Informatics in Education, 2023, Vol. 22, No. 2, 277–294

DOI: 10.15388/infedu.2023.15

Automatically Detecting Previous Programming Knowledge from Novice Programmer Code Compilation History

Erno LOKKILA[1,2], Athanasios CHRISTOPOULOS[2], Mikko-Jussi LAAKSO[2]

[1]*University of Turku, Faculty of Technology, Department of Computation Turku, Finland*
[2]*University of Turku, Faculty of Natural Sciences, Turku Research Institute of Learning Analytics Turku, Finland*
e-mail: eolokk@utu.fi, atchri@utu.fi, milaak@utu.fi

Received: May 2022

Abstract. Prior programming knowledge of students has a major impact on introductory programming courses. Those with prior experience often seem to breeze through the course. Those without prior experience see others breeze through the course and disengage from the material or drop out. The purpose of this study is to demonstrate that novice student programming behavior can be modeled as a Markov process. The resulting transition matrix can then be used in machine learning algorithms to create clusters of similarly behaving students. We describe in detail the state machine used in the Markov process and how to compute the transition matrix. We compute the transition matrix for 665 students and cluster them using the k-means clustering algorithm. We choose the number of cluster to be three based on analysis of the dataset. We show that the created clusters have statistically different means for student prior knowledge in programming, when measured on a Likert scale of 1–5.

Keywords: machine learning, higher education, programming skill.

1. Introduction

Programming education is often started already in primary school in various ways. This education given from kindergarten to the 12th grade is called K-12 and involves teaching primary school children programming concepts such as basic control structures, often with graphical programming (e.g., Naz *et al.*, 2017). Computational and algorithmic thinking (Grover and Pea, 2013) are often covered as well. Because of this, programming courses given at the university level often face the problem of possibly drastically different beginning skill levels in programming (Strong *et al.*, 2017).

The terminology for student skill levels in this paper is as follows: *Beginners* are those who are completely new to programming. *Novice* programmers are those who have some experience with programming, but still struggle occasionally when creating small programs. Novice programmers are sometimes called ineffective novices (Robins *et al.*, 2003). *Advanced novices* are "someone who no longer makes the basic novice error such as missing guards or input/output statements" (Yarmish and Kopec, 2007). Robins *et al.* (2007) call this type of novice effective novices. The different levels of knowledge and needs induce several problems, both in course design as well as student motivation-wise.

Firstly, the course structure must be designed such that all levels of novice programmers, as well as absolute beginners find the course engaging and become better programmers. On one hand, if the course is catered towards the advanced novices, the beginners might lose motivation and drop out. On the other hand, if the course is catered towards the absolute beginners, the advanced novices feel the course has nothing to offer them and effectively waste their time. Secondly, the different skill levels present on the student cohort may possibly lead to a situation, where the beginners notice the advanced novices performing better on the course, thus causing the beginners feelings of self-doubt and loss of motivation.

In light of these problems, identifying those students with prior programming knowledge is crucial for introductory programming courses. Traditionally this has been done with pre-course surveys (Strong *et al.*, 2017; Smith *et al.*, 2019). However, the problem with surveys is that not everyone answers them. Additionally, of those who answer, a percentage may not answer truthfully. Thus, being able to automatically determine students with prior programming knowledge from their course activities (in this case, programming) is vital to the educator, who can then give students appropriately challenging exercises.

This study sets out to contribute in the following ways:

1. Provide a model to cluster students solely based on their submission history from small coding assignments.
2. Prove that the produced clusters are distinct in terms of previous programming knowledge
3. Demonstrate that the found clusters can be labeled by data inferred from the submission history alone, without requiring surveys or other additional data.

2. Related Work

Machine learning algorithms in clustering students has been on the rise since the early 2000s. Lahtinen (2007) clustered novice programmers using an administered test. She was able to find six distinct clusters: competent, practical, unprepared, theoretical, memorizing and indifferent students. The clusters were discovered using the k-means clustering algorithm, which was first described in MacQueen (1967).

Novais *et al.* (2017) introduced a tool they call the Programmer Profiler (PP). The tool only analyzes Java code and attempts to determine the programmer's skill level

from static features found from the code. They defined four distinct types of programmers: novice, advanced beginner, expert, and proficient. The PP does not use machine learning methods, instead is searching for static features regarding skill (ability to write code) and readability (coding convention) and scores them according to a predefined ruleset.

Rubio *et al.* (2019) used a variation of k-means clustering, called k-medioids clustering, wherein the cluster centroids are created from the median of the data points instead of the means. Instead of a completely data driven and automatic clustering, they administered a test and created the clustering based on the results of the test. The optimal number of clusters was discovered to be three. Three was also deemed the best number of clusters best fit for clustering students on a MOOC (Moubayed, 2020).

Jian *et al.* (2022) used machine learning methods, namely the Ward agglomerative clustering algorithm, to create a dendrogram from which four clusters were deemed optimal. The found clusters were representative of different styles of approaching problems: quitters, approaches, solvers, and knowers. The clustering was based on a history of edits to the program code, computed from an abstract syntax tree representation of the code.

Eloy *et al.* (2022) describe a data driven model using students' scratch projects to determine their determine their programming ability. They performed a static analysis of the code to quantify the scratch code's seven coefficients (such as events, loops, conditionals and operators). While the model was not used to directly determine students' previous programming experience, the same model may well be applied to this task. The goal of Eloy *et al.* was to quantify the computational thinking abilities of the student. They achieved this by applying a k-nearest neighbors clustering algorithm to the coefficients computed for each student. They were able to determine a typical project for each edition of the course, given on different years to different students.

Previous programming experience has been proven to have an effect on the perception and performance of students on introductory programming courses (Nowaczyk, 1984; Hagan and Markham, 2000; Gjelsten *et al.*, 2021). Hagan and Markham (2000) show that prior programming experience is best defined as the number of programming languages studied or used, whereas no significance was found in the number of language paradigms known. However, Ahadi *et al.* (2015) built a predictive model using a random forest and found that previous programming experience is not a significant feature when predicting students' final grade on introductory programming courses. However, their dataset consisted of answers to programming exercises, which naturally offer more information to the machine learning algorithm. Conversely, Lister *et al.* (2010) demonstrate that previous programming experience correlates strongly with tracing code.

Prior programming knowledge has traditionally been identified in students with surveys (Strong *et al.*, 2017; Smith *et al.*, 2019). More modern approaches to prior programming experience identification is using the keypresses produced by students in the code editor (Leinonen *et al.*, 2016). Prior experience is often used as a predictive tool for the student's course grade. The effect of prior knowledge is not straightforward, as some studies (Nowaczyk, 1984; Hagan and Markham, 2000; Gjelsten *et al.*, 2021) show significance, whereas others show no significance (Bergin and Reilly, 2005).

In addition to being used mainly in predicting course grades, identifying prior programming knowledge also allows pedagogical affordances in course design. One important such method is differentiation, wherein students of differing skill level are offered exercises suitable to their skill level (Mok, 2011). This is in hopes of reaching and placing the student firmly in their zone of proximal development (ZPD). The ZPD is the contextual space, wherein the student's proficiency increases (Vygotsky, 1978).

Proficiency in natural languages consist of four main skills: reading, writing, speaking and listening (Sadiku, 2015). It is not difficult to think of analogies in the programming world for the first two, as they are the written skills. The latter two can be thought of as speaking and listening to other programmers. Novice programmers lack all these skills, and must learn them. Venables *et al.* (2009) suggest there is a hierarchy in the acquisition of these skills. Nevertheless, novices perform poorly in tracing code (Lister *et al.*, 2004; Vainio and Sajaniemi, 2007). They are unable to write programs effectively (Perkins *et al.*, 1986). They are also unable to explain verbally what a piece of code does (van der Werf, 2021). To our knowledge, no research has been done on how beginning programmers understand problems when explained by another programmer, but presumably they are unable to effectively do so.

Most research on measuring programmer skill has been done on measuring the skill of writing code. Metrics such as the Error Quotient (Jadud, 2006) and the Watwin score (Watson *et al.*, 2013) have been used to measure the code producing capabilities of students. The other skills rarely produce artefacts which could be studied, thus different methods must be used, such as interviews or analyzing video recordings.

3. Research Methodology

The work is a continuation of the authors work from Lokkila *et al.* (2022), which applied the same method to determine engagement with course material from the submission histories of students in introductory programming courses. The basis for the model described in this paper is a transition matrix, also known as a right stochastic matrix (Asmussen, 2003). The matrix is built from a state machine, by calculating the probability of moving from one state to another. This matrix is unique to every student and can be used with machine learning algorithms to e.g., create clusters from the matrices.

The state machine contains eight different states, and the states are connected as a directed graph (Fig. 1). The states themselves are as follows:

1. **Met success.** This state is entered on the first successful attempt after an error. The student continues to work on the exercise.
2. **Met error.** This state is entered on the first compilation error after a successful attempt.
3. **Repeated error.** This state is entered when the student receives a compilation error after one or more compilation errors
4. **Repeated success.** This state is entered when the student submits non-erroneous code after one or more non-erroneous code submissions.

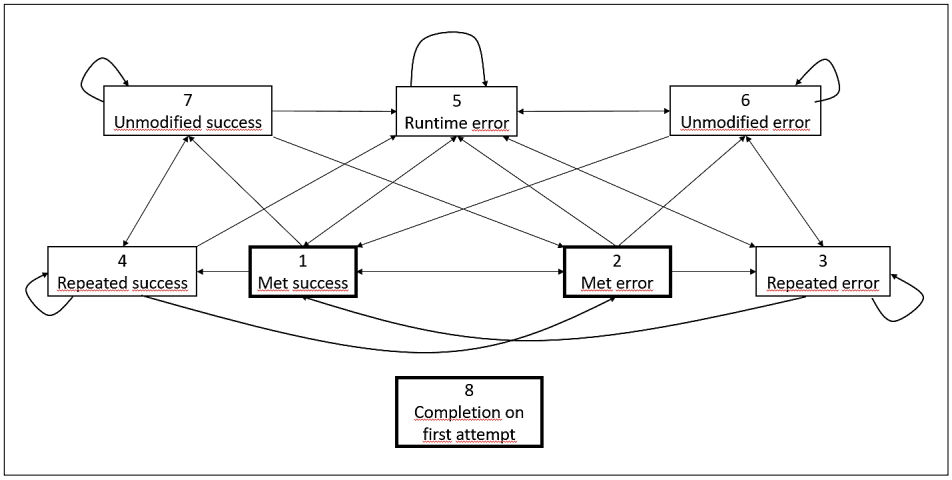

Fig. 1. The state machine. Start states are indicated with stronger borders.

5. **Runtime error.** This state is entered whenever the code begins to execute, but fails to complete successfully.
6. **Unmodified error submission.** Student resubmits the previous non-working code with no changes.
7. **Unmodified success submission.** The student resubmits the previous working code with no changes.
8. **Completion on first attempt.** The student successfully completes the programming assignment on the first submission with full points and moves to a different exercise.

The algorithm for creating a transition matrix for a student is as follows:

1. **Collate**. Consider all n submissions from a given student. Order then by date. Label them s_1 to s_n. Then, build the list L of n − 1 pairs such that
   $L = [(s_1, s_2), (s_2, s_3), \ldots, (s_{n-1}, s_n)]$.
2. **Aggregate**. For each pair in the list L, determine which state transition occurred and increase the count in the corresponding cell in the transition matrix, T.
3. **Normalize**. For each row in T, divide the vales in the cells by the sum of the row.
4. **Finalize** Compute the ratio of successes on first attempt to all exercises attempted and store this value for state 8.

Acceptable exercises for this model are those wherein the student has written code. The exercises on the courses were short code writing exercises, where the student wrote methods according to specification. The course and its exercises are described in more detail in Section 4. In case of exercises containing multiple files, step one should be done for each individual file. The output of the algorithm is the transition matrix for the given student. This method is completely data driven, as its accuracy increases with more data. The accuracy of the transition matrix here refers to its capability of predicting the next

state transition the student will take, given a starting state. Essentially, a transition matrix models the studied behavior as a Markov chain (Gagniuc, 2017).

The transition matrix is perfectly suited for machine learning applications. It produces a 29-dimensional variable space, which we only cluster in this study. The variables are the state transition possibilities from the transition matrix itself, as well as the eighth state (completion on first attempt). Because only 29 variables are involved, we sidestep the curse of dimensionality (Bellman, 1966).

An additional data point can be computed from the transition matrix itself. By labeling states 1, 4, 7 and 8 as success states and the other states as error states, the transition matrix will contain four distinct transitions: Error to error (EE), Error to success (ES), Success to error (SE) and Success to success (SS). Table 1 contains the adjacency mtatrix for the state machine as well as the transitions between error and non-error states. Then, by summing all transition probabilities to erroneous states and subtracting that from the sum of the transition probabilities to error-free states (including state 8), we have derived a simplistic value for the skill level of the student. We call this value the Syntactic Score. If we denote the transition matrix as T, the transitions ending in a success state (columns 1, 4, 7 and state 8) as P and the transitions ending in an error state (columns 2, 3, 5 and 6) as N, we arrive at the following formula for calculating the Syntactic Score:

$$SS = \sum P - \sum N$$

The Syntactic Score is an estimate of how likely the student is to create working code. It is a rough metric on how well a student is likely to perform: a strong negative correlation exists with student submission amounts (Spearman's correlation coefficient $\rho = -.68$, $p < .001$), as well as error rates of the student (Spearman's correlation coefficient $\rho = -.72$, $p < .001$). This means that high values for the Syntactic Score mean they are more likely to create working code with less attempts, whereas low values indicate a student who struggles, as they need more attempts and create more errors to create working code.

Table 1

The adjacency matrix for the state machine behind the transition matrix. States are classified either as a success state (S) or an error state (E). Transitions happen along the rows

	1 (S)	2 (E)	3 (E)	4 (S)	5 (E)	6 (E)	7 (S)
1 (S)		SE		SS	SE		SS
2 (E)	ES		EE		EE	EE	
3 (E)	ES		EE		EE	EE	
4 (S)		SE		SS	SE		SS
5 (E)	ES		EE		EE	EE	
6 (E)	ES		EE		EE	EE	
7 (S)		SE		SS	SE		SS

The value for the Syntactic Score is bound in the range [−1,2], but typical values are between [0,1]. The value of negative one can be achieved only by students who never succeed in creating error-free code. Values above one can only be achieved if a student never does a single error and succeeds in most of the exercises on the first attempt (i.e. state 8 in the state machine). Notably, in the case of the student succeeding immediately, the Syntactic Score remains exactly one. In order for the Syntactic Score to go above one, the student must succeed in most exercises with the first attempt and only create non-erroneous code in the remaining exercises. In this study, the Syntactic Score is clipped to the range [−1,1], as all values over one mean the student never submits erroneous code.

4. Data

The data was collected at the University of Turku and consists of two courses: an introduction to Java, given in fall 2017 as well as an introduction to Python, given in fall 2021. The dataset contains a total of 665 students and 376 475 submissions to 192 coding exercises. The data is described in more detail in Table 2. The courses also contained non-coding exercises, such as multiple-choice questions and Parsons problems, which were excluded from this dataset. The students were also given a pre-course questionnaire, which asked their previous programming experience on a Likert scale of 1 to 5 as well as a list of any programming languages they had used to write more than 200 lines of code. For the Java course, the mean and median self-reported prior programming experience was 2.326 and 2, respectively. For the Python course, the mean and median values were 1.821 and 2. Standard deviations were 1.018 for Java and 0.941 for Python. Thus, on average, the Java course had a slightly higher mean in self-estimated programming knowledge than Python. However, the medians of the courses were the same at 2 (on a scale of 1 to 5).

The courses were given using ViLLE (Laakso *et al.*, 2018). The course contained multiple types of questions, such as multiple-choice questions, Parsons problems, visualization exercises and short coding exercises. The short coding exercises on the course tasked the student to complete a method (or several) according to the problem statement. Most of the exercises offered a template, which included the calls to the student-written methods, so students had context on how the methods were used (Fig. 2). Some exercises only gave the student a blank coding area, with no template, to create code according to specification. We made no distinction between these two types of programming exercises in the data.

Table 2
Description of the data used in the study

Course	Language	Students	Submissions	Assignments
Course 1	Java	279	105 356	53
Course 2	Python	386	271 119	139

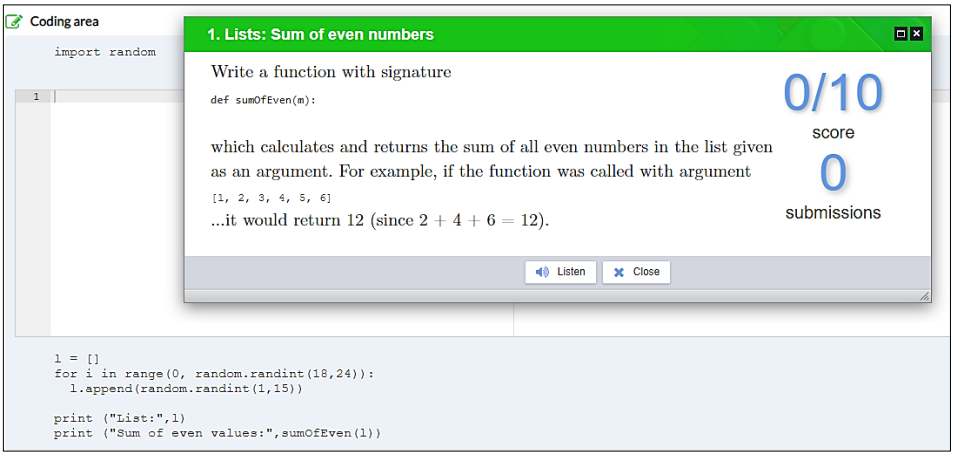

Fig. 2. An example exercise where the student codes according to specification.

The correctness of the code submitted by the student is done by comparing the output of the model answer and the student's code. Each submission collected by the system contains the following data points: Timestamp, the student's code, maximum points for the exercise and points awarded to the student. From these data points, further data can be inferred, such as any compilation errors or whether the submission contained an infinite loop. The Learning Management System detected infinite loops by giving the program two seconds of run time, after which the program was forcefully terminated.

The course structure was remarkably similar between both courses. The Python course focused slightly more on the practicalities of programming, whereas the Java course was slightly more theoretical. However, both courses aimed to bring students to the advanced novice level. This means that at the end of the course, the students were expected to be able to write small programs that solve an immediate need, such as filtering a text file for specific information.

Both courses begun with basic programming concepts: variables and control structures. The Java course went deeper into the theoretical aspects of computer science in

Table 3
The weekly themes for both courses

	Java	Python
Week1	Introduction	Intro, selection
Week2	Variables and selection	Loops
Week3	Loops	Methods
Week4	Methods	Data structures (list)
Week5	Data structures (Arrays)	Data structures (dict)
Week6	Existing classes (Lists, StringBuilder, BigInteger, etc.)	Files and exceptions
Week7	Revision	Libraries and 'pythonese' (slices, list comprehensions)

terms of how data is stored in memory, as well as the stack and the heap. The Python course was more practical and the focus was on how the demonstrated constructs are used when programming. The Java course was given before the pandemic, and utilized weekly in-person pair programming sessions called tutorials. The Python course was given during the pandemic, so such sessions were not possible. However students were offered virtual office hours, when they were able to ask for assistance from the course staff. The weekly topics are summarized in detail in Table 3.

5. Results

The transition matrices for each student on a programming course were used as input for unsupervised machine learning. This study chose the k-means clustering algorithm. In order to determine the correct number of clusters, we computed the inertia of different number of clusters (Fig. 3, left frame). There was no apparent elbow joint or other sharp change in the derivative of the plotted inertia, but the rate of change seems to slow down after around 10 clusters. This seems to suggest a suitable number of clusters to be less than ten. In order to determine the number of clusters more accurately, we also computed the silhouette score for a number of clusters under 10. Namely the score was computed for 2, 3, 6 and 8 clusters (Fig. 3, right frame).

The silhouette score describes the fit for each data point within the cluster. Thus, clusters with a sharp drop indicate a distinct cluster, whereas a cluster with a gentle slope indicate a cluster with more diverse data points. Negative scores indicate a possibly misclustered data point. Based on the inertia and silhouette score, we choose the number of clusters to be three. For the Python course, this produces one distinct cluster and two other clusters with a fuzzier border. The java clusters are more homogenous. Three clusters are also easily explainable: one cluster of capable students, one of average students, and one of struggling students.

We clustered the transition matrices created for each student on both courses separately. We used the k-means clustering algorithm with k = 3, which was deemed the most

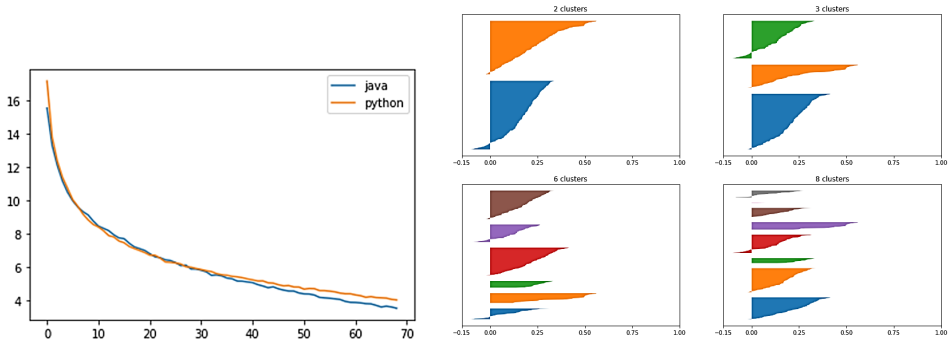

Fig. 3. The inertia of the clusters on the left and silhouette score (Python course) on the right.

appropriate number of clusters based on cluster inertia and silhouette score analysis. After clustering, we had clusters C1, C2 and C3. We then computed the average self-reported prior programming experience for each cluster. We denote C1 as the group with the least (or none) prior programming experience, C2 with some prior experience and C3 with the most (Fig. 4).

For the Python course, the clustering was indeed as indicated by the silhouette score (Fig. 3, right frame). One cluster was distinct and the other two were more similar; the distinct cluster was the one with the least prior programming experience. The Java course clustering had a similar trend, but the differences were not as dramatic.

To determine whether the clustering had produced statistically distinct clusters, we performed a one-way analysis of variance on the created clusters. We applied the Kruskal-Wallis H-test, as the data was not normally distributed (Shapiro-Wilk $W = 0.87$, $p < .001$). The non-normality of the data lead us to apply the Kruskal-Wallis test by ranks. Statistically significant differences ($p < .05$) were found for all weeks except the two last weeks. To determine specifically which groups differed in their prior programming knowledge, we did a post-hoc Dunn's test.

For both courses, we found statistically significant differences in means of prior programming experience between clusters for weeks 1 through 5 (Table 4). After the fifth course week, the statistical differences vanish between clusters. For the weeks with differences, the comparison between the weakest cluster and the strongest cluster was always significantly different, albeit the last two weeks were nearly significant. An exception to this is the sixth week of the Python course, where the clustering produced practically no differences. We postulate this is because the week's theme – files and exceptions – is conceptually difficult, so all students struggle equally.

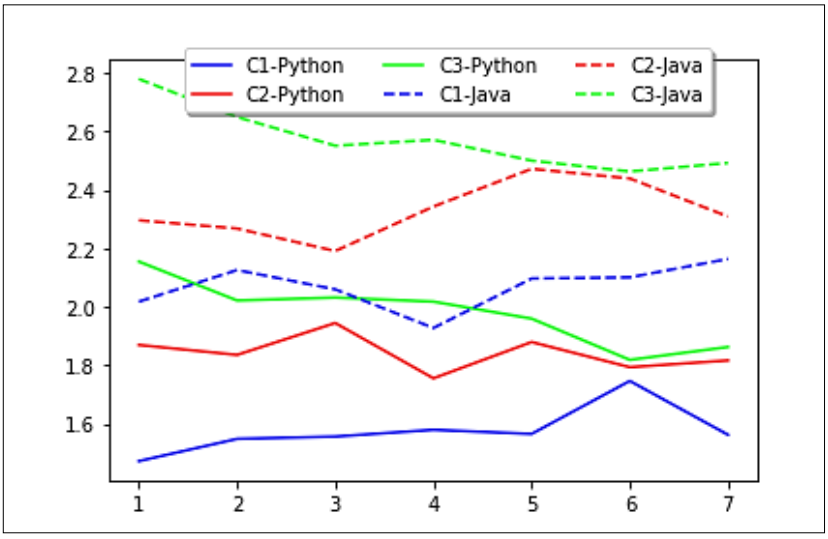

Fig. 4. The mean of prior programming knowledge for each weekly cluster.

Table 4

p-values of cluster-wise comparisons of prior programming knowledge using Dunn's test with Bonferroni correction. Significance ($p < .05$) in bold

	Java			Python		
Comparison	C1, C2	C1, C3	C2, C3	C1, C2	C1, C3	C2, C3
Week 1	.361	**<.001**	.108	**<.001**	**<.001**	.053
Week 2	.793	**.008**	.161	**.026**	**<.001**	.088
Week 3	.883	**.002**	.062	**<.001**	**<.001**	1
Week 4	**.041**	**.005**	1	.151	**<.001**	.06
Week 5	**.006**	**.031**	1	**.013**	**.003**	1
Week 6	.103	.18	1	1	1	1
Week 7	1	.096	.826	.108	.322	1

A possible explanation for these results is that the subject matter of the courses progressively increases in difficulty. At some point any prior programming knowledge held by students will have been covered by the course material and all further material will be new for all students. This is especially apparent in week 6 of the Python course, wherein the difference between the two strongest clusters effectively disappears when the course material moves to files, exceptions and 'pythonese', which here includes list comprehensions, slicing and other ways of writing idiomatic Python code.

We found significant differences between the weakest cluster and the strongest for all weeks except the last two. This implies that the proposed method is indeed capable of clustering students into clusters, one of which is the group of students without any prior programming knowledge and two groups wherein students have at least some prior experience in programming. We identified the clusters by their mean of self-reported programming experience. One question thus remains: how to identify the clusters without data from a survey?

We provide two ways of identifying the clusters: the number of submissions and the Syntactic Score. C1, being the cluster with the least programming experience, will have the lowest Syntactic Score and highest average number of submissions. C3, being the cluster with the most programming experience, will have the highest Syntactic Score and lowest number of average submissions. C2 will have values between these two clusters (Fig. 5).

While these two metrics generally agree with self-reported previous programming skill, several anomalous weeks exist: Week3 (Java), Week6 (Java) and the last week for both courses.

Week3 in Java (loops) is odd, because C2 performed the worst in terms of the Syntactic Score. This may, however, be explained by the fact that most students on the Java course had reported their previous programming experience to be in Python, which has markedly different for-loops. This forced the students in C2 to learn the syntax of the Java for-loop and do all the mistakes involved in the learning process in addition to 'unlearning' the Python syntax. Possibly, this anomaly may also be attributable to the Dunning-Kruger effect, where initiated novices overestimate their skills (Dunning, 2011).

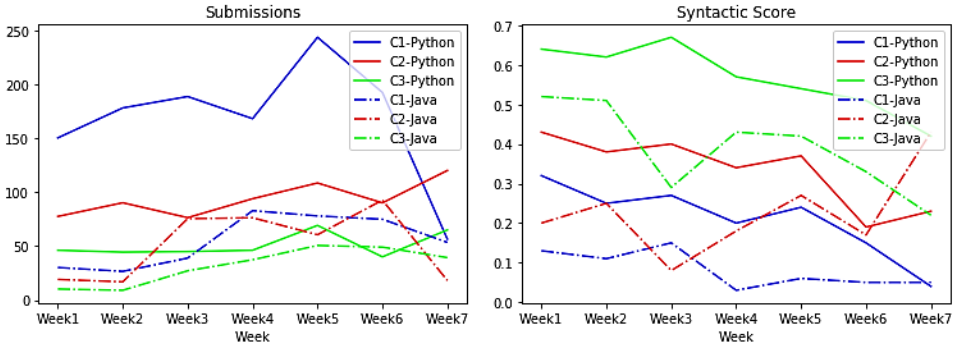

Fig. 5. The average submissions (left) and Syntactic Score (right) for each weekly cluster.

Week6 (existing classes) in Java was anomalous in that C2 made the most submissions on average. We believe this was the point where the students' existing skill ran out and they were unable to apply their knowledge from Python to the Java course. Another possible explanation is that the weakest cluster, C1, felt the topic too difficult for them and did not even attempt to do the exercises for that week.

Week7 shows anomalous values across the board when compared to the previous weeks. We believe this is because the students did not put as much effort into the remaining programming exercises. As week7 is the week before exams, other coursework and exam preparations evidently were prioritized over the programming exercises.

We performed our analysis on clusters generated from each week's data. To determine the stability of the clusters, we followed the procedure outlined by Henning (2007), which uses the Jaccard coefficient as a similarity measure between clusters. To compute the similarity, we computed the Jaccard coefficient for each cluster for one week and the next one, found the most similar one and then computed the average of the found values. We found the clusters to be relatively stable, with nearly 40% of the students remaining in the same cluster from one week to the next for the Python course and nearly one third for Java (Table 5).

We also examined how reliable only the students' own declaration of known programming languages is in terms of prior programming knowledge. We first checked the data for internal validity, using Cronbach's alpha and found it to be good for both the Python course ($\alpha = 0.834$) and the Java course ($\alpha = 0.735$). We then analyzed the correlation between self-reported prior programing knowledge and self-reported number of languages proficient in. We found the number of previous programming languages used had a medium correlation with the self-reported general programming experience the

Table 5

Average similarities of clusters between consecutive weeks

Weeks	1 & 2	2 & 3	3 & 4	4 & 5	5 & 6	6 & 7
Python	39.7 %	36.1 %	38.7 %	40.4 %	37.6 %	29.3 %
Java	29.7 %	27.4 %	26.6 %	30.8 %	27.4 %	29.4 %

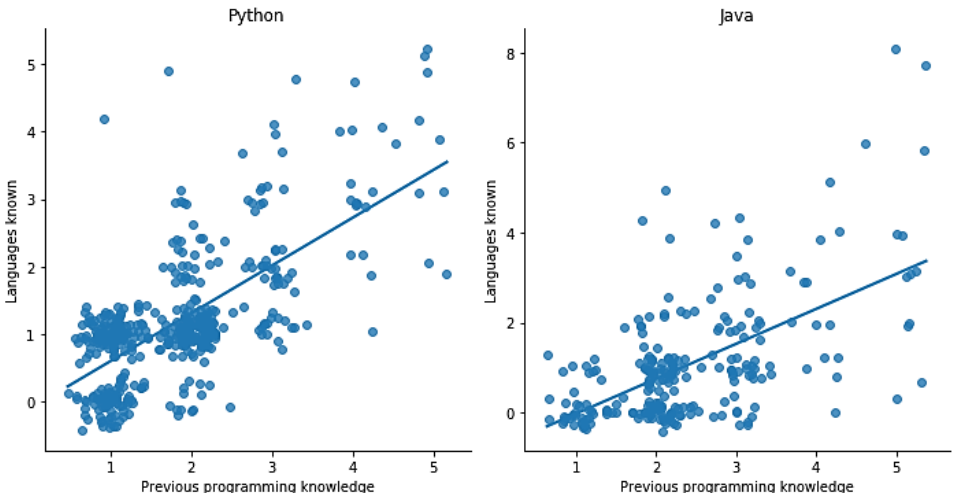

Fig. 6. A scatterplot (with added jitter) of students' previous programming knowledge to the number of languages they are familiar with.

Table 6

Average number of self-reported programming languages for clusters, when clustered using whole course data

	Java	Python
C1	0.631	0.936
C2	0.991	1.162
C3	2.289	1.486

students had on the Java course (Spearman's correlation coefficient $\rho = 0.552$, $p < .001$) and a stronger correlation on the Python course (Spearman's correlation coefficient $\rho = 0.65$, $p < .001$). Most of the students had little to no previous programming experience (Fig. 6).

Because the number of programming languages and previous programming experience were strongly correlated, our model managed to capture also this aspect (Table 6). This gives indications that a very rough estimate of students' programming skill can be achieved by merely asking how many languages the student has experience with.

6. Discussion

We interpret our results as a success in creating a clustering using the k-means clustering algorithm. The generated clusters, C1, C2 and C3 were indeed distinct in terms of self-reported previous programming experience; C3 had the highest mean, C1 the lowest and C2 between the other two. We also provided two ways to identify these clusters: 1) from

the average number of submissions of each cluster and 2) from a value derived from the transition matrix itself, the Syntactic Score.

This work provides some practical benefits to educators. For instance, clustering students into groups with much, some and little prior programming experience enables educators to differentiate students. Differentiation then enables better learning outcomes, as students are given exercises in their own skill level instead of those too easy or too difficult. Mok (2011) has tentatively shown improved student engagement and motivation using three tiers of differentiation: exercises for little, moderately and high skilled students.

Achieving these benefits, does, unfortunately require the educator to be able to collect data and train this model from the collected data. Currently there is no publicly available tool to easily use the model. Plans for integrating this into the LMS of the authors, ViLLE (Laakso *et al*., 2018) are underway. Regardless, once the model has been trained on one instance of the programming course, it can be used on subsequent instances. Additionally, as is with machine learning models, as more and more data is collected and added to the model, the model should become more and more accurate. As long as the exercises on the course remain of similar difficulty and the student demography remains the same (beginners or novices with little programming experience), the model should remain relevant. However, proving this claim is planned as future work.

This work also provides theoretical benefits to the programming education community. We demonstrate that student programming behavior can be modeled using a Markov process and creating a transition matrix. This matrix captures features outside the data from which it was created (the submission history), as we demonstrated by clustering the transition matrices intro three distinct groups which had a statistically different average of prior programming knowledge.

Our work also provides one way to measure student current skill in programming. Unlike many metrics in use today, such as the EQ (Jadud, 2006), Watwin score (Watson *et al.*, 2013), the model described in this study generates a 29-dimensional variable space which can easily be clustered using modern machine learning techniques. The generated clusters contain similar students, thus if one student is identified as skilled, the other students in the same clusters may also be inferred to be skilled. To this end, we also introduced the Syntactic Score is a simple metric and similar to the EQ and Watwin score.

Because of the application possibilities of the transition matrix with modern machine learning methods, other machine learning methods, such as dimensionality reduction or deep learning may also be utilized. Lu and Hsiao (2019) were successful in detecting students in need of help during a long programming exercise using deep learning. We believe the transition matrix, and especially how it changes over time could identify students in need of help during an entire course.

Our results show student programing performance evening out towards the end of the course. This is likely due to the course material increasing steadily in difficulty and at some point the difficulty of the course material reaches the level of the student's prior programming experience. This is however, a welcome benefit, as it means the subsequent programming courses can then trust their student population to be of somewhat similar skill level.

The last week of both courses were markedly different from the earlier ones: the Java course had a revision week with no new material and the Python course had an extremely difficult last week. The revision week for Java effectively measured learning over the course and those who had learned to program functioned effectively at the same level as those who had previous programming knowledge. The last week for Python seems to have been so difficult, that most of the students in C1 resigned after only a few attempts, as indicated by an extremely low average submission count (even lower than that of C3) and an extremely low Syntactic Score.

Hagan and Markham (2000) argue that the number of languages known by the student is a valid metric for roughly measuring prior programming knowledge. Our results prove this by having the cluster with the most prior programming experience also had experience with the most number of programming languages. This seems to imply, that programming students should be introduced to multiple languages, at some point during their programming education. After all, the students with experience with the highest number of languages performed the best on both courses. According to some teachers, learning multiple programming languages adds excitement and motivation to studies (Tshukudu, 2021). However, a question remains: *when* to teach students their second programming language? The students without a firm grasp of their first programming language are more likely to quit than find excitement and motivation in the second programming language. This metric, or other similar metrics, might be applicable to determining when a student is ready to be introduced to his or her second programming language.

7. Conclusions and Future Work

This study presents a model for clustering students on an introductory programming course from only the generated submissions to short programming exercises. We used the k-means clustering algorithm to cluster students based on their transition matrices. We chose the number of clusters to be three based on analysis of the inertia and silhouette score for the dataset. While similar clustering models have been described (ElGamal, 2013; Ahadi *et al.* 2015; Rubio *et al.*, 2019), they needed external data. The model in this study can be computed without the use of surveys or external tests using only the student's submission history.

We can also conclude that previous programming experience has a marked effect on the performance of programming students on an introductory course. The cluster with the highest self-reported prior programing experience had the lowest number of submissions and the least amount of errors (as indicated by the Syntactic Score). This difference in student skill should be taken into account when designing programming courses.

This study paves way for interesting future work. The transition matrix was created from a normal Markov model. It would be interesting to see if the model could be improved by using a Hidden Markov Model, as it contains latent states, which could be used to determine attributes of the programmer, such as frustration or engagement. Aside from improving the model, we believe applying dimensionality reduction techniques to

the transition matrix data could reveal interesting combinations of state transitions. For instance a Principal Component Analysis (PCA) might provide new insights into modelling student behaviour, and this is planned as future work.

Because the model successfully creates clusters of students with similar prior programming experience, the generated clusters may correlate well with other features of programmers as well. For instance, because skill in writing programs correlates strongly with tracing code (Lister *et al.*, 2010), we strongly believe that C3, the cluster with the most self-reported programming experience, would score highly on both these aspects of programming. As such, future work could include using the model described in this study to directly determine students' capabilities of tracing code.

As with all machine learning, the training dataset should be chosen carefully. This model was created with a dataset containing students with prior programming knowledge as well as those without. It is vitally important to have both groups when training the model, as otherwise the k-means clustering algorithm will not successfully place the cluster centroids and thus fail to create meaningful clusters.

This study contains some internal limitations. The study was performed on only two datasets from the same university. Different universities or educational level may produce different transition matrices, which may cluster differently in regards to previous programming skill. We also only used one clustering algorithm: k-means clustering. While we received good results, it may not be the optimal choice and better results are achievable with different algorithms. We also measured student prior programming knowledge using a survey, where students self-reported their programming experience. Self-reporting is subject to psychological effects, such as the Dunning-Kruger effect, where people with low skill overestimate their skill level and conversely, people with high skill underestimate their skill (Dunning, 2011).

References

Ahadi, A., Lister, R., Haapala, H., Vihavainen, A. (2015, August). Exploring machine learning methods to automatically identify students in need of assistance. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 121–130).

Asmussen, S. (2003). Markov chains. *Applied Probability and Queues*, 3–38.

Bellman, R. (1966). Dynamic programming. *Science*, 153(3731), 34–37.

Bergin, S., Reilly, R. (2005, February). Programming: factors that influence success. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 411–415).

Dunning, D. (2011). The Dunning–Kruger effect: On being ignorant of one's own ignorance. In: *Advances in Experimental Social Psychology* (Vol. 44, pp. 247–296). Academic Press.

ElGamal, A.F. (2013). An educational data mining model for predicting student performance in programming course. *International Journal of Computer Applications*, 70(17), 22–28.

Eloy, A., Achutti, C.F., Fernandez, C., de Deus Lopes, R. (2022). A Data-Driven Approach to Assess Computational Thinking Concepts Based on Learners' Artifacts. *Informatics in Education*, *21*(1).

Gagniuc, P.A. (2017). *Markov Chains: from Theory to Implementation and Experimentation*. John Wiley & Sons.

Gjelsten, B.K., Bergersen, G.R., Sjøberg, D.I., Cutts, Q. (2021, November). No Gender Difference in CS1 Grade for Students with Programming from High School: An Exploratory Study. In: *21st Koli Calling International Conference on Computing Education Research* (pp. 1–5).

Grover, S., Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43.

Hagan, D., Markham, S. (2000, July). Does it help to have some programming experience before beginning a computing degree program?. In: *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education* (pp. 25–28).

Hennig, C. (2007). Cluster-wise assessment of cluster stability. *Computational Statistics & Data Analysis*, 52(1), 258–271.

Jadud, M.C. (2006). *An Exploration of Novice Compilation Behaviour in BlueJ* (Doctoral dissertation). University of Kent.

Jiang, B., Zhao, W., Zhang, N., Qiu, F. (2022). Programming trajectories analytics in block-based programming language learning. *Interactive Learning Environments*, 30(1), 113–126.

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

Lahtinen, E. (2007, July). A Categorization of Novice Programmers: A Cluster Analysis Study. In: *PPIG* (Vol. 16, pp. 32–41).

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

Lu, Y., Hsiao, I.H. (2019, March). Exploring programming semantic analytics with deep learning models. In: *Proceedings of the 9th International Conference on Learning Analytics & Knowledge* (pp. 155–159).

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., ... Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119–150.

Lister, R., Clear, T., Bouvier, D.J., Carter, P., Eckerdal, A., Jacková, J., ... Thompson, E. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, 41(4), 156–173.

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

MacQueen, J. (1967, June). Some methods for classification and analysis of multivariate observations. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, No. 14, pp. 281–297).

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

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

Naz, A., Lu, M., Zackoski, C.R., Dingus, C.R. (2017, June). Applying Scratch programming to facilitate teaching in k-12 classrooms. In: *2017 ASEE Annual Conference & Exposition*.

Novais, D.J.F., Pereira, M.J.V., Henriques, P.R. (2017, September). Program analysis for clustering programmers' profile. In: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* (pp. 701–705). IEEE.

Nowaczyk, R.H. (1984). The relationship of problem-solving ability and course performance among novice programmers. *International Journal of Man-Machine Studies*, 21(2), 149–160.

Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1), 37–55.

Robins, A., Rountree, J., Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137–172.

Rubio, M.A. (2019, May). Automatic Categorization of Introductory Programming Students. In: *International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2019) and 10th International Conference on EUropean Transnational Education (ICEUTE 2019)* (pp. 302–311). Springer, Cham.

Sadiku, L.M. (2015). The importance of four skills reading, speaking, writing, listening in a lesson hour. *European Journal of Language and Literature*, 1(1), 29–31.

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

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

Tshukudu, E., Cutts, Q., Goletti, O., Swidan, A., Hermans, F. (2021, August). Teachers' Views and Experiences on Teaching Second and Subsequent Programming Languages. In: *Proceedings of the 17th ACM Conference on International Computing Education Research* (pp. 294–305).

Vainio, V., Sajaniemi, J. (2007). Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin*, 39(3), 236–240.

van der Werf, V., Aivaloglou, E., Hermans, F., Specht, M. (2021, November). What does this Python code do? An exploratory analysis of novice students' code explanations. In: *Proceedings of the 10th Computer Science Education Research Conference* (pp. 94–107).

Venables, A., Tan, G., Lister, R. (2009, August). A closer look at tracing, explaining and code writing skills in the novice programmer. In: *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (pp. 117–128).

Vygotsky, L.S. (1978). *Mind in Society: Development of Higher Psychological Processes*. Harvard university press.

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

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

Yarmish, G., Kopec, D. (2007). Revisiting novice programmer errors. *ACM SIGCSE Bulletin*, 39(2), 131–137.

E. Lokkila (MSc, Computer Science) is a doctoral student in the Department of Computing at the University of Turku, Finland. He is currently working as a university lecturer and teaches several first year computer science courses.His PhD thesis involves improving programming education by identifying the students in need and providing them with targeted assistance. His other research interest are gamification, learning analytics and programming language theory.

A. Christopoulos (Ph.D., Computer Science) is a Research Fellow in the Faculty of Science at the University of Turku, Finland. Dr. Christopoulos is currently working for the Turku Research Institute for Learning Analytics where he investigates matters related to digital inclusion, educational technology advancement, immersive technologies and learning analytics. The Institute is also developing 'ViLLE', a digital learning platform that includes content and exercises for studying mathematics, programming, and languages.

M.-J. Laakso (Ph.D., Tech) is the director of the Turku Research Institute for Learning Analytics at the University of Turku, Finland. His main research interests are Learning Analytics, Computer Assisted Learning, Math & Programming Education, Gamification, Learning Design, Machine Learning & AI in Education. He has 20 years of experience from university and research-based development of the education through educational technology solutions. He has published more than 100 international peer-reviewed articles, and collected more than 4M € in R&D projects. The centre is developing an UNESCO-awarded tool named 'ViLLE' – The collaborative education platform. The system is utilized by 60% of schools in Finland and learners are doing more than 200.000.000 tasks annually.