawan, T. (2017). A Systematic Review on Educational Data Mining. *IEEE Access*, *5*, 15991–16005. `https://doi.org/10.1109/ACCESS.2017.2654247`

Edwards, J. M., Fulton, E. K., Holmes, J. D., Valentin, J. L., Beard, D. V., & Parker, K. R. (2018). Separation of syntax and problem solving in Introductory Computer Programming. *2018 IEEE Frontiers in Education Conference (FIE)*, 1–5. `https://doi.org/10.1109/FIE.2018.8658852`

Edwards, S. (2006). *Web-CAT : The Web-based Center for Automated Testing.*

Edwards, S. H., & Pérez-Quiñones, M. A. (2007). Experiences using test-driven development with an automated grader. *Journal of Computing Sciences in Colleges*, *22*(3), 44–50.

Engelbart, D. C., & English, W. K. (1968). A research center for augmenting human intellect. *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, 395–410.

Farrell, S., Reid, I., Orchard, D., Sankar, K., Moses, T., Edwards, E. N., Pato, J., Knouse, C., Cantor, O. S., & Platt, D. (2002). Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML). *Organization for the Advancement of Structured Information Standards (OASIS) Standard.*

Feldman, M. Q., Wang, Y., Byrd, W. E., Guimbretière, F., & Andersen, E. (2019). Towards answering "Am I on the right track?" automatically using program synthesis. *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, 13–24. https://doi.org/10.1145/3358711.3361626

Ferguson, R. (2012). Learning analytics: Drivers, developments and challenges. *International Journal of Technology Enhanced Learning*, *4*(5/6), 304–317.

Ferguson, T. S. (1982). An Inconsistent Maximum Likelihood Estimate. *Journal of the American Statistical Association*, *77*(380), 831–834. https://doi.org/10.1080/01621459.1982.10477894

Fincher, S. (1999). What are we doing when we teach programming? *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011*, *1*, 12A4/1-12A4/5 vol.1. https://doi.org/10.1109/FIE.1999.839268

Fonseca, I., Martins, N. C., & Lopes, F. (2023). A web-based platform and a methodology to teach programming languages in electrical engineering education – evolution and student feedback. *2023 32nd Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE)*, 1–3. https://doi.org/10.23919/EAEEIE55804.2023.10181316

Forisek, M. (2006). On the Suitability of Programming Tasks for Automated Evaluation. *Informatics in Education*, *5*(1), 63–76. https://doi.org/10.15388/infedu.2006.05

Forsythe, G. E., & Wirth, N. (1965). Automatic grading programs. *Communications of the ACM*, *8*(5), 275–278. https://doi.org/10.1145/364914.364937

Gibbs, G., & Simpson, C. (2005). Conditions Under Which Assessment Supports Students' Learning. *Learning and Teaching in Higher Education*, *1*, 3–31.

Glass, A. L., & Kang, M. (2022). Fewer students are benefiting from doing their homework: An eleven-year study. *Educational Psychology*, *42*(2), 185–199. https://doi.org/10.1080/01443410.2020.1802645

Goold, A., & Rimmer, R. (2000). Factors affecting performance in first-year computing. *ACM SIGCSE Bulletin*, *32*(2), 39–43. `https://doi.org/10.1145/355354.355369`

Gordon, J., Henry, P., & Dempster, M. (2013). Undergraduate Teaching Assistants: A Learner-Centered Model for Enhancing Student Engagement in the First-Year Experience. *International Journal of Teaching and Learning in Higher Education*, *25*(1), 103–109.

Graham, D., Black, R., & van Veenendaal, E. (2021). *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning.

Grgić-Hlača, N., Zafar, M. B., Gummadi, K. P., & Weller, A. (2018). *The Case for Process Fairness in Learning: Feature Selection for Fair Decision Making*. 11.

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for CS education. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 579–584. `https://doi.org/10.1145/2445196.2445368`

Hanks, B., Fitzgerald, S., McCauley, R., Murphy, L., & Zander, C. (2011). Pair programming in education: A literature review. *Computer Science Education*, *21*(2), 135–173. `https://doi.org/10.1080/08993408.2011.579808`

Hannebauer, C., Hesenius, M., & Gruhn, V. (2018). Does syntax highlighting help programming novices? *Empirical Software Engineering*, *23*(5), 2795–2828. `https://doi.org/10.1007/s10664-017-9579-0`

Hardt, D. (2012). *The OAuth 2.0 Authorization Framework* (Request for Comments RFC 6749). Internet Engineering Task Force. `https://doi.org/10.17487/RFC6749`

Hattie, J., & Timperley, H. (2007). The Power of Feedback. *Review of Educational Research*, *77*(1), 81–112. `https://doi.org/10.3102/003465430298487`

Hext, J. B., & Winings, J. W. (1969). An automatic grading scheme for simple programming exercises. *Communications of the ACM*, *12*(5), 272–275. `https://doi.org/10.1145/362946.362981`

Higgins, C., Hegazy, T., Symeonidis, P., & Tsintsifas, A. (2003). The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies*, *8*(3), 287–304. `https://doi.org/10.1023/A:1026364126982`

Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, *3*(10), 528–529. https://doi.org/10.1145/367415.367422

Hughes, R. J., Shewmake, J., & Okelberry, C. R. (1998). Ceilidh: Collaborative writing on the Web. *Proceedings of the 1998 ACM Symposium on Applied Computing*, 732–736. https://doi.org/10.1145/330560.331081

Hung, S.-L., Kwok, I.-F., & Chan, R. (1993). Automatic programming assessment. *Computers & Education*, *20*(2), 183–190. https://doi.org/10.1016/0360-1315(93)90086-X

Hunt, A. (1999). *The pragmatic programmer*. Pearson Education India.

Hylén, J. (2021). *Open educational resources: Opportunities and challenges.*

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 86–93. https://doi.org/10.1145/1930464.1930480

Insa, D., & Silva, J. (2018). Automatic assessment of Java code. *Computer Languages, Systems & Structures*, *53*, 59–72. https://doi.org/10.1016/j.cl.2018.01.004

Isaacson, P. C., & Scott, T. A. (1989). Automating the execution of student programs. *ACM SIGCSE Bulletin*, *21*(2), 15–22.

Jackson, D., & Usher, M. (1997). Grading student programs using ASSYST. *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, 335–339. https://doi.org/10.1145/268084.268210

Jessen, J. H. (2010). An overview of ESI storage & retrieval. *Sedona Conf. J.*, *11*, 237.

Karabenick, S. A., & Knapp, J. R. (1991). Relationship of academic help seeking to the use of learning strategies and other instrumental achievement behavior in college students. *Journal of Educational Psychology*, *83*(2), 221–230. https://doi.org/10.1037/0022-0663.83.2.221

Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., & Pausch, R. (2002). Alice2: Programming without syntax errors. *User Interface Software and Technology*, *2*, 35–36.

Keuning, H., Jeuring, J., & Heeren, B. (2018). A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education*, *19*(1), 3:1–3:43. https://doi.org/10.1145/3231711

Khalifa, M., & Lam, R. (2002). Web-based learning: Effects on learning process and outcome. *IEEE Transactions on Education*, *45*(4), 350–356. https://doi.org/10.1109/TE.2002.804395

Kleinbaum, D. G. (1994). Introduction to Logistic Regression. In D. G. Kleinbaum (Ed.), *Logistic Regression: A Self-Learning Text* (pp. 1–38). Springer. https://doi.org/10.1007/978-1-4757-4108-7_1

Kosowski, A., Małafiejski, M., & Noinski, T. (2008). Application of an Online Judge & Contester System in Academic Tuition. In H. Leung, F. Li, R. Lau, & Q. Li (Eds.), *Advances in Web Based Learning – ICWL 2007* (pp. 343–354). Springer. https://doi.org/10.1007/978-3-540-78139-4_31

Kovacic, Z. (2012). *Predicting student success by mining enrolment data.*

Laplante, P. A. (2007). *What every engineer should know about software engineering.* CRC Press.

Lebuda, I., & Karwowski, M. (2013). Tell Me Your Name and I'll Tell You How Creative Your Work Is: Author's Name and Gender as Factors Influencing Assessment of Products' Creativity in Four Different Domains. *Creativity Research Journal*, *25*(1), 137–142. https://doi.org/10.1080/10400419.2013.752297

Lee, A. V. Y. (2023). Supporting students' generation of feedback in large-scale online course with artificial intelligence-enabled evaluation. *Studies in Educational Evaluation*, *77*, 101250. https://doi.org/10.1016/j.stueduc.2023.101250

Leiba, B. (2012). OAuth Web Authorization Protocol. *IEEE Internet Computing*, *16*(1), 74–77. https://doi.org/10.1109/MIC.2012.11

Lienard, J., Mens, K., & Nijssen, S. (2023). Extracting unit tests from patterns mined in student code to provide improved feedback in autograders. *Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE).*

Livieris, I. E., Drakopoulou, K., Tampakas, V. T., Mikropoulos, T. A., & Pintelas, P. (2019). Predicting Secondary School Students' Performance Utilizing a Semi-supervised Learning Approach. *Journal of Educational Computing Research*, *57*(2), 448–470. https://doi.org/10.1177/0735633117752614

Maertens, R., Van Petegem, C., Strijbol, N., Baeyens, T., Jacobs, A. C., Dawyndt, P., & Mesuere, B. (2022). Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning*. https://doi.org/10.1111/jcal.12662

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, *10*(4), 16:1–16:15. https://doi.org/10.1145/1868358.1868363

Malouff, J. M., Emmerton, A. J., & Schutte, N. S. (2013). The Risk of a Halo Bias as a Reason to Keep Students Anonymous During Grading. *Teaching of Psychology*, *40*(3), 233–237. https://doi.org/10.1177/0098628313487425

Malouff, J. M., & Thorsteinsson, E. B. (2016). Bias in grading: A meta-analysis of experimental research findings. *Australian Journal of Education*, *60*(3), 245–256. https://doi.org/10.1177/0004944116664618

Mani, A., Venkataramani, D., Petit Silvestre, J., & Roura Ferret, S. (2014). Better feedback for educational online judges. *Proceedings of the 6th International Conference on Computer Supported Education, Volume 2: Barcelona, Spain, 1-3 April, 2014*, 176–183. https://doi.org/10.5220/0004842801760183

McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Mens, K., Nijssen, S., & Pham, H.-S. (2021). The good, the bad, and the ugly: Mining for patterns in student source code. *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, 1–8. https://doi.org/10.1145/3472673.3473958

Miller, J. C., & Maloney, C. J. (1963). Systematic mistake analysis of digital computer programs. *Communications of the ACM*, *6*(2), 58–63.

Mishra, D. S., & Edwards, S. H. (2023). The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 395–401. `https://doi.org/10.1145/3545945.3569734`

Molnar, C. (2019). *Interpretable Machine Learning*.

Moons, F., Vandervieren, E., & Colpaert, J. (2022). Atomic, reusable feedback: A semi-automated solution for assessing handwritten tasks? A crossover experiment with mathematics teachers. *Computers and Education Open*, *3*, 100086. `https://doi.org/10.1016/j.caeo.2022.100086`

Myers, E. W. (1986). AnO(ND) difference algorithm and its variations. *Algorithmica*, *1*(1), 251–266. `https://doi.org/10.1007/BF01840446`

Nandi, D., Hamilton, M., & Harland, J. (2012). Evaluating the quality of interaction in asynchronous discussion forums in fully online courses. *Distance Education*, *33*(1), 5–30. `https://doi.org/10.1080/01587919.2012.667957`

Naur, P. (1964). Automatic grading of students' ALGOL programming. *BIT*, *4*(3), 177–188. `https://doi.org/10.1007/BF01956028`

Newman, R. S., & Schwager, M. T. (1993). Students' Perceptions of the Teacher and Classmates in Relation to Reported Help Seeking in Math Class. *The Elementary School Journal*, *94*(1), 3–17. `https://doi.org/10.1086/461747`

Nidhra, S., & Dondeti, J. (2012). Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, *2*(2), 29–50.

Nievergelt, J. (1976). *ACSES: The Automated Computer Science Education System at the University of Illinois*.

Norvig, P. (2001). *Teach Yourself Programming in Ten Years*. http://norvig.com/21-days.html.

Nüst, D., Eddelbuettel, D., Bennett, D., Cannoodt, R., Clark, D., Daróczi, G., Edmondson, M., Fay, C., Hughes, E., Kjeldgaard, L., Lopp, S., Marwick, B., Nolis, H., Nolis, J., Ooi, H., Ram, K., Ross, N., Shepherd, L., Sólymos, P., … Xiao, N. (2020). The Rockerverse: Packages and Applications for Containerisation with R. *The R Journal*, *12*(1), 437–461. `https://doi.org/10.32614/RJ-2020-007`

Oberkampf, W. L., & Roy, C. J. (2010). *Verification and Validation in Scientific Computing.* Cambridge University Press.

OECD. (2021). *OECD Digital Education Outlook 2021* [Doi:https://doi.org/10.1787/589b283f-en].

O'Reilly, T. (2007). *What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software* (SSRN Scholarly Paper 1008839).

Paiva, J. C., Leal, J. P., & Figueira, Á. (2022). Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education*, *22*(3), 34:1–34:40. https://doi.org/10.1145/3513140

Paiva, J. C., Queirós, R., Leal, J. P., Swacha, J., & Miernik, F. (2022). Managing Gamified Programming Courses with the FGPE Platform. *Information*, *13*(2), 45. https://doi.org/10.3390/info13020045

Parihar, S., Dadachanji, Z., Singh, P. K., Das, R., Karkare, A., & Bhattacharya, A. (2017). Automatic Grading and Feedback using Program Repair for Introductory Programming Courses. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 92–97. https://doi.org/10.1145/3059009.3059026

Patton, E. W., Tissenbaum, M., & Harunani, F. (2019). MIT app inventor: Objectives, design, and development. *Computational thinking education*, 31–49.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*(85), 2825–2830.

Peveler, M., Maicus, E., & Cutler, B. (2019). Comparing Jailed Sandboxes vs Containers Within an Autograding System. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 139–145. https://doi.org/10.1145/3287324.3287507

Pham, H. S., Nijssen, S., Mens, K., Di Nucci, D., Molderez, T., De Roover, C., Fabry, J., & Zaytsev, V. (2019). Mining Patterns in Source Code Using Tree Mining Algorithms. In P. Kralj Novak, T. Šmuc, & S. Džeroski (Eds.), *Discovery Science* (pp. 471–480). Springer International Publishing. https://doi.org/10.1007/978-3-030-33778-0_35

Pintrich, P. R. (1995). Understanding self-regulated learning. *New Directions for Teaching and Learning*, *1995*(63), 3–12. https://doi.org/10.1002/tl.37219956304

Popham, W. J. (1997). What's Wrong–and What's Right–with Rubrics. *Educational Leadership*, *55*(2), 72–75.

Poulos, A., & Mahony, M. J. (2008). Effectiveness of feedback: The students' perspective. *Assessment & Evaluation in Higher Education*, *33*(2), 143–154. https://doi.org/10.1080/02602930601127869

Prince, M. (2004). Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, *93*(3), 223–231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x

Reek, K. A. (1989). The TRY system -or- how to avoid testing student programs. *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, 112–116. https://doi.org/10.1145/65293.71198

Romero, C., & Ventura, S. (2010). Educational Data Mining: A Review of the State of the Art. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *40*(6), 601–618. https://doi.org/10.1109/TSMCC.2010.2053532

Romero, C., & Ventura, S. (2020). Educational data mining and learning analytics: An updated survey. *WIREs Data Mining and Knowledge Discovery*, *10*(3), e1355. https://doi.org/10.1002/widm.1355

Romero, C., Ventura, S., & García, E. (2008). Data mining in course management systems: Moodle case study and tutorial. *Computers & Education*, *51*(1), 368–384. https://doi.org/10.1016/j.compedu.2007.05.016

Rountree, N., Rountree, J., Robins, A., & Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 101–104. https://doi.org/10.1145/1044550.1041669

Sakimura, N., Bradley, J., Jones, M., De Medeiros, B., & Mortimore, C. (2014). Openid connect core 1.0. *The OpenID Foundation*, S3.

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. *Proceedings of the 2003 ACM*

SIGMOD International Conference on Management of Data, 76–85. https://doi.org/10.1145/872757.872770

Schunk, D. H., & Zimmerman, B. J. (1994). *Self-regulation of learning and performance: Issues and educational applications.* Lawrence Erlbaum Associates, Inc.

Sels, B., Dawyndt, P., Mesuere, B., Strijbol, N., & Van Petegem, C. (2021). *TESTed: programmeertaal-onafhankelijk testen van oplossingen voor programmeeroefeningen: Eenvoudig oefeningen opstellen met een DSL.* Ghent University.

Shah, A. R. (2003). *Web-cat: A web-based center for automated testing.* Virginia Tech.

Staubitz, T., Klement, H., Renz, J., Teusner, R., & Meinel, C. (2015). Towards practical programming exercises and automated assessment in Massive Open Online Courses. *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, 23–30. https://doi.org/10.1109/TALE.2015.7386010

Stenerson, D., & Dawson, F. (1998). *Internet Calendaring and Scheduling Core Object Specification (iCalendar)* (Request for Comments RFC 2445). Internet Engineering Task Force. https://doi.org/10.17487/RFC2445

Stevens, W. P., Myers, G. J., & Constantive, L. L. (1999). Structured design. *IBM Systems Journal*, *38*(2.3), 231–256. https://doi.org/10.1147/sj.382.0231

Strickroth, S., & Holzinger, F. (2023). Supporting the Semi-automatic Feedback Provisioning on Programming Assignments. In M. Temperini, V. Scarano, I. Marenzi, M. Kravcik, E. Popescu, R. Lanzilotti, R. Gennari, F. De La Prieta, T. Di Mascio, & P. Vittorini (Eds.), *Methodologies and Intelligent Systems for Technology Enhanced Learning, 12th International Conference* (pp. 13–19). Springer International Publishing. https://doi.org/10.1007/978-3-031-20617-7_3

Strijbol, N., Dawyndt, P., & Mesuere, B. (2020). *TESTed: one judge to rule them all.* Ghent University.

Svetnik, V., Liaw, A., Tong, C., Culberson, J. C., Sheridan, R. P., & Feuston, B. P. (2003). Random Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling. *Journal* *of Chemical Information and Computer Sciences*, *43*(6), 1947–1958. https://doi.org/10.1021/ci034160g

Temperly, J. F., & Smith, B. W. (1968). A Grading Procedure for PL/1 Student Exercises. *The Computer Journal*, *10*(4), 368–373. https://doi.org/10.1093/comjnl/10.4.368

Tucker, B. (2012). The flipped classroom. *Education next*, *12*(1), 82–83.

Tuck, J. (2012). Feedback-giving as social practice: Teachers' perspectives on feedback as institutional requirement, work and dialogue. *Teaching in Higher Education*, *17*(2), 209–221. https://doi.org/10.1080/13562517.2011.611870

Tuomi, I. (2013). Open Educational Resources and the Transformation of Education. *European Journal of Education*, *48*(1), 58–78. https://doi.org/10.1111/ejed.12019

Van Petegem, C., & Dawyndt, P. (2018). *Computationele benaderingen voor deductie van de computationele complexiteit van computerprogramma's*. Ghent University.

Van Petegem, C., Maertens, R., Strijbol, N., Van Renterghem, J., Van der Jeugt, F., De Wever, B., Dawyndt, P., & Mesuere, B. (2023). Dodona: Learn to code with a virtual co-teacher that supports active learning. *SoftwareX*, *24*, 101578. https://doi.org/10.1016/j.softx.2023.101578

Van Petegem, P., De Loght, T., & Shortridge, A. M. (2004). Powerful Learning Is Interactive: A Cross-Cultural Perspective. *E-Journal of Instructional Science and Technology*, *7*(1).

Vasyliuk, A., & Lytvyn, T. B. V. (2023). Design and Implementation of a Ukrainian-Language Educational Platform for Learning Programming Languages. *Proceedings of the Modern Machine Learning Technologies and Data Science Workshop (MoMLeT&DS 2023)*, *3426*, 406–420.

Verhoeff, T. (2008). Programming Task Packages: Peach Exchange. *Olympiads in Informatics*, 192.

Vihavainen, A. (2013). Predicting Students' Performance in an Introductory Programming Course Using Data from Students' Own Programming Process. *2013 IEEE 13th International Conference on Advanced Learning Technologies*, 498–499. https://doi.org/10.1109/ICALT.2013.161

Vihavainen, A., Luukkainen, M., & Kurhila, J. (2013). Using students' programming behavior to predict success in an introductory mathematics course. *Educational Data Mining 2013*.

Vittorini, P., Menini, S., & Tonelli, S. (2021). An AI-Based System for Formative and Summative Assessment in Data Science Courses. *International Journal of Artificial Intelligence in Education*, *31*(2), 159–185. `https://doi.org/10.1007/s40593-020-00230-2`

Wagner, N. R. (2000). Plagiarism by student programmers. *The University of Texas at San Antonio Division Computer Science San Antonio, TX.*

Wasik, S., Antczak, M., Badura, J., Laskowski, A., & Sternal, T. (2018). A Survey on Online Judge Systems and Their Applications. *ACM Computing Surveys*, *51*(1), 3:1–3:34. `https://doi.org/10.1145/3143560`

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39–44. `https://doi.org/10.1145/2591708.2591749`

Werth, L. H. (1986). Predicting student performance in a beginning computer science class. *ACM SIGCSE Bulletin*, *18*(1), 138–143. `https://doi.org/10.1145/953055.5701`

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., Dunnington, D., Posit, & PBC. (2023). *Ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.*

Wiegers, K. E. (1996). *Creating a software engineering culture.* Pearson Education.

Wiley, D., Bliss, T. J., & McEwen, M. (2014). Open Educational Resources: A Review of the Literature. In J. M. Spector, M. D. Merrill, J. Elen, & M. J. Bishop (Eds.), *Handbook of Research on Educational Communications and Technology* (pp. 781–789). Springer. `https://doi.org/10.1007/978-1-4614-3185-5_63`

Wiliam, D. (2011). What is assessment for learning? *Studies in Educational Evaluation*, *37*(1), 3–14. `https://doi.org/10.1016/j.stueduc.2011.03.001`

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., Bouwman, J., Brookes, A. J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C. T., Finkers, R., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*(1), 160018. `https://doi.org/10.1038/sdata.2016.18`

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education*, *12*(3), 197–212. `https://doi.org/10.1076/csed.12.3.197.8618`

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

Yourdon, E., & Constantine, L. L. (1979). Structured design. Fundamentals of a discipline of computer program and systems design. *Englewood Cliffs: Yourdon Press.*

Zaki, M. (2005). Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, *17*(8), 1021–1035. `https://doi.org/10.1109/TKDE.2005.125`

Zhidkikh, D., Heilala, V., Van Petegem, C., Dawyndt, P., Järvinen, M., Viitanen, S., De Wever, B., Mesuere, B., Lappalainen, V., Kettunen, L., & Hämäläinen, R. (2024). Reproducing Predictive Learning Analytics in CS1: Toward Generalizable and Explainable Models for Enhancing Student Retention. *Journal of Learning Analytics*, 1–21. `https://doi.org/10.18608/jla.2024.7979`