Review of CSEDM Data and Introduction of Two Public CS1 Keystroke Datasets

John Edwards
Utah State University
john.edwards@usu.edu

Kaden Hart
Utah State University
kaden.hart@usu.edu

Raj Shrestha
Utah State University
raj.shrestha@usu.edu

Analysis of programming process data has become popular in computing education research and educational data mining in the last decade. This type of data is quantitative, often of high temporal resolution, and it can be collected non-intrusively while the student is in a natural setting. Many levels of granularity can be obtained, such as submission, compilation, edit, and keystroke events, with keystroke-level logs being the most fine-grained of commonly used dataset types. However, the lack of open datasets, especially at the keystroke level, is notable. There are several reasons for this failing, with the most prominent being the challenges of deidentification that are peculiar to keystroke log data. In this paper, we present the public release of two fully deidentified keystroke datasets that are the first of their kind in terms of both event and metadata richness. We describe our collection technique and properties of the data along with deidentification techniques that, while not fully relieving researchers of significant effort, at least reduce and streamline manual work in hopes that researchers will release similar datasets in the future.

Keywords: datasets, keystrokes, CS1

1. Introduction

In an ITiCSE working group report on educational data mining, Ihantola et al. (2015) report that only 5% of studies they investigated were based on open datasets. As a grand challenge, the authors suggest that "future research in the area of collection of programming process data should seek to consider how data could be collected to contain enough granularity to be meaningful, yet be in a completely unidentifiable form" (Ihantola et al., 2015).

Making programming process data publicly available has numerous benefits, including enabling re-analysis studies, simplifying replication studies, and providing material for innovative studies. As a computing sciences educational data mining (CSEDM) community, we tend to do well in providing data collection tools to the research community, e.g., Web-CAT (Edwards et al., 2009) and BlueJ (Brown et al., 2014), but sharing data is not as common. While it is true that providing tools allows other researchers to collect similar data, it is also problematic for a number of reasons: researchers at small institutions may not have the resources to collect large datasets; virtually all tools require the adoption of certain integrated development environments (IDEs) or other software that some departments may resist; the ethics review board process can be daunting for theoreticians and others not accustomed to human-based empirical studies; and some educational institutions may have instructors who are less than enthusiastic about running

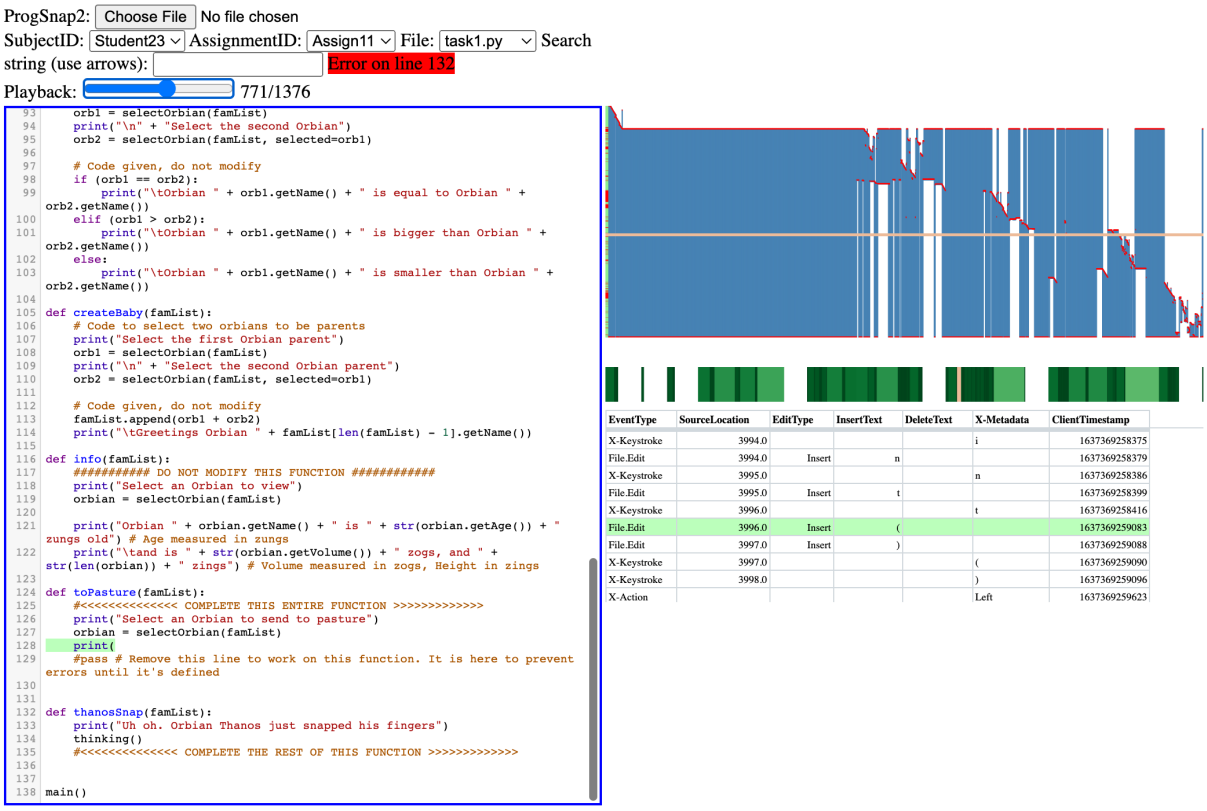


Figure 1: *KeystrokeExplorer*. We are making two keystroke log datasets public. These files can be loaded into *KeystrokeExplorer* to see playback and other details. On the left is the playback window. The line highlighted in green is the location of the most recent event (a press of the left parenthesis key). On the top right is the *CodeProcess* chart (Shrestha et al., 2022), middle right shows keystroke latencies, and bottom right shows the log file with selected columns.

studies in their courses. Having open datasets could have a great impact on researchers affected by some or all of these factors.

The lack of public programming process datasets, especially keystroke log data, is perhaps unsurprising for one reason in particular: deidentification. The process of deidentifying this type of data is intimidating because it is more involved than simply doing a search and replace of names with unique identifiers on a collection of files. The biggest challenge is that all edits during the programming process, not just the final files, must be deidentified. Manually reviewing playback of millions of keystrokes is not only daunting, but it is also error-prone.

In this paper, we discuss the release of two keystroke datasets[1,2] for use by the CSEDM community. These datasets have been used in multiple published studies (Edwards et al., 2020; Edwards et al., 2020; Edwards et al., 2020; Edwards et al., 2020; Zavgorodniaia et al., 2021; Shrestha et al., 2022; Edwards et al., 2022; Shrestha et al., 2022). The datasets are CSV log files of keystrokes and other events of CS1 students while working on their programming projects. See Fig. 1. The first of these datasets, collected in a CS1 course in 2019, has 5 million unique events across 505 subjects and 5 assignments. It includes assignment descriptions, due dates,

[1]2019 dataset: https://doi.org/10.7910/DVN/6BPCXN
[2]2021 dataset: https://doi.org/10.7910/DVN/BVOF7S

and student grades. The second dataset, collected in a CS1 course in 2021, is smaller, with 1 million events, but is richer, with all of the metadata in the 2019 dataset, plus course grades, student high school GPAs, ACT scores, and college major. We describe the datasets, their uses for both researchers and educators, and how to acquire them. The datasets are in the *ProgSnap2* format (Price et al., 2020). To our knowledge, these are the first open datasets of their kind.

In this paper, we also discuss a data collection tool and an analysis tool that are available to ease entry into the world of keystroke analysis. Our *ShowYourWork* plugin is available in the JetBrains (PyCharm, IntelliJ) plugin marketplace. Once installed, it logs keystrokes to a file called *ShowYourWork.csv*. The analysis tool is called *KeystrokeExplorer*[3], a web-based visualization tool for replay and simple analysis of keystroke logs (Fig. 1). We also have made a Jupyter notebook available that has code for tasks that could be useful to researchers, including submission reconstruction from keystrokes, time-on-task estimation (Edwards et al., 2022), and various scatterplots and statistics. Both *KeystrokeExplorer* and the Jupyter notebook accept any keystroke log file in the *ProgSnap2* format.

In this paper, we also describe the deidentification process[4] of keystroke log data. Deidentification is not only complex but is also labor-intensive. The section on deidentification is written with an eye toward making it easier for other research groups to deidentify and share their keystroke log data. Admittedly, it still takes a lot of work, but we believe our approach makes the process at least achievable with a reasonable amount of effort.

One motivation for making these datasets public is to enable and encourage replication studies. It has been found that only 1/3 to 1/2 of replication studies found similar findings to the original studies among 100 articles published in top psychology venues (Collaboration, 2015). Replication studies are neither common nor particularly valued in the computing education research community (Ihantola et al., 2015; Ahadi et al., 2016). Perhaps having more data that is easily usable will help the community to engage in more such replication work.

Specific contributions of this paper are:

- Description of two keystroke datasets that are being publicly released together with this paper.
- Description of our deidentification process which enabled public release of the datasets.
- Description of two tools we are releasing, *KeystrokeExplorer* and a Jupyter notebook, to assist in the use of these datasets.
- A review of CSEDM data and tools with emphasis on work since 2015.
- A taxonomy of the uses of CSEDM data and open problems.

This paper is organized as follows. We begin with a somewhat detailed look at recent work that uses CSEDM data (Section 2) followed by a description of two datasets that we have made public (Section 3). We then describe our process of deidentification (Section 4) and discuss tools that accompany the data (Section 5). We then make concluding remarks (Section 6).

[3] edwardsjohnmartin.github.io/KeystrokeExplorer. The source code can be found at github.com/edwardsjohnmartin/KeystrokeExplorer.

[4] The source code for our deidentification can be found at github.com/EdwardsLabUSU/DeidentifyKeystrokes.

2. Recent work with CSEDM data

In this section, we discuss studies that use programming process data in general, and keystroke logs specifically. These studies span a number of objectives, use varying data granularities, and offer and take advantage of various tools. Our discussion focuses primarily on work done from 2015 to 2022, as the survey by Ihantola et al. (2015) provides an excellent overview of work done in the decade prior to 2015, though we do include work prior to 2015 that is either foundational or particularly relevant to our keystroke datasets.

This section is fairly detailed. Since the two datasets that we describe in this paper are the first of their kind to be released, we take some care in giving context and situating the data in the CSEDM space. We first categorize the use of CSEDM data by objective, then granularity. We then discuss tools for collecting and analyzing CSEDM data, then discuss open datasets, and then list some open questions. We conclude this section by discussing what role the two datasets we are releasing play relative to prior and future work.

2.1. Objectives

In this section, we discuss related work categorized by study objectives. We discuss five such objectives: predicting course outcomes, descriptive analytics, differences in context, biometrics, adaptive support systems, and plagiarism detection.

2.1.1. Predicting course outcomes

One of the most common objectives of educational data mining in the context of computing education is that of predicting academic outcomes of students such as exam scores and final grades. The most common goal is to identify struggling students early in the course, in time for effective intervention, and to do so with data that can be collected without placing a burden on the instructor.

Early work on predictive analytics in CSEDM included compiler error analysis. Jadud (2006), in addition to seminal work on descriptive statistics of compiler errors in practice, formulated the Error Quotient (EQ), a characterization of a student's struggles with compiler errors. He used the EQ to predict both exam scores and final grades. Adjustments to and improvements of the EQ include the Watwin score (Watson et al., 2013) and the RED score (Becker, 2016). Another line of inquiry focused on replacing or enhancing compiler features with other features, such as using non-edit events and pauses as stand-ins for compile events to predict low- versus high-performing students (Ahadi et al., 2015; Castro-Wunsch et al., 2017). The Normalized Programming State Model (NPSM) uses an approximation of semantic correctness of the program in addition to syntactic correctness (Carter et al., 2015). The same research group later incorporated the concept of the Programming State Model (PSM), a categorization given by a 2x2 syntactic/semantic state matrix, into EQ, Watwin, and NPSM scores (Carter et al., 2017). They also discovered that a student's path through different programming states was predictive of final grade (Carter and Hundhausen, 2017).

Compiler error analysis sparked increased interest in finding programming process features beyond compiler errors that predict academic outcomes such as exam scores and final grades. Features that have been investigated include number of statements added or removed before each save (Kazerouni et al., 2017), number of attempts on programming exercises (Ahadi et al., 2016; Ahadi et al., 2017; Koutcheme et al., 2022), number of keystrokes (Spacco et al., 2015),

debugger and git events (Kazerouni et al., 2017), code submissions (Vinker and Rubinstein, 2022), success rate on programming exercises (Spacco et al., 2015; Koutcheme et al., 2022), typing speed (Thomas et al., 2005; Leinonen et al., 2016), keystroke latencies (Edwards et al., 2020), number of study sessions (Koutcheme et al., 2022), behavior in a block-based language IDE (Gao et al., 2021), time-on-task approximations (Leinonen et al., 2022), when students start their assignments relative to the due date (Edwards et al., 2009; Leinonen et al., 2021), number of IDE hints (Estey and Coady, 2016; Estey et al., 2017), and number of posts in a social media environment (Carter et al., 2017).

2.1.2. Descriptive analytics

While most CSEDM work attempts to predict outcomes, some work is focused solely on describing student behaviors without a predictive component other than, possibly, to predict elements of the programming process itself.

Similar to prediction, compiler errors are a popular topic in descriptive analytics. Smith and Rixner (2019) studied the distribution and evolution of different error types. McCall and Kölling (2019) characterized the "severity" of compiler errors, which they define as the product of frequency and difficulty, where difficulty is defined using time-to-fix. Other work also looked at time-to-fix of different compiler errors (Altadmri and Brown, 2015). Spacco et al. (2015) found that more difficult exercises require more time and effort, yet without increasing the number of compilation failures and, as a corollary, found that students improved in writing syntactically correct code despite the increasing difficulty of the exercises throughout the course. Similar work used error analysis to find that students gradually show fewer misconceptions as they progress through a course (Kurvinen et al., 2016).

In what could be seen as auxiliary work to prediction tasks, Leinonen et al. (2017) calculated pair-correlations between fine-grained features including time-on-task (computed from keystrokes), edit event count, run event count, assignment correctness, and educational value. Among other contributions, this work can help avoid prediction studies that unnecessarily test features that are correlated with features that have already been shown to be predictive or not.

Keystrokes have been used for emotion detection. Allen et al. (2016) used keystroke data to predict the affective state of people while writing natural language essays. They found that boredom and engagement were particularly well-suited to prediction, and even the simple median keystroke latency could be used. Similar studies using natural language typing patterns to predict affective state include (Vizer et al., 2009; Epp et al., 2011; Bixler and D'Mello, 2013). The CSEDM community has also investigated this space. Kołakowska (2016) predicted student stress using keystroke features and Tiam-Lee and Sumi (2019) correlated engagement with code modification.

Programming process data give excellent insight into student experience while programming. It can be captured while the student is in a natural setting and with little-to-no disruption to their process. This enables time-of-day and time-on-task studies. Zavgorodniaia et al. (2021) clustered students into circadian rhythm characterizations, finding along the way that most students do their programming during the day and evening, contrary to some stereotypes of programmers as night owls. Time-on-task has been an important measure for predicting and describing the student experience. However, ad hoc estimates, until recently the only way of calculating time-on-task from process data, have been shown to be unreliable (Kovanovic et al., 2015; Nguyen, 2020; Edwards et al., 2022). Leinonen et al. (2021) compared time-on-task esti-

mates using course-grained (submission) versus fine-grained (keystrokes) data, finding that the two modalities had only very weak time-on-task correlations, implying that fine-grained data should be used for time estimation. Edwards et al. (2022) performed an empirical study in which students were periodically prompted while working on programming assignments asking whether they were on task or not. Using elapsed time between the last keystroke and the student's response, they fit a regression model that predicts a probability that a student is engaged or not for any given gap between keystrokes. Using this model to more accurately predict time-on-task could improve time-on-task estimates for fixing compilation errors (Altadmri and Brown, 2015; McCall and Kölling, 2019), correlations between time-on-task and other programming process data (Leinonen et al., 2017), predicting time to complete a solution (Kazerouni et al., 2017), measuring the effect of an intervention (Leinonen et al., 2019; Edwards et al., 2020), intervening when students are falling behind (Rodriguez-Rivera et al., 2022), and using time-on-task for exam score prediction (Leinonen et al., 2022).

Other works look specifically at the process students take to write programs. Kazerouni et al. (2017) report how the number of statements and methods added or removed can be used to compute an index of how early and often a student works on their project, a measure of incremental test checking, and a measure of incremental test writing. Using code snapshots at saves and compiles, Blikstein et al. (2014) cluster students as tinkerers and planners. Piech et al. (2012) similarly compare code snapshots to cluster students into groups who use similar problem-solving trajectories. Similar to the matrix view of Piech et al. (2012), Shrestha et al. (2022) introduced *CodeProcess* charts. These charts indicate where in the code the changes are made, so observers can see whether a student wrote their code linearly or if they jumped around making changes. The *Pensieve* tool is designed to use programming process snapshots to facilitate discussions between instructor and student about the process taken to complete a programming project. The authors suggest that its use improves metacognition (Yan et al., 2019). See Section 2.5 for a discussion of open questions regarding programming process.

A perhaps under-utilized use of CSEDM data is in measuring differences between groups in a controlled study. Two works analyze the number of keystrokes and time-on-task between students who used a syntax practice intervention and those who didn't (Leinonen et al., 2019; Edwards et al., 2020). The lack of use of CSEDM data in this area could be attributed to researchers' desire to use more direct measures of success (e.g. course outcomes), but in terms of richness of insight into student experience, keystroke, and other fine-grained process data may be superior.

2.1.3. Differences in context

While conducting a study in different contexts is generally geared toward ensuring that findings are generalizable, some CSEDM work has used programming process data to understand the contexts themselves. Jadud and Dorn (2015) computed EQ on Blackbox data and compared distributions between countries and found that some cultural affinities may be discovered through analysis of, e.g., the number of days spent programming. Edwards et al. (2020) did an analysis of the predictive power of digraph latencies as features across two contexts: Python/English and Java/Finnish. They showed digraph distributions differ significantly between programming languages (Python and Java) and spoken languages (English and Finnish). Edwards et al. (2020) looked at the effect of differences between programming and typing natural language on typing characteristics such as the speed of typing particular constructs and how quickly a person

corrects an error.

2.1.4. Biometrics, identification, and authentication

Studies of biometrics using programming process data are entirely based on keystroke data, which may not come as a surprise as, intuitively, keystrokes are the only data type that have the granularity to admit identification and authentication. Early work was done using data from natural language typing. Morales and Fierrez (2015) use keystroke dynamics to authenticate students in an exam situation. They reported 90% authentication accuracy using only 100 keystroke digraphs and trigraphs. The keystroke data was taken from the OhKBIC dataset (Monaco et al., 2015). Krishnamoorthy et al. (2018) used a richer set of keystroke dynamics features, including length of time a key is pressed, interval between release of a key and press of the next, and interval between presses of consecutive keys, as features into a support vector machine for biometric authentication.

In the CSEDM space, Leinonen et al. (2016) used the top 25-50 digraph latencies, as given by feature selection, to identify whether the student was a novice at programming or not. Longi et al. (2015) were the first to identify users with programming data using average latency, average latency for given keys, and average latency for digraphs using a nearest neighbor classifier. This was followed by work geared toward authentication in an exam setting, where fewer keystrokes are available and students are using unfamiliar computers and keyboards (Leinonen et al., 2016).

2.1.5. Adaptive support systems

Adaptive support is an umbrella term that comprises systems that adapt learning environments, activities, and hinting to the current knowledge state of a student. A common adaptive support system is the Intelligent Tutoring System (ITS) which is typically integrated into an IDE or learning environment and that gives feedback and help that are individualized to students. Submission-level CSEDM data is typically used to design and train the ITS. Since most systems are trained using only canonical solutions, possibly along with sample sets of input and output, little is said about the training data: educators generally have solution data, whether generated by themselves or from previous offerings of a course, and so no customized software or public data repository is needed in order to train the ITS.

Foundational work in Intelligent Tutoring Systems was done by Barnes and Stamper (2008) who used Markov Decision Processes from a set of canonical solutions to dynamically generate hinting in the context of learning logic proofs. Rivers and Koedinger (2017) used similar ideas for a Python programming course but first converted input solutions into abstract syntax trees (ASTs) in order to analyze a more meaningful structure of the code, which is independent of identifiers but unfortunately still suffers from how the code is structured. For this reason, Rivers and Koedinger (2017) uses multiple input solutions to find the closest one to the code the student is working on. The *SourceCheck* ITS from (Price et al., 2017) similarly uses ASTs for comparison to and hint generation from solution code. Being based on ASTs, *SourceCheck* is largely language independent and, indeed, is implemented for *iSnap*, a block-based language (Price et al., 2017). A 2018 survey looks at 14 Intelligent Programming Tutors described in the literature (Crow et al., 2018).

2.1.6. Plagiarism detection

Despite providing a rich view into the process students use to write a computer program, very little work has been done in using programming process data for plagiarism detection. Rodriguez-Rivera et al. (2022) use repository commit data to look for projects with mostly additions and few deletions of code. Two finer-grained approaches use keystroke-level data. Hellas et al. (2017) construct a line chart that shows the edit distance from the current snapshot to the final submission for each keystroke event. Sudden drops in the edit distance indicate pasted code and consistent, linear drops indicate that the student may be re-typing someone else's solution. Shrestha et al. (2022) improved on edit distance line charts with *CodeProcess* charts that indicate where in the code temporal changes are made. Almost all types of plagiarism, including large pastes, type-copying, and changing identifier names and comments at the last minute, can be readily identified.

2.2. Data granularity

Ihantola et al. (2015) define six levels of programming process data granularity which we here condense into four levels. The least granular data includes submissions and commits. In the works reviewed already in this section, most submission and commit data are used for academic outcome prediction (Ahadi et al., 2016; Castro-Wunsch et al., 2017; Ahadi et al., 2017; Vinker and Rubinstein, 2022; Koutcheme et al., 2022) and Intelligent Tutoring Systems (Barnes and Stamper, 2008; Price et al., 2017; Crow et al., 2018) with a little work in descriptive analytics (Rodriguez-Rivera et al., 2022) and plagiarism detection (Rodriguez-Rivera et al., 2022).

The next level is executions, compilations, and file saves, which are also used extensively in academic outcome prediction (Carter et al., 2015; Jadud, 2006; Watson et al., 2013; Kazerouni et al., 2017; Estey and Coady, 2016; Estey et al., 2017; Becker, 2016; Spacco et al., 2015) but is also used extensively in descriptive analytics (Kazerouni et al., 2017; Kurvinen et al., 2016; Spacco et al., 2015; Altadmri and Brown, 2015; Yan et al., 2019; Smith and Rixner, 2019; Blikstein et al., 2014; Piech et al., 2012; Carter et al., 2017; Carter et al., 2015; McCall and Kölling, 2019).

The next level is line-level edits, where a single event captures statistics of all changes made to a line. The Blackbox dataset captures line-level edits (Brown et al., 2014), yet most studies using Blackbox analyze only the compile events (Brown et al., 2018). However, line-level edits have been used in academic outcome prediction (Gao et al., 2021) and differences in context (Jadud and Dorn, 2015).

The finest-grain data are keystrokes and character edits which, unsurprisingly, offer the greatest flexibility in analysis. They are used in all five of our categories: academic outcome prediction (Ahadi et al., 2015; Leinonen et al., 2021; Leinonen et al., 2016; Carter and Hundhausen, 2017; Edwards et al., 2020; Leinonen et al., 2022), descriptive analytics (Leinonen et al., 2017; Shrestha et al., 2022; Leinonen et al., 2019; Edwards et al., 2022; Allen et al., 2016; Leinonen et al., 2021), differences in context (Edwards et al., 2020; Edwards et al., 2020), biometrics (Morales and Fierrez, 2015; Krishnamoorthy et al., 2018; Leinonen et al., 2016; Leinonen et al., 2016), and plagiarism detection (Hellas et al., 2017; Shrestha et al., 2022; Longi et al., 2015).

2.3. TOOLS

A number of data collection tools have been used in the studies we have reviewed. Some of these have been made available to the research community. In this section, we discuss data collection tools that are finer-grained than submission-level data and include studies that have either used the tool directly or have used data collected with the tool.

2.3.1. Programming exercise tools

Test My Code (Vihavainen et al., 2013; Ahadi et al., 2015; Ahadi et al., 2017; Ahadi et al., 2016; Leinonen et al., 2021; Leinonen et al., 2016; Leinonen et al., 2022; Leinonen et al., 2017; Leinonen et al., 2019; Leinonen et al., 2021; Edwards et al., 2020; Leinonen et al., 2016; Leinonen et al., 2016; Longi et al., 2015; Hellas et al., 2017) is a server[5] that automates the testing of programming exercise solutions and has an accompanying NetBeans plugin[6] that collects keystroke-level data. *BitFit* (Estey and Coady, 2016; Estey et al., 2017) is an open source[7] programming practice tool that provides intelligent hints to struggling students and collects usage log data at the compilation level. *CloudCoder* (Spacco et al., 2015) is a web-based programming exercise system that captures code edits at the character level. As of March 2022, the project is no longer active, though the source code is still available.[8] ViLLE (Kurvinen et al., 2016) is a programming practice environment that is publicly available under the name Eduten.[9] *Phanon* (Edwards et al., 2020) is a web-based syntax exercise tool that collects keystroke-level data and that is now marketed under the name *CodeKeyz*.[10]

2.3.2. Plugins

OSBIDE (Carter, 2012; Carter et al., 2015; Carter and Hundhausen, 2017; Carter et al., 2017) is a *VisualStudio* plugin[11] with social media elements that captures at least compile- and edit-level events. *DevEventTracker* (Kazerouni et al., 2017; Kazerouni et al., 2017) is an addition to an Eclipse plugin that allows students to submit code to the Web-CAT submission support system. It collects data with compilation, execution, and save granularity. Its source code is available on GitHub.[12] *PyPhanon* (Edwards et al., 2022), now called *ShowYourWork*, is a plugin for JetBrains IDE software (e.g. *PyCharm*, *IntelliJ*) that collects keystroke-level events. It is available through the JetBrains plugin marketplace. *EnCourse* (Rodriguez-Rivera et al., 2022) is a management system that commits a project to a git repository at every compile. It is open-source and available for download.[13]

2.3.3. Integrated Development Environments (IDE)

Decaf (Becker, 2016) is a Java IDE that is written primarily to research improved compiler error messages. It is unclear whether *Decaf* is publicly available or not. *BlueJ* (Braught and Midkiff,

[5] https://github.com/testmycode/tmc-server
[6] https://github.com/testmycode/tmc-netbeans
[7] https://github.com/ModSquad-AVA/BitFit
[8] https://cloudcoder.org/
[9] https://www.eduten.com/
[10] https://codekeyz.com/
[11] https://github.com/WSU-HELPLAB/OSBIDE
[12] https://github.com/web-cat/deveventtracker-feedback
[13] https://www.cs.purdue.edu/homes/grr/Encourse

2016; Jadud and Dorn, 2015; Altadmri and Brown, 2015; McCall and Kölling, 2019) is a Java development environment designed for beginners that is available for general use.[14] Blackbox is a data collection project that captures line-edit events in BlueJ and stores the logs on a server. The Blackbox database has a large number of entries and is frequently used in CSEDM studies, only a few of which are discussed in this paper. A compile-level data collection extension to BlueJ has also been used for a study (Watson et al., 2013). *Snap* [15] is a block-based programming environment fitted with ProgSnap2-format log data collection. It is available online for free use.[16] *CodeSkulptor* (Warren et al., 2014; Smith and Rixner, 2019) is a web-based Python programming environment that logs data at every file save event.[17]

2.3.4. Analysis tools

Far fewer data analysis tools are available compared to data collection tools. The lack of data analysis tools could perhaps be attributed to the lack of consensus as to what exactly in programming process data is generally important, e.g., do researchers and practitioners want to see the EQ, Watwin, RED, or NPSM score? Which features should be used for exam score prediction? Compounding the problem is that no standard data format has gained general acceptance by the community, though ProgSnap2 (Price et al., 2020) has gained some traction. But possibly the main reason for the lack of analysis tools is that they may not necessarily get used – researchers tend to prefer more flexible scripting tools such as *Pandas* and *scikit-learn*, and practitioners, well, arguably, there is not a lot of evidence that programming process data is used much at all in practice (with the exception of IDE hinting tools, such as code completion, that use transient data). But perhaps that is because few analysis tools are available to instructors. One tool that is available is *Pensieve* (Yan et al., 2019), a tool that facilitates discussions between instructor and student about the process taken to complete a programming project. Pensieve uses code snapshots to enable discussions that can increase both student success and student metacognition. The source code is available on GitHub.[18]

2.4. Public datasets

To our knowledge, no data has been released publicly that captures programming behavior at the keystroke level. Furthermore, there is a general lack of open datasets in CSEDM (Ihantola et al., 2015). See Mihaescu and Popescu (2021) for a recent review of EDM datasets which includes some of the datasets described here. The most prominent available dataset is Blackbox (Brown et al., 2014; Brown et al., 2018; Brown and Kölling, 2020) which was made public in 2013 and has perpetually been added to since, with terabytes of data available as of 2022. Blackbox logs events from the BlueJ online Java IDE at line-level granularity. This level of granularity was selected for two stated reasons. The first is because data is logged on a server, so every event requires network activity. The second reason is the difficulty of analysis. The dataset proposed by Thomas et al. (2003) had both mouse and keystroke events and proved difficult to clean and analyze, informing the Blackbox decision to simplify. While the Blackbox dataset is powerful and voluminous, there is little supporting metadata, such as assignment descriptions,

[14] https://www.bluej.org/
[15] https://snap.berkeley.edu/
[16] https://snap.berkeley.edu/
[17] https://py2.codeskulptor.org/
[18] https://github.com/chrispiech/pensieve

assignment scores, and student background. This is by design, as subjects could be using BlueJ for any purpose (Brown et al., 2018). The Blackbox dataset is stored in a database and access is granted only to established researchers because of the potential for misuse in the form of plagiarism and deidentification challenges (see Section 4).

code.org has made a dataset from their 2013 Hour of Code public.[19] It contains the progression of abstract syntax trees as students progress toward a solution in a block-based programming environment and has been used for research of automatic hint generation (Piech et al., 2015). A series of programming logs from the *iSnap* environment are available.[20] Data from both code.org and *iSnap* are from block-based environment where there is little to no opportunity for subjects to include identifying information in their solutions, and so deidentification is a simple matter of removing identifiers from the metadata.

The annual CSEDM challenge has been a catalyst for the release of open datasets. The 2019 challenge used a dataset[21] collected from subjects using the Intelligent Teaching Assistant for Programming (ITAP) tutoring system (Rivers et al., 2016) and captures submission and hint events. The second CSEDM data challenge used a dataset[22] with compile/run granularity, which was used for both knowledge tracing, by predicting performance on later programming problems, and early grade prediction, by predicting exam scores.

2.5. Open questions

As has been shown, programming process data has largely been used to predict academic outcomes. We propose that a shift in focus be considered. Programming process data is just that, a record of the process a student went through to write a program, and can provide powerful insights into learning and into the student experience. While some work has been done in this space (Shrestha et al., 2022; Shrestha et al., 2022; Blikstein et al., 2014; Piech et al., 2012), far more can be done to understand exactly what students are experiencing and thinking as they write their computer programs.

One question posed by Shrestha et al. (2022) is what exactly good programming process is. Should students be writing their code linearly from start to finish, or should they be moving between different sections of code making changes? It is possible that the answer depends on the student and/or the particular problem being solved. It may also depend on how advanced the student is. Answering this question could not only affect how we teach programming, but it could give insights into programming at a professional level as well.

Another question is what exactly *flailing* looks like (i.e., students who are working unproductively and are unlikely to make progress) and what *learning* looks like (i.e., students who are working productively) (Spacco et al., 2015). This is more than an academic exercise – realtime, automatic interventions embedded in an IDE could be implemented to support the productive use of a student's time and effective learning.

A related question is how we can characterize engagement (Edwards et al., 2022). Can we statistically detect, using keystrokes, whether a student is thinking, consulting resources, checking their phone, or taking a longer break? If we can identify, even just in the aggregate, these activities we can implement interventions as well as improve course design.

[19] https://code.org/research
[20] https://pslcdatashop.web.cmu.edu/Project?id=321
[21] https://pslcdatashop.web.cmu.edu/Files?datasetId=2865
[22] https://pslcdatashop.web.cmu.edu/Files?datasetId=3458

And finally, we have a large class of questions relating to interventions and their effect, not only on outcomes but also their effect on students' programming process. Hundhausen et al. (2017) outline a framework for pursuing these questions using CSEDM data. CSEDM data can play an integral role in controlled studies of interventions and can enrich our insights beyond the statistical analysis of traditionally measured outcomes.

2.6. The role of our datasets

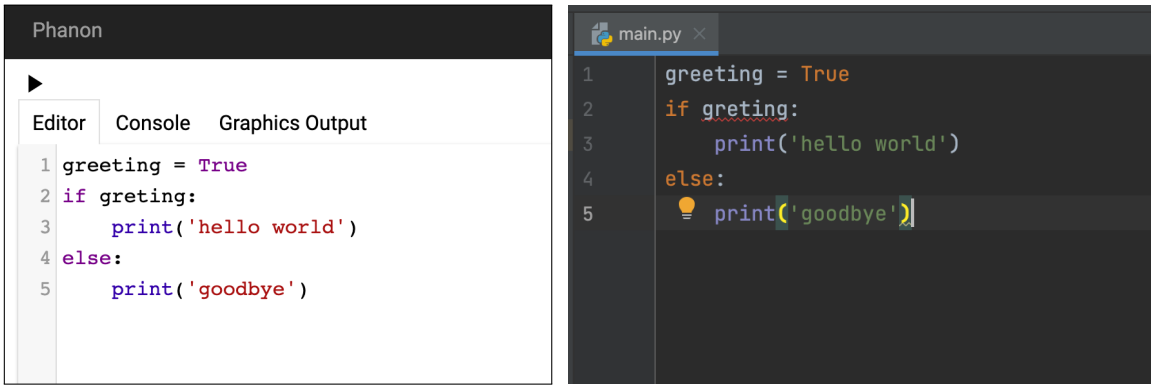
The five objectives we have discussed for work using CSEDM data are predicting course outcomes, descriptive analytics, differences in context, biometrics, and plagiarism detection. Our and other keystroke datasets have been used in studies in each of these areas (see Sec. 3.5 for specifics on studies our datasets have been used in). Each of these objective areas is actively being researched and in need of data, and so this release of the first set of public keystroke data is timely. Much of the work done recently has used coarser-grained data, such as compile- or submission-level, and so one could ask if all the effort to deidentify and acclimate to keystroke data is really necessary. In answer, we point out that in the context of the objectives listed above, keystroke data provides a richness of features that can't be matched by lower-resolution data. Indeed, the data we are releasing includes compilation and run data.

Predicting of course outcomes, biometrics, and plagiarism detection are heavily dependent on selected features, and fine-grained data provides more opportunity for combining features as well as discovering novel features. Differences in context can also benefit from a richer feature set. But perhaps the objective where fine-grained data such as keystrokes can offer the most value is in descriptive analytics. Much of our attention in our discussion of open questions has focused on descriptive analytics, e.g., characterizing and understanding the process of writing a computer program. While course-grained data provides a data point every hour or even minute, fine-grained data provides data at the millisecond level and, if we are able to harness it, can provide a veritable play-by-play view of what a student programmer is doing while writing code. Of course, there are plenty of larger pauses between keystrokes during which students are thinking, reading, reviewing, searching, and taking breaks, but during those two or three hours during which we do have keystroke data, we may be able to gain important and actionable insights into student behavior.

As we make progress in understanding the programming process, it is possible that Intelligent Tutoring Systems could be enhanced by training them not just with canonical solution code, as is done now, but with canonical processes. Much of the hint generation today is done by finding optimal paths from the current code state to a canonical code state. Yet, with programming process data, especially fine-grained data such as keystrokes, we can begin to define "optimal" as "a path taken by an expert" and generate hints to guide students in not only ending up with canonical code but developing the code in a canonical way.

3. Two keystroke datasets

We present two keystroke log datasets that we have made publicly available for researchers and practitioners. In this paper, we refer to them as the 2019 and 2021 datasets. They are available for download as cvs files along with various metadata and other supporting files. They were both collected in CS1 courses taught at Utah State University.



(a) Phanon - 2019 dataset

(b) PyCharm - 2021 dataset

Figure 2: Phanon and PyCharm IDEs used to collect 2019 and 2021 data, respectively. Phanon has no hinting mechanisms whereas PyCharm has underlining and textual hints via the light bulb icon.

3.1. Data Collection Context

The 2019 dataset was collected as part of a study on "syntax exercises," which are short, high-repetition exercises intended to help students become proficient in Python syntax before they work on their weekly programming assignment (Edwards et al., 2020). We wrote a custom, browser-based IDE called *Phanon* to deliver the syntax exercises and so that students would not have to learn a second development environment, we added an interface for students to complete their programming assignments within the *Phanon* environment (Fig. 2a). Keystrokes were collected in order to measure whether students were spending significantly less effort on their weekly projects as a result of the syntax exercises. Participation rates in the study were extremely high (74%, resulting in 505 participants) because students did not need to do anything to participate in the study beyond signing the informed consent form.

The 2021 dataset was collected as part of a study on attention and engagement (Edwards et al., 2022). Students used the *PyCharm* IDE (Fig. 2b) to complete their weekly programming projects. We wrote a plugin called *PyPhanon* that had two primary functionalities. The first was that it would periodically prompt students to declare if they were working on their assignment or not. The second functionality was to log keystrokes. In the attention study, we correlated student responses to the prompts with their keystrokes and built a regression model to predict if a student was on- or off-task based on the amount of time since their last keystroke. Participation rates were much lower in this study (44 participants) because students would be required to reply to the periodic prompts as to whether they were on task or not.

As will be seen in Section 3.3, both datasets include course metadata such as assignment descriptions and exam scores. Only the ethics review board protocol for the 2021 study included prior student academic data such as ACT score and college major. Neither study included demographic information such as gender or ethnicity. In 2019 and 2021, 21% and 20% of students enrolled in CS1 at Utah State University were female, respectively. 15% of students in 2019 and 20% in 2021 were declared Computer Science majors. Utah State University undergraduate enrollment is approximately 83% white and 6% Hispanic or Latinx.

Table 1: Summary of programming project tasks. Details are in a readme that is downloadable with the data. *Time* is the median time in h:mm taken by students to complete the assignment (both tasks are included in the time estimate). Tasks prefixed with "Turtle:" use Turtle graphics for drawing.

Year	Assign	Time	Task	Description
2019	p4	0:21	1	Turtle: Draw snowman using turtle graphics
2019	p4		2	Calculate surface area and volume of a cuboid
2019	p5	0:50	1	Investment/interest calculations
2019	p5		2	Turtle: Draw bullseye
2019	p6	1:33	1	Calculate area of a polygon
2019	p6		2	Calculate employee pay
2019	p7	1:26	1	Rock, paper, scissors game with randomness
2019	p7		2	Calculate if two circles intersect
2019	p8	1:15	1	Discover perfect numbers
2019	p8		2	Simulate the Monty Hall problem
2021	Assign6	3:03	1	Calculate fluky numbers
2021	Assign6		2	Zookeeper and elephants in pens
2021	Assign7	3:02	1	Number pyramids
2021	Assign7		2	Turtle: Draw a chessboard
2021	Assign8	2:18	1	Turtle: Write functions that draw patterns
2021	Assign9	2:35	1	Turtle: Draw a face
2021	Assign9		2	Use classes to write a blobber pet
2021	Assign10	0:05	1	Generate output based on two words
2021	Assign11	1:23	1	Use classes to create and manage a human-like family
2021	Assign12	2:05	1	Modify lists
2021	Assign12		2	User enters numbers into list
2021	Assign12		3	Card game
2021	Assign13	0:40	1	Memory game

3.2. Dataset descriptions

The 2019 dataset was collected during spring and fall terms of 2019. There were five assignments with due dates between January 26 and February 23 for spring term and September 14 through October 12 for fall term. Each weekly assignment had two parts: the first was a text-based logic problem and the second was a turtle graphics drawing. The dataset contains keystrokes of students using *Phanon* to complete their weekly Python programming assignments. The code editor of *Phanon* is minimal, using the *CodeMirror* library to provide syntax highlighting, but no hints or auto completion functionality are available. Being browser-based, the keystrokes were logged and stored in a database on a server then downloaded and converted to *ProgSnap2* format. Of the 505 study participants, 499 completed the first programming assignment and 448 completed the fifth programming assignment, which was the final assignment measured in the study. The median score on the first assignment was 100 (IQR: 100-100) and the median score on the fifth assignment was 94 (IQR: 74-100).

The 2021 dataset was collected during fall term of 2021. There were eight weekly assignments with due dates from October 18 to December 10. Assignments were similar to the 2019 dataset, with both text-based logic problems and turtle graphics drawings. Students completed their assignments in Python using the *PyCharm* IDE with the *PyPhanon* plugin that logged keystrokes. *PyCharm* provides syntax highlighting, self-closing parentheses and strings, and simple code completion. Keystrokes are logged in a custom format to a local file that is submitted together with the students' code to our institution's learning management system (LMS). We downloaded the log files from the LMS and converted them to *ProgSnap2* format. Of the 44 study participants, 39 completed the first programming assignment and 28 completed the eighth programming assignment, which was the final assignment measured in the study. The median score on the first assignment was 82 (IQR: 47-94) and the median score on the eighth assignment was 39 (IQR: 0-68).

For both datasets, we have made assignment descriptions available for download (the same descriptions given to the students) along with grades they received and other metadata described in Sec. 3.3. For a very brief summary of the assignment descriptions see Tab. 1. Median time-on-task for the assignments, calculated using Edwards et al's statistical model (Edwards et al., 2022), is included in the table.

In the cases of both datasets, instructors were independent of the researchers and did not have access to the keystroke log files or which students were participating in the studies. While the instructor strongly encouraged students to write their code using *Phanon* and *PyCharm*, enforcing it was impossible, and some students wrote their code using a different IDE and then pasted their solutions into *Phanon* and *PyCharm*. This is reflected in the keystroke log files as a large paste with little, if any, change afterwards.

3.3. Dataset metadata

Our datasets have a significant amount of metadata associated with them. This is important: even subtle details regarding the context in which the data is collected (e.g. assignment type) can have an impact on the analysis of the data, even to the point of seeing opposite effects between two datasets (Ihantola et al., 2015). A simple example is the difference in time that different assignments take. In the 2021 dataset, students made an average of 5570 edits to complete assignment 6, while they only made an average of 999 edits for assignment 10. Without the assignment descriptions, this difference could be difficult to understand. (Assignment 6 includes

Table 2: Details about datasets.

	2019	2021
Participants	505	44
Assignments	5	8
Submissions	3126	1304
Events	5 million	1 million
Collected	Spring/fall 2019	Fall 2021
Collection mechanism	Phanon	PyPhanon (PyCharm)
Assignment descriptions	✓	✓
Due dates	✓	✓
Assignment/exam scores	✓	✓
Final grade		✓
Highest ACT score		✓
High school GPA		✓
Major		✓
Execute success/failure	✓	✓
Execute output		✓
Error underlining		✓

two difficult tasks with numerical manipulations and assignment 10 requires just a few simple string manipulations.) For another example demonstrating the importance of detailed metadata, see Section 5.2 of Ihantola et al. (2015).

Table 2 shows what metadata is included in each of the two datasets. These features include:

- *Assignment descriptions* - For each assignment, we include a PDF file with the assignment description given to the students.
- *Due dates* - Each assignment has a due date. This is useful for studies relating to procrastination, starting early, typing behavior relating to remaining time before the due date, etc.
- *Assignment, exam, and final scores* - For each assignment, we include student scores for studies relating to behaviors leading to immediate outcomes. We also have exam and final course scores to understand relations to medium-term learning. The exam questions themselves are not released.
- *Highest ACT score and high school GPA* - These measures help us understand preparation and its relation to student experience in CS1. The ACT score is the highest ACT score reported to the university for the student. Other reported ACT scores are not included.
- *Major* - This is the students' chosen major program of study at the time they took the course. This helps in studies looking at interest and success in CS from other fields.
- *Execute success/failure/output* - We include, as do other datasets such as Blackbox, results from running the programs. Execution success and failure are in the same CSV files as the other event data. Execution output is in a different file and is large (˜3 Gb). This is because some executions result in many lines of output, especially those resulting in infinite loops.

- *Error underlining* - The 2019 dataset was collected with Phanon, a piece of academic software with very few features and no realtime compile helps, such as underlining of syntax errors. The 2021 dataset, on the other hand, does have realtime error underlining, the so-called squiggly line. This distinction can be useful in studying the effect of realtime error assist mechanisms in an IDE.

With each dataset, we include a readme file describing the different files of the dataset. We also include the IRB-approved informed consent documents that were filled out by student participants.

The number of events for the 2021 dataset that is listed in Table 2 is the effective number of events: for this dataset each keystroke results in two events, a keystroke event and a file edit event, so the actual number of events stored in the files is closer to two million.

3.4. Format

We selected the ProgSnap2 format for the keystroke data not necessarily because it is the most suited to the data (indeed, we'll see below that it presented some challenges) but because it appears that it is becoming more widely accepted as a standard format for computer programming event data. Some additions and non-standard modifications were made to make it suitable for keystroke data.

The ProgSnap2 format was designed for datasets for which storing an entire snapshot of the code state at every event is reasonable. This makes sense when the snapshots are at low temporal resolution, such as commits or compiles. Our data, however, often contains multiple events *per second*, and so storing a snapshot at each event would result in prohibitively large files. An alternative would be to store snapshots periodically, at, say, every compile. However, since reconstruction of a given state of a file is straightforward (we provide Python code to do so), little is lost in not including full snapshots.

Other changes are simply extensions using the ProgSnap2 extension mechanism and are described in readme files included with each of the dataset downloads.

3.5. Use of the datasets

The two keystroke datasets we have released for public use have already been used in a number of studies. The 2019 dataset was first collected during a controlled study analyzing the effectiveness of a syntax exercise intervention (Edwards et al., 2020) and was used for effort analysis between the control and test groups. Edwards et al. (2020) combined the 2019 dataset with a dataset collected at a Finnish university to analyze differences in keystroke patterns across different programming languages and across natural language. Edwards et al. (2020) has a similar analysis, except that it takes advantage of the fact that the 2019 dataset has a small assignment in written English prose. Edwards et al. (2020) analyzes the 2019 dataset to find that some predictive features are more reliable across assignments than others. Zavgorodniaia et al. (2021) uses the 2019 dataset to look at circadian rhythms and chronotypes of introductory programming students. Shrestha et al. (2022) introduces *CodeProcess* charts, Shrestha et al. (2022) analyzes pausing behavior among students, and Edwards et al. (2022) reports an important study looking into accurate estimation of time-on-task.

All of these studies were conducted by the same research group. Because the datasets were identifiable, they could not easily be shared with others, complicating collaborations and virtually eliminating the possibility of widespread use. The richness and usefulness of the data, as

evidenced by the publications and the growing list of interested investigators, motivated us to undertake the significant work of deidentification and public release.

The datasets' potential for future usefulness relates especially to the open questions discussed in Section 2.5. The question of what exactly qualifies as good programming process, as suggested in Shrestha et al. (2022), is a question well-suited to keystroke log data. *CodeProcess* charts are a good start, but there are at least two interesting directions the research could go in this respect. The first is to discover ways to quantify programming processes. Finding features that indicate what process is being used is an important first step towards finding effective processes. The second research direction is in conducting controlled studies and measuring process using log data. This is dependent on finding process features.

These datasets could also potentially be used for engagement studies. Similar to the pause analysis (Shrestha et al., 2022), the keystroke data could be analyzed, especially using unsupervised learning, to understand better when students are engaged and its relationship with outcomes.

4. Deidentification

Deidentification is hard and is particularly so with programming process data. The creators and maintainers of the Blackbox dataset, in fact, have restricted access because of the technical challenges of deidentification (Brown et al., 2018), and since all primary analysis must take place on a designated machine, dissemination and adoption are impaired. Indeed, it can't be used for public data challenges such as the annual Mining Software Repositories data challenge.

Deidentification is hard not just because source code files may contain unexpected identifying information that needs to be inspected and masked. Even more challenging is the fact that over half of the characters entered into a source code file are eventually deleted and don't show up in the final, submitted code.[23] The ephemeral nature of most keystrokes means that inspecting final submitted files is not sufficient – the *process* of writing the final code must be inspected as well. The most straightforward way of reviewing keystroke log files is to watch a replay of the keystrokes. Replays are interesting and insightful, but watching replays of hundreds of submissions and millions of keystrokes is daunting. Remaining attentive during replay is challenging, which is a major problem because even a momentary lapse in attention could result in missing an inadvertent paste that the student quickly deletes. A further problem is that changes are made in various parts of the file, so even for only moderately large files, a replay could jump around in the file, making it difficult for the reviewer to maintain context.

Nevertheless, deidentification is possible and can be done without "almost destroying [the source code]" (Brown et al., 2018). In consultation with our institution's ethics review board (IRB) we use the regulatory definition of identifiable, in that a document is identifiable if "the identity of the human subjects [can be] readily be ascertained directly or through identifiers linked to the subjects" (hhs, 2022). Examples of data that can readily identify participants include names, email addresses, and student identification numbers. This is the information that we target for removal in our deidentification process. Ihantola et al. (2015) point out that participants can be identified by other features in the data, such as time elapsed between keystrokes (Longi et al., 2015), time of day (Zavgorodniaia et al., 2021), and other typing patterns (Thomas et al., 2005). However, all of these identification techniques require a labeled

[23] In our datasets, 58% of characters entered into a source code file are eventually deleted.

sample, which the regulatory definition of identifiability assumes is not available.

A major challenge in preparing keystroke datasets for public release is deidentification. Deidentification of keystroke log datasets is more complex than just a search/replace of student names in submitted code files for at least two reasons. The first is that students sometimes misspell their names and add other identifying information in comments, strings, and even variable names (Ihantola et al., 2015). Indeed, the Blackbox project suggested that to fully deidentify this type of data would be to blank all comments and strings and to rename all identifiers (Brown et al., 2014). The second, and more challenging, issue is that much of the data is ephemeral. That is, the final, submitted file is only a part of what was typed, with the majority of characters being deleted before submission, yet all deleted characters can easily be recovered through a replay of the keystroke log. An example case of both of these challenges was encountered during the manual phase (see below) of deidentification when we discovered an invite for a Zoom meeting that had been (inadvertently?) pasted and then deleted. At the time that we discovered it, the linked-to meeting was still valid.

An option that we considered early on in the process was to tell the students in the informed consent document that, if they agreed to participate in the study, the only deidentification we would do would be to find their name in the final, submitted file, and mask those characters. This is the general approach taken by the Blackbox project (Brown et al., 2014). We ultimately decided that, while this type of waiver would simplify deidentification, there were ethical concerns. Consider a student who, intentionally or not, enters additional identifying text. They may realize that they don't want that text submitted and delete it. Deleting the text gives a false sense of security because, while the text disappears from the file, the deleted text, in fact, remains recorded in the keystroke log. And even if the student does realize that there remains a record of the text, there is not an immediately obvious way of removing it from the log file. The case of the accidentally pasted Zoom meeting invitation described above highlights the importance of handling deleted text with care. Even having an easy and obvious way for students to remove identifying text is not enough. See Shrestha et al. (2022) for an example of a student who, apparently having forgotten that their keystrokes were being logged, committed an egregious act of plagiarism. This demonstrated that the logging of keystrokes can be forgotten after a short time and therefore students may not think to remove identifying information beyond simply deleting the text. A comprehensive deidentification process is needed to fully protect participants' privacy.

Our approach to deidentification is to mask the characters of identifying text with the @ character. We chose this character because it is only used in the Python language as a *NumPy* matrix multiplication operator, which we do not use in our CS1 course and so, if it appears in a compilable program, would only appear in a string or comment. Alphabetic characters, such as *x*, are not good candidates for masking characters because they would confuse the statistics of character and digraph counts, which have been used in keystroke studies (e.g., (Thomas et al., 2005; Edwards et al., 2020; Edwards et al., 2020)).

A shortcoming of the @ character, however, is that if an identifier gets masked it can cause the program to fail to compile. For example, when masking the name "Eve," a function `is_even()` would get masked as `is_@@@n()`. See Sec. 4.2 for other examples. It turns out that masking keywords or identifiers is rare: for the 2021 dataset, final submissions have a total of 1172645 characters. 7195 (0.6%) of those are mask (@) characters. Of the mask characters, 12 appear outside of strings or comments, affecting compilation of a single submission among the 1304 total submissions.

The following subsections describe our deidentification process in detail. The process does not rely on data format and can be used generally with any keystroke data that is rich enough to be used to reconstruct code snapshots after any event.

4.1. Data structures

The first step in deidentification is to reconstruct the final submission of each file, maintaining a pointer for each character back to the row in the log file where that character was added (Tables 3 and 4). At the same time, we also construct a string of the deleted characters for which each character has pointers back to two events in the log file: where the character was added and where it was deleted. We are required to mask the deletion event because it also stores the character that was deleted. (It is stored for convenience and efficiency in playback.) Both the final submission and deleted text string will undergo automatic and manual masking (see the next two sections). When characters are found that need to be masked, the indices of the events (Tab. 4) are used to mask the character(s) in the *Insert/Delete Text* fields of the event (Tab. 3). When deleting characters, students typically use the backspace key, which usually results in characters being deleted in reverse order of what they were typed. Because names and other identifying information are being searched for, we reverse the order of the deleted text string. We do not reverse it character-by-character, but rather, event-by-event. The reason is because block deletes (highlighting a block of text and pressing the delete key) will appear in the deleted text string in the correct order. For example, deleting the text "Hello • w[orld]", where characters in brackets are block deleted, might be deleted in the following order: "orld"-w-•-o-l-l-e-H. If we consider these deletes as an array of strings, we reverse the array and concatenate to get "Hello • world". This approach restores most deleted text into a readable order, but there are cases that cause failure. For example, deleting every other character in "Hello • world" and then deleting the remainder would result in the deleted text string "drwolHro • le". More simply, using the delete key instead of backspace would result in "dlsow • olleH". These cases are rare, and whole, reversed strings from using the delete key can be manually discovered, as we found in the manual review phase of deidentification. See Listing 1 for an example snippet of deleted text. With these data structures in place, we can proceed with masking.

4.2. Name search

We first mask the student's name for each submission. We still know the student's name at this stage, so we do a case-insensitive search in the program text and deleted text (in reverse; see below) for the student's name. We first search and mask the student's full name by searching in the final and deleted text and following the pointers of each character back to the rows in the log file and masking those characters. This takes care of the majority of identifiable text. After masking the full name we search for each name individually (first, middle1, middle2,..., last) and replace it. Then we looked at the first five lines of each file where students were required to put their names, and looked for misspelled names and nicknames that hadn't been masked. We added these to our set of names and masked them.

Masking names can result in false positives. For example, (examples are contrived – actual cases are not reported to protect participant identity) a student with the last name of "Green" could mask the color green in turtle graphics assignments. A student named "Tim" could mask

```
# Display the calculated area
print("The area of the polygon is" + " " +
str(round(area, 5)))Jane SmithCourseName          Ncand ,
39390000000AY (ex. 0.06) (ex. 0.12)sf11281128
2811112811.2281128112839s)39s)39s)s39)) 36"""
+  + "" + "188())(39sfoms039<039s)(rts + 82
" "+ str(hours), 3839.0sc0= sDe feJane SmitjhWithholding
RatelWithholedingRattotal de)format)f + "  )
+ str(grossPay) os i  + rtatAgdoFnion print
hodlingpayosgrossPay = osprint(fprint(format
(ing;:  "ytNin;ut m/n
```

Listing 1: Deleted text example. The text is jumbled because deletions are scattered through the process of writing code. Note the names in bold on lines 3 and 7 (with the name on line 7 apparently misspelled). The student typed in a name and then deleted it, which tends to stand out among the disorder even without using the boldface font. We added and anonymized the names for illustrative purposes and the remaining text is actual deleted text from our datasets. See the text for details on how to detect and mask these events deleting identifiable text.

a part of a function called `get_time()` as `get_@@@e()`. We identified these cases and unmasked code manually.

With the name search, we also do a regular expression match looking for student identifiers which, in our case, are 8-digit numbers.

4.3. Manual search

Reconstructed files and deleted characters are mutually exclusive.

After the automatic name search, we do a manual search for additional identifying text. The apparently ideal approach would be to manually watch replays of all keystroke logs. If replaying at four events per second, this would take over 400 hours for six million events. But more importantly, reviewers are at risk of losing focus and missing a piece of identifying text. For example, if a student accidentally pastes something like the Zoom invite and immediately deletes it, the text would appear on the screen for review for only 250 milliseconds. In addition, when the code being replayed is long, it can be challenging to follow if the student jumps around in the code making changes.

Instead of reviewing replays, we manually reviewed final code submissions as well as reversed versions of deleted text. In order to ease the challenge of review we first removed all non-alphabetic, non-whitespace characters. We also removed non-identifying words as follows: we first created a set of all names, nicknames, and known name misspellings of our study participants. We then took a set of 370,102 English language words (DWYL, 2022) and removed all words that were also in the set of names (e.g., Holly, Jack, Summer). We then removed all of the non-name English words from the text to review. More formally,

$$\text{to\_review} = (\text{program} + \text{deleted}) - (\text{words} \setminus \text{names})$$

where $+$ is the append operator, $-$ indicates removal of words, and $\setminus$ is set subtraction. This approach ensures that names remain in the text to be reviewed, but other words do not.

Table 3: Log file for a student entering their first name. The first three characters are pasted, and the `e` and the `x` are initially typed in the wrong order, deleted, then retyped correctly. `TextIdx` is where the insertion or deletion occurs in the file. `Curr` is the current state of the code and does not appear in the actual log files – it is included here for clarity.

Event Idx	Text Idx	Edit Type	Insert Text	Delete Text	Curr
0	0	Paste	# A		# A
1	3	Insert	l		# Al
2	4	Insert	x		# Alx
3	5	Insert	e		# Alxe
4	5	Delete		e	# Alx
5	4	Delete		x	# Al
6	4	Insert	e		# Ale
7	5	Insert	x		# Alex

Table 4: Data structures for deidentification. Each character of the reconstructed submission text has the index of the log event (row) at which the character appeared. Each character of the deleted text has the index of where the character was added as well as where it was deleted. The text index is needed when insert and deleted events have multiple characters (e.g. pastes and block deletes, respectively).

submission text	#		A	l	e	x
event index	0	0	0	1	2	3
text index	0	1	2	0	0	0
deleted text	e	x				
insert event index	2	3				
insert text index	0	0				
delete event index	5	4				
delete text index	0	0				

After the text was manually reviewed by two researchers, which discovered issues in approximately 10 submissions, we manually masked them. The fact that only 10 submissions among the 4,430 total submissions contained issues both encourages us that the automatic tools are quite effective and reinforces in our minds that the manual search needs to be done with care – with so few issues, the manual reviewer must remain vigilant even after minutes or hours of not finding any problems. Identifying text included additional misspelled names and nicknames, such as shown in Listing 1, the Zoom invite, and, surprisingly, two additional Zoom meeting links. The names were found in both the submission and deleted text while the Zoom links were found only in deleted text.

We recognize that identifying information could have slipped through our deidentification process, whether through algorithmic imperfections or through missing something during our manual review. As a result, we are prepared to update log files as necessary should identifying information be found. We have included the following statement in the metadata of our published datasets: "This dataset has undergone deidentification, though it is possible, being a complex, temporal, and ephemeral dataset, that identifying keystrokes may have been missed. Ethical use of this dataset includes avoiding attempts at reconstructing identities. That said, if researchers discover anything identifiable in the data, they are encouraged to contact the dataset authors (john.edwards@usu.edu)."

5. Tools

We have attempted to make the adoption of our keystroke datasets as straightforward as possible by storing the data in the ProgSnap2 format (Price et al., 2020), providing a web-based tool for easy keystroke exploration, and providing a Python script, in the form of Jupyter Notebook, for common tasks.

5.1. *KeystrokeExplorer*

KeystrokeExplorer is a web-based software that enables exploration of keystroke data. See Fig. 1. It is useful for both researchers and instructors. It supports any keystroke log file in the *ProgSnap2* format with customizations described in Section 3.4.

The main feature of *KeystrokeExplorer* is the playback tool. The process of writing the program can be replayed using a slider tool or using automatic replay. In automatic replay, there are two modes. The first is to run with edits replayed at approximately 15 Hz. This mode is useful for getting an idea of how the code structure evolves. The second mode is to use the original keystroke latencies (though pauses are clamped to be no longer than five seconds). This mode is useful for understanding what parts of the program might be frustrating to students or what students are working on when they take breaks.

A version of a *CodeProcess* chart (Shrestha et al., 2022) is also included. This chart is a static 2D graphic showing what parts of the code were written when that can be used in concert with playback. We include one enhancement over *CodeProcess* charts – we include all characters that were deleted. This makes it straightforward to immediately see if a student struggled with their program or not.

Other features of *KeystrokeExplorer* include a chart of keystroke latencies as well as a view of the original csv file.

KeystrokeExplorer has a number of use cases. Researchers can use it to formulate theories around how students write code, including the order in which they write code, pause behavior, struggle, variation across students, and variation across assignments. Educators can use it in a number of different contexts. One is in conference with individual students. Students may have difficulty reasoning about the process they take when discussing challenges with an instructor, and the playback tool can help them explain their process. Another context is instructors understanding the difficulty students have in their assignments. A third use is plagiarism detection. Instructors or teaching assistants can use the tool to quickly screen for different types of plagiarism (Shrestha et al., 2022) but, equally as important, students can use the playback to defend against false positives. That is, if an instructor uses a static code plagiarism tool that flags a program as similar to another, the student can show the playback to prove that they used a different process to come up with a similar program. However, anyone using keystroke data for plagiarism detection should consider the practical and ethical issues it presents.

5.2. Python script

With the data distribution, we include a Jupyter Notebook file with various procedures that can help researchers read keystroke data, reconstruct programs at any point, estimate time-on-task, and correlate keystroke statistics with course outcomes.

6. Conclusions

In this paper, we have presented two keystroke datasets intended to play a role in CSEDM. We have taken considerable space in the paper to situate where these datasets, and hopefully others like them in the future, can contribute the most. Keystrokes are the highest resolution among common CSEDM dataset types. The only type of data with higher granularity are mouse movement logs, but these are rare and their utility hasn't been established. Keystroke datasets, however, have been used with considerable success in prediction tasks, descriptive analytics, context difference studies, biometrics, and plagiarism detection.

The most significant limitation of this paper is the possibility of missed code identifying students. As noted in Sec. 4.3, our deidentification process, because it has a manual component, could have errors. Furthermore, as noted, the data is deidentified only as long as there is no other labeled sample of a student's keystrokes. When creating similar datasets, care should be taken that labeled samples are not available. One other significant limitation is the data itself, which is processed only for deidentification, so some cleaning may need to take place. One cleaning issue is tasks; sometimes it isn't completely clear which task a keystroke belongs to, and so paying attention to what file is being modified is sometimes required if analyzing data where task distinction is important. And, of course, the lack of demographic data – gender and ethnicity in particular – reduces the applicability of the keystroke data for some studies.

A considerable challenge that we have discussed is deidentification, ostensibly a major reason that no keystroke data is publicly available. Even the line edit-level Blackbox dataset is not completely open because of deidentification issues (Brown et al., 2014; Brown et al., 2018). In this paper, we have outlined our deidentification process in detail. It includes a process for efficiently and robustly reviewing keystrokes, both those that end up in the final submission and those that are eventually deleted. Our results show that deidentification, while still burdensome, is possible.

We were aware from the beginning that deidentification would be a challenge, but we did not expect that so much manual labor would be needed. Writing scripts to prepare the data, filter and mask known identifiers, and generate text for manual review was time-consuming, but it was interesting and enjoyable. However, the manual review, while actually requiring fewer hours than writing the scripts, was taxing. The dullness makes it challenging to stay alert to catch the very few cases of identifiable data that slipped through automatic masking. Thus, we suggest that a fruitful direction of future work could be in further automating deidentification of keystroke data. This suggestion makes one major assumption: that public keystroke data is worth the effort. We remain enthusiastic that fine-grained CSEDM data is and will remain a boon to computing education research and education research in general, and believe that its value is justified by the extant literature. However, if its value needs further proof, we are delighted to release these datasets to the community for testing of its merits.

ACKNOWLEDGEMENTS

We thank Matt Davidson at the University of Washington for his great efforts in beta testing the data.

REFERENCES

2022. Title 45, Public Welfare, Department of Health and Human Services, Part 46, Protection of Human Subjects. https://www.hhs.gov/ohrp/regulations-and-policy/regulations/45-cfr-46/revised-common-rule-regulatory-text/index.html#46.102 (accessed 13 April, 2022).

AHADI, A., HELLAS, A., IHANTOLA, P., KORHONEN, A., AND PETERSEN, A. 2016. Replication in computing education research: researcher attitudes and experiences. In *Proceedings of the 16th Koli calling International Conference on Computing Education Research*. Association for Computing Machinery, 2–11.

AHADI, A., HELLAS, A., AND LISTER, R. 2017. A contingency table derived method for analyzing course data. *ACM Transactions on Computing Education (TOCE) 17,* 3, 1–19.

AHADI, A., LISTER, R., HAAPALA, H., AND VIHAVAINEN, A. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *ICER '15: Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, J. Sheard and Q. Cutts, Eds. Association for Computing Machinery, 121–130.

AHADI, A., LISTER, R., AND VIHAVAINEN, A. 2016. On the number of attempts students made on some online programming exercises during semester and their subsequent performance on final exam questions. In *ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 218–223.

ALLEN, L. K., MILLS, C., JACOVINA, M. E., CROSSLEY, S., D'MELLO, S., AND MCNAMARA, D. S. 2016. Investigating boredom and engagement during writing using multiple sources of information: the essay, the writer, and keystrokes. In *LAK16: Proceedings of the sixth International Conference on Learning Analytics & Knowledge*. Association for Computing Machinery, 114–123.

ALTADMRI, A. AND BROWN, N. C. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 522–527.

BARNES, T. AND STAMPER, J. 2008. Toward automatic hint generation for logic proof tutoring using historical student data. In *International Conference on Intelligent Tutoring Systems*, B. P. Woolf, E. Aïmeur, R. Nkambou, and S. Lajoie, Eds. Springer, 373–382.

BECKER, B. A. 2016. A new metric to quantify repeated compiler errors for novice programmers. In *ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 296–301.

BIXLER, R. AND D'MELLO, S. 2013. Detecting boredom and engagement during writing with keystroke analysis, task appraisals, and stable traits. In *IUI '13: Proceedings of the 2013 International Conference on Intelligent User Interfaces*. Association for Computing Machinery, 225–234.

BLIKSTEIN, P., WORSLEY, M., PIECH, C., SAHAMI, M., COOPER, S., AND KOLLER, D. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences 23,* 4, 561–599.

BRAUGHT, G. AND MIDKIFF, J. 2016. Tool design and student testing behavior in an introductory java course. In *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Association for Computing Machinery, 449–454.

BROWN, N. C., ALTADMRI, A., SENTANCE, S., AND KÖLLING, M. 2018. Blackbox, five years on: An evaluation of a large-scale programming data collection project. In *ICER '18: Proceedings of the 2018 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 196–204.

BROWN, N. C. AND KÖLLING, M. 2020. Blackbox mini-getting started with blackbox data analysis. In *SIGCSE '20: Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 1387–1387.

BROWN, N. C. C., KÖLLING, M., MCCALL, D., AND UTTING, I. 2014. Blackbox: A large scale repository of novice programmers' activity. In *SIGCSE '14: Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 223–228.

CARTER, A. S. 2012. Supporting the virtual design studio through social programming environments. In *ICER '13: Proceedings of the ninth annual International Conference on International Computing Education Research*. Association for Computing Machinery, 157–158.

CARTER, A. S. AND HUNDHAUSEN, C. D. 2017. Using programming process data to detect differences in students' patterns of programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. Association for Computing Machinery, 105–110.

CARTER, A. S., HUNDHAUSEN, C. D., AND ADESOPE, O. 2015. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *ICER'15: Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. Association for Computing Machinery, 141–150.

CARTER, A. S., HUNDHAUSEN, C. D., AND ADESOPE, O. 2017. Blending measures of programming and social behavior into predictive models of student achievement in early computing courses. *ACM Transactions on Computing Education (TOCE) 17,* 3, 1–20.

CASTRO-WUNSCH, K., AHADI, A., AND PETERSEN, A. 2017. Evaluating neural networks as a method for identifying students in need of assistance. In *SIGCSE'17: Proceedings of the 2017 ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 111–116.

COLLABORATION, O. S. 2015. Estimating the reproducibility of psychological science. *Science 349,* 6251, aac4716.

CROW, T., LUXTON-REILLY, A., AND WUENSCHE, B. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*. Association for Computing Machinery, 53–62.

DWYL. 2022. List of English Words. https://raw.githubusercontent.com/dwyl/english-words/master/words_alpha.txt (accessed 14 April, 2022).

EDWARDS, J., DITTON, J., SAINJU, B., AND DAWSON, J. 2020. Different assignments as different contexts: predictors across assignments and outcome measures in cs1. In *2020 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, 1–6.

EDWARDS, J., DITTON, J., TRNINIC, D., SWANSON, H., SULLIVAN, S., AND MANO, C. 2020. Syntax exercises in CS1. In *ICER '20: Proceedings of the 2020 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 216–226.

EDWARDS, J., HART, K., AND WARREN, C. 2022. A practical model of student engagement while programming. In *Proceedings of the 2022 ACM SIGCSE Technical Symposium on Computer Science Education*. Association for Computing Machinery, 558–564.

EDWARDS, J., LEINONEN, J., BIRTHARE, C., ZAVGORODNIAIA, A., AND HELLAS, A. 2020. Programming versus natural language: On the effect of context on typing in cs1. In *ICER'20: Proceedings of the 2020 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 204–215.

EDWARDS, J., LEINONEN, J., AND HELLAS, A. 2020. A study of keystroke data in two contexts: Written language and programming language influence predictability of learning outcomes. In *SIGCSE '20: Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 413–419.

EDWARDS, S. H., SNYDER, J., PÉREZ-QUIÑONES, M. A., ALLEVATO, A., KIM, D., AND TRETOLA, B. 2009. Comparing effective and ineffective behaviors of student programmers. In *ICER'09: Proceedings of the fifth International Workshop on Computing Education Research workshop*. Association for Computing Machinery, 3–14.

EPP, C., LIPPOLD, M., AND MANDRYK, R. L. 2011. Identifying emotional states using keystroke dynamics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 715–724.

ESTEY, A. AND COADY, Y. 2016. Can interaction patterns with supplemental study tools predict outcomes in cs1? In *ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 236–241.

ESTEY, A., KEUNING, H., AND COADY, Y. 2017. Automatically classifying students in need of support by detecting changes in programming behaviour. In *SIGCSE'17: Proceedings of the 2017 ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 189–194.

GAO, G., MARWAN, S., AND PRICE, T. W. 2021. Early performance prediction using interpretable patterns in programming process data. In *SIGCSE '21: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 342–348.

HELLAS, A., LEINONEN, J., AND IHANTOLA, P. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In *ITiCSE '17: Proceedings of the 2017 ACM conference on innovation and technology in computer science education*. Association for Computing Machinery, 238–243.

HUNDHAUSEN, C. D., OLIVARES, D. M., AND CARTER, A. S. 2017. Ide-based learning analytics for computing education: a process model, critical review, and research agenda. *ACM Transactions on Computing Education (TOCE) 17,* 3, 1–26.

IHANTOLA, P., VIHAVAINEN, A., AHADI, A., BUTLER, M., BÖRSTLER, J., EDWARDS, S. H., ISOHANNI, E., KORHONEN, A., PETERSEN, A., RIVERS, K., ET AL. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. *Proceedings of the 2015 ITiCSE on Working Group Reports*, 41–63.

JADUD, M. C. 2006. Methods and tools for exploring novice compilation behaviour. In *ICER '06: Proceedings of the second International Workshop on Computing Education Research*. Association for Computing Machinery, 73–84.

JADUD, M. C. AND DORN, B. 2015. Aggregate compilation behavior: Findings and implications from 27,698 users. In *ICER '15: Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. Association for Computing Machinery, 131–139.

KAZEROUNI, A. M., EDWARDS, S. H., HALL, T. S., AND SHAFFER, C. A. 2017. Deveventtracker: Tracking development events to assess incremental development and procrastination. In *ITiCSE '17: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 104–109.

KAZEROUNI, A. M., EDWARDS, S. H., AND SHAFFER, C. A. 2017. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, J. Tenenberg and L. Malmi, Eds. Association for Computing Machinery, 191–199.

KOŁAKOWSKA, A. 2016. Towards detecting programmers' stress on the basis of keystroke dynamics. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds. IEEE, 1621–1626.

KOUTCHEME, C., SARSA, S., HELLAS, A., HAARANEN, L., AND LEINONEN, J. 2022. Methodological considerations for predicting at-risk students. In *Australasian Computing Education Conference*. Association for Computing Machinery, 105–113.

KOVANOVIC, V., GAŠEVIĆ, D., DAWSON, S., JOKSIMOVIC, S., AND BAKER, R. 2015. Does time-on-task estimation matter? implications on validity of learning analytics findings. *Journal of Learning Analytics 2,* 3, 81–110.

KRISHNAMOORTHY, S., RUEDA, L., SAAD, S., AND ELMILIGI, H. 2018. Identification of user behavioral biometrics for authentication using keystroke dynamics and machine learning. In *ICBEA '18: Proceedings of the 2018 2nd International Conference on Biometric Engineering and Applications*. Association for Computing Machinery, 50–57.

KURVINEN, E., HELLGREN, N., KAILA, E., LAAKSO, M.-J., AND SALAKOSKI, T. 2016. Programming misconceptions in an introductory level programming course exam. In *ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 308–313.

LEINONEN, A., NYGREN, H., PIRTTINEN, N., HELLAS, A., AND LEINONEN, J. 2019. Exploring the applicability of simple syntax writing practice for learning programming. In *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 84–90.

LEINONEN, J., CASTRO, F. E. V., AND HELLAS, A. 2021. Does the early bird catch the worm? earliness of students' work and its relationship with course outcomes. In *ITiCSE '21: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. Association for Computing Machinery, 373–379.

LEINONEN, J., CASTRO, F. E. V., AND HELLAS, A. 2022. Time-on-task metrics for predicting performance. In *SIGCSE '22: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 871–877.

LEINONEN, J., CASTRO, F. E. V., HELLAS, A., ET AL. 2021. Fine-grained versus coarse-grained data for estimating time-on-task in learning programming. In *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*, S. Hsiao, S. Sahebi, F. Bouchet, and J.-J. Vie, Eds. The International Educational Data Mining Society, 648–653.

LEINONEN, J., LEPPÄNEN, L., IHANTOLA, P., AND HELLAS, A. 2017. Comparison of time metrics in programming. In *ICER '17: Proceedings of the 2017 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 200–208.

LEINONEN, J., LONGI, K., KLAMI, A., AHADI, A., AND VIHAVAINEN, A. 2016. Typing patterns and authentication in practical programming exams. In *ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 160–165.

LEINONEN, J., LONGI, K., KLAMI, A., AND VIHAVAINEN, A. 2016. Automatic inference of programming performance and experience from typing patterns. In *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Association for Computing Machinery, 132–137.

LONGI, K., LEINONEN, J., NYGREN, H., SALMI, J., KLAMI, A., AND VIHAVAINEN, A. 2015. Identification of programmers from typing patterns. In *Proceedings of the 15th Koli Calling conference on computing education research*. Association for Computing Machinery, 60–67.

MCCALL, D. AND KÖLLING, M. 2019. A new look at novice programmer errors. *ACM Transactions on Computing Education (TOCE) 19,* 4, 1–30.

MIHAESCU, M. C. AND POPESCU, P. S. 2021. Review on publicly available datasets for educational data mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 11,* 3, e1403.

MONACO, J. V., PEREZ, G., TAPPERT, C. C., BOURS, P., MONDAL, S., RAJKUMAR, S., MORALES, A., FIERREZ, J., AND ORTEGA-GARCIA, J. 2015. One-handed keystroke biometric identification competition. In *2015 International Conference on Biometrics (ICB)*. IEEE, 58–64.

MORALES, A. AND FIERREZ, J. 2015. Keystroke biometrics for student authentication: A case study. In *ITiCSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 337–337.

NGUYEN, Q. 2020. Rethinking time-on-task estimation with outlier detection accounting for individual, time, and task differences. In *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge*. Association for Computing Machinery, 376–381.

PIECH, C., SAHAMI, M., HUANG, J., AND GUIBAS, L. 2015. Autonomously generating hints by inferring problem solving policies. In *L@S '15: Proceedings of the second (2015) ACM Conference on Larning@ Scale*. Association for Computing Machinery, 195–204.

PIECH, C., SAHAMI, M., KOLLER, D., COOPER, S., AND BLIKSTEIN, P. 2012. Modeling how students learn to program. In *SIGCSE '12: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 153–160.

PRICE, T., ZHI, R., AND BARNES, T. 2017. Evaluation of a data-driven feedback algorithm for open-ended programming. In *Proceedings of The 10th International Conference on Educational Data Mining (EDM 2017)*, X. Hu, T. Barnes, A. Hershkovitz, and L. Paquette, Eds. International Educational Data Mining Society, 192–197.

PRICE, T. W., DONG, Y., AND LIPOVAC, D. 2017. isnap: towards intelligent tutoring in novice programming environments. In *SIGCSE'17: Proceedings of the 2017 ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 483–488.

PRICE, T. W., HOVEMEYER, D., RIVERS, K., GAO, G., BART, A. C., KAZEROUNI, A. M., BECKER, B. A., PETERSEN, A., GUSUKUMA, L., EDWARDS, S. H., ET AL. 2020. Progsnap2: A flexible format for programming process data. In *ITiCSE '20: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 356–362.

RIVERS, K., HARPSTEAD, E., AND KOEDINGER, K. R. 2016. Learning curve analysis for programming: Which concepts do students struggle with? In *ICER '16: Proceedings of the 2016 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 143–151.

RIVERS, K. AND KOEDINGER, K. R. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education 27,* 1, 37–64.

RODRIGUEZ-RIVERA, G., TURKSTRA, J., BUCKMASTER, J., LECLAINCHE, K., MONTGOMERY, S., REED, W., SULLIVAN, R., AND LEE, J. 2022. Tracking large class projects in real-time using fine-grained source control. In *SIGCSE '22: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 565–570.

SHRESTHA, R., LEINONEN, J., HELLAS, A., IHANTOLA, P., AND EDWARDS, J. 2022. Codeprocess charts: Visualizing the process of writing code. In *Australasian Computing Education Conference*. Association for Computing Machinery, 46–55.

SHRESTHA, R., LEINONEN, J., ZAVGORODNIAIA, A., HELLAS, A., AND EDWARDS, J. 2022. Pausing While Programming: Insights From Keystroke Analysis. In *ACM International Conference on Software Engineering (ICSE), Software Engineering Education and Training (SEET) track*. Association for Computing Machinery, 187–198.

SMITH, R. AND RIXNER, S. 2019. The error landscape: Characterizing the mistakes of novice programmers. In *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 538–544.

SPACCO, J., DENNY, P., RICHARDS, B., BABCOCK, D., HOVEMEYER, D., MOSCOLA, J., AND DUVALL, R. 2015. Analyzing student work patterns using programming exercise data. In *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 18–23.

THOMAS, R., KENNEDY, G., DRAPER, S., MANCY, R., CREASE, M., EVANS, H., AND GRAY, P. 2003. Generic usage monitoring of programming students. In *Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education*, G. Crisp, D. Thiele, I. Scholten, S. Barker, and J. Baron, Eds. Vol. 3. ASCILITE.

THOMAS, R. C., KARAHASANOVIC, A., AND KENNEDY, G. E. 2005. An investigation into keystroke latency metrics as an indicator of programming performance. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, A. Young and D. Tolhurst, Eds. Association for Computing Machinery, 127–134.

TIAM-LEE, T. J. AND SUMI, K. 2019. Analysis and prediction of student emotions while doing programming exercises. In *International Conference on Intelligent Tutoring Systems*, A. Coy, Y. Hayashi, and M. Chang, Eds. Springer, 24–33.

VIHAVAINEN, A., VIKBERG, T., LUUKKAINEN, M., AND PÄRTEL, M. 2013. Scaffolding students' learning using test my code. In *ITiCSE '13: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, 117–122.

VINKER, E. AND RUBINSTEIN, A. 2022. Mining code submissions to elucidate disengagement in a computer science mooc. In *LAK22: 12th International Learning Analytics and Knowledge Conference*. Association for Computing Machinery, 142–151.

VIZER, L. M., ZHOU, L., AND SEARS, A. 2009. Automated stress detection using keystroke and linguistic features: An exploratory study. *International Journal of Human-Computer Studies 67,* 10, 870–886.

WARREN, J., RIXNER, S., GREINER, J., AND WONG, S. 2014. Facilitating human interaction in an online programming course. In *SIGCSE '14: Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 665–670.

WATSON, C., LI, F. W., AND GODWIN, J. L. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies (ICALT)*. IEEE, 319–323.

YAN, L., HU, A., AND PIECH, C. 2019. Pensieve: Feedback on coding process for novices. In *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, 253–259.

ZAVGORODNIAIA, A., SHRESTHA, R., LEINONEN, J., HELLAS, A., AND EDWARDS, J. 2021. Morning or evening? an examination of circadian rhythms of cs1 students. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 261–272.