Pensieve: Feedback on coding process for novices

Lisa Yan
Stanford University
yanlisa@stanford.edu

Annie Hu
Stanford University
anniehu@stanford.edu

Chris Piech
Stanford University
piech@cs.stanford.edu

ABSTRACT

In large undergraduate computer science classrooms, student learning on assignments is often gauged only by the work on their final solution, not by their programming process. As a consequence, teachers are unable to give detailed feedback on how students implement programming methodology, and novice students often lack a *metacognitive* understanding of how they learn. We introduce *Pensieve* as a drag-and-drop, open-source tool that organizes snapshots of student code as they progress through an assignment. The tool is designed to encourage sit-down conversations between student and teacher about the programming process. The easy visualization of code evolution over time facilitates the discussion of intermediate work and progress towards learning goals, both of which would otherwise be unapparent from a single final submission. This paper discusses the pedagogical foundations and technical details of *Pensieve* and describes results from a particular 207-student classroom deployment, suggesting that the tool has meaningful impacts on education for both the student and the teacher.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; **Student assessment**;

KEYWORDS

Programming courses; assessment; pedagogy; formative feedback; metacognition

ACM Reference Format:
Lisa Yan, Annie Hu, and Chris Piech. 2019. Pensieve: Feedback on coding process for novices. In *Proceedings of 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, February 27–March 2, 2019 (SIGCSE'19),* 7 pages. https://doi.org/10.1145/3287324.3287483

1 INTRODUCTION

Assignment feedback is a critical component of student learning [19, 34]. Given that one of the primary learning goals of CS1 is to teach students *how* to solve programming challenges, it would be immensely useful to provide formative feedback on *how* a student worked through solving an assignment. Yet for a variety of reasons, many contemporary classrooms only provide summative feedback on a student's final answer [23, 41]. The hours during which a student actively learns and interacts with the material are manifested in a single deliverable—a single snapshot into the student's thinking—from which an instructor must glean enough information to discuss student process. This misses the opportunity to give students feedback on their problem solving approaches and to help students develop metacognitive abilities [5, 9, 30, 34]. However, providing feedback on a final student submission is hard, and providing feedback on the hundreds of steps a student takes to get to their answer seems prohibitively difficult.

To enable feedback on student progress, we built *Pensieve*, a simple-to-use tool that provides an interactive visualization of student work history over the course of completing an assignment (Figure 1). *Pensieve* gives both students and teachers a means to see assignment progress—from the time a student first looked at assignment starter code to when they submit the final product—facilitating conversation around metacognition and student learning that is critical for introductory computer science learners. When teachers can observe a student's process, they are able to give timely feedback addressing student mistakes and to adjust and personalize their own teaching [6]. When students observe their own process—even in the absence of the instructor—they can internalize metacognitive observations [9]. Through our streamlined user interface, understanding progress becomes quick and feasible, even in a large classroom.

In this paper, we begin with an overview of the pedagogical motivations for the *Pensieve* tool (Sections 2 and 3). We then discuss the technical implementation details in Section 4. In Section 5, we share positive experiences by both teachers and students when *Pensieve* was deployed in a 10-week course. The tool was correlated with significantly better student performance, with a particularly large benefit for students with the least programming background. Furthermore, there was almost no additive effort for the teaching assistants to learn the tool, nor was it difficult to incorporate into the existing course pipeline. Finally, we close in Section 6 with best practices and discussion of how our tool can be used in many CS1 classrooms today. *Pensieve* is an open source tool which you can download and modify.[1]

2 PEDAGOGY AND MOTIVATION

Pensieve aims to "push back" somewhat against the trend of automated grading tools in classrooms by thoughtfully integrating a human grader into assignment feedback. We designed the tool with several objectives in mind:

(1) Foster metacognitive skills
(2) Identify methodology errors early
(3) Counteract plagiarism effects in a large classroom
(4) Gentle introduction of version control.

Our first objective is to develop metacognition in the way students learn computer science. A key finding of a landmark National

[1] https://github.com/chrispiech/pensieve

SIGCSE'19, February 27–March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287483

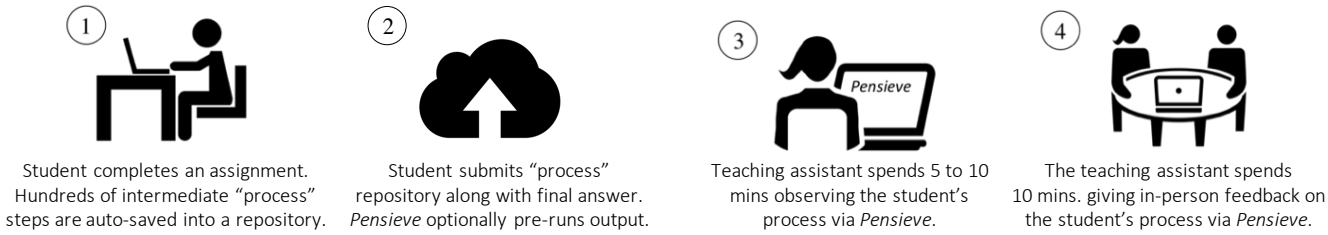

Figure 1: Pensieve allows teachers to visualize and give feedback on *how* a student progresses through an assignment.

Academy of Sciences study on learning science was the effectiveness of a metacognitive approach to instruction [7]. *Metacognition* is a learning theory for "thinking about thinking." To most effectively learn, students must not only understand the problem but also understand and reflect on where they are in the problem-solving process. Metacognitive practices directly correspond to increasing students' performance in evaluating concepts, one of the highest cognitive skill levels in Bloom's Revised Taxonomy for learning [2, 20, 39]. Research has also shown that students with stronger metacognitive awareness tend to perform better on programming tasks [4, 12, 18]. Importantly, metacognition directly supports the concept of a *growth mindset*, the theory that intelligence can be developed with experience [15]. Students who believe that ability is a fixed trait are at a significant disadvantage in STEM fields compared to their peers who believe in a growth mindset.

A related objective of our tool is to encourage early identification of methodology errors. Most assignments in large-scale computer science classes are assessed summatively: the teaching assistant sees and grades only the students' final submission. However, research has conclusively shown that nongraded *formative assessments*, or ongoing assessments designed to make students' thinking visible to both teachers and students, are key to improved learning [7, 22]. They allow teachers to identify which thought processes work for students and to provide useful, directed feedback so that each student can improve. As such, this goal is intertwined with our goal of fostering metacognition in computer science—through reflecting critically on their programming process, students will be able to both identify areas for improvement and understand how to achieve that improvement earlier.

Finally, by emphasizing the importance of the learning process, we hope that *Pensieve* can act as a preemptive deterrent to potential plagiarism. Students who plagiarize all or parts of their assignments stunt their metacognitive development in programming and reap fewer benefits from formative feedback. They may also become stuck in a cycle of plagiarism in which they are increasingly unable to complete work independently [40]. Many current approaches to combating plagiarism in large CS classrooms focus on detecting similar code in the final submission; however, this further reinforces the importance of the final grade received above the intrinsic value of learning how to learn, a way of thinking that underlies much student plagiarism [8]. We hope that by monitoring the development process [40], we can better support students who may otherwise feel overwhelmed and driven to plagiarize code.

3 RELATED WORK

In many large classrooms, the technology focus is on Automated Assessment Tools (AATs), which relegate a significant portion of the feedback pipeline to tools that support teacher- or student-written unit tests [1, 16, 21]. Larger classrooms like massive open online courses push towards full-automation by systems like intelligent tutoring systems to personalize the learning experience for online students [14]. In our study, we focus on large CS1 courses that have in-classroom human teacher support. While there is research on how to automate computer agents to personalize debugging hints per student [36], our work seeks solutions that allow humans to provide higher-level feedback on problem solving process.

Novice programmers benefit from metacognitive awareness. Loksa et al. found that students who are trained in problem solving procedures are more productive and have higher self-efficacy [28]. There are several metacognitive tools that have been studied in a CS context: Lee et al. found that personifying the programming process increases online engagement with a coding task [26]; Prather et al. discuss how metacognitive awareness ties into how students use AATs [32]; and Marceau et al. studied how novice programmers interact with error messages [29]. Outside of CS classrooms, Tanner et al. reported on generalizable teaching practices for promoting metacognition, such as instructor modeling of problem solving, tools to help students identify learning strategies, and guided reflection [38]. *Pensieve* incorporates all three of these strategies by engaging students—with the support of teaching assistants—with their own metacognitive understanding of computer science.

Formative feedback in the classroom can have a variety of impacts [3]. Van der Kleij et al. found that detailed feedback contributes more to student learning than feedback on correctness does [41]. Students also benefit more from an assignment when they have interactive, dialogue-based critiques with peers or instructors [9, 30]. We design our tool with the awareness that feedback is not a one-sided conversation; teachers should discuss with the students to promote student self-regulation of learning for the rest of the course [6].

Other classrooms have used version control systems to give formative feedback, as version control simplifies management of large courses [11] and makes it easier to identify problems in work habits and progress [24, 25, 35]. We take a slightly different approach, placing less emphasis on learning *how* to manage a version control system than on enabling self-evaluation from students and good feedback from teachers [27]. Our tool focuses on surfacing this information in a clear manner, providing a light introduction to the benefits of version control.

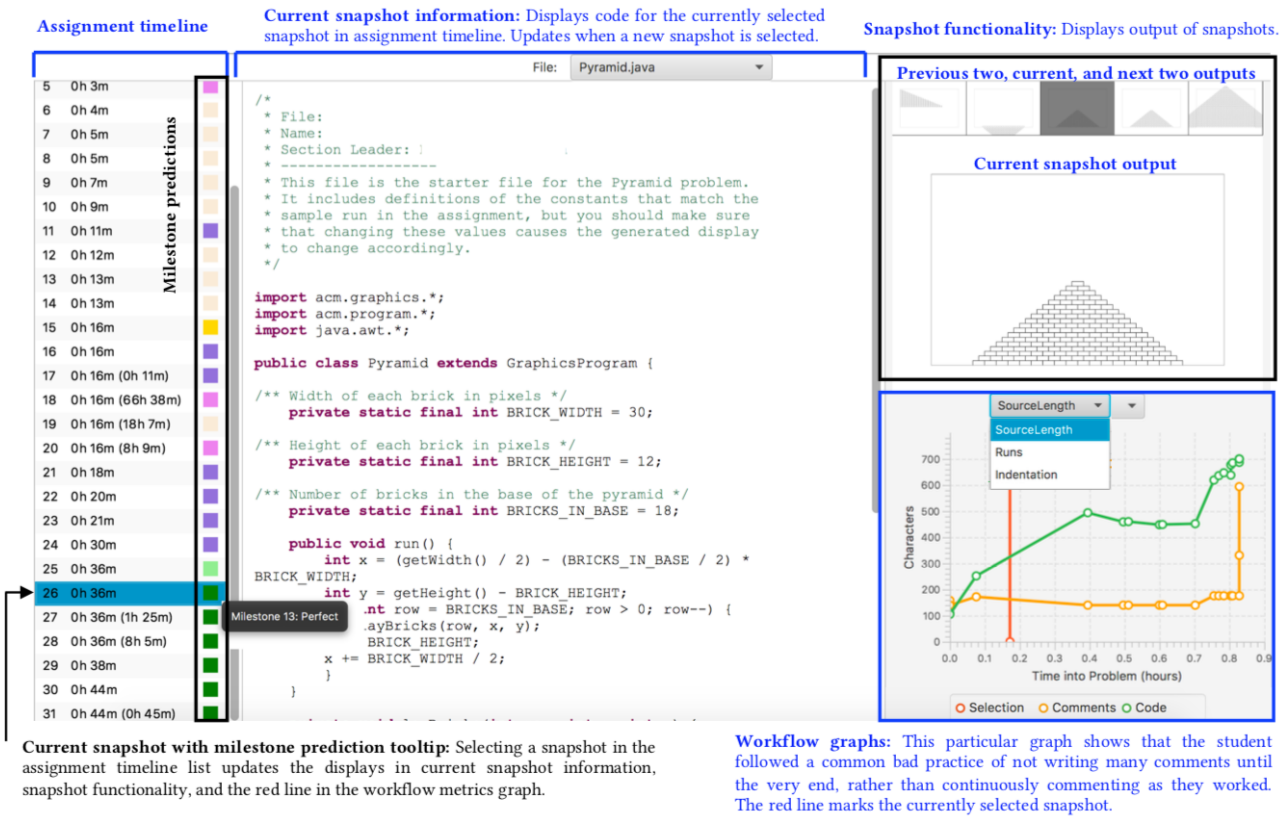

Figure 2: Diagram of *Pensieve* display, composed of four main components (discussed more in Section 4): Assignment timeline (left), Current snapshot information (center), Snapshot functionality (top right), and Workflow graphs (bottom right).

4 PENSIEVE DETAILS

Pensieve is primarily designed for educators to easily give feedback on student process. A pipeline of how it fits into a classroom is shown in Figure 1. As a student works on an assignment, their process is recorded as a local repository which is submitted along with their final answer. A screenshot of the *Pensieve* tool is shown in Figure 2. To use *Pensieve*, loaded with a student's process, the instructor interacts with the four main components of the tool: a timeline of snapshots for this program file (left panel), a selectable, highlightable text view of the snapshot code (center), the visual output corresponding to running the snapshot code (top-right), and metrics of this code file over time (bottom-right).

As part of our efforts to minimize instructor workload, *Pensieve* is designed to be a drag-and-drop, out-of-the-box tool for viewing student assignment progress. It is ported as a JAR file; an instructor can simply download the JAR into a student assignment folder, run it, and begin viewing student intermediate snapshot data.

4.1 Tool Implementation

The only required folder for *Pensieve* to work is a repository of *timestamped code* snapshots of a single student working on an assignment—data which is increasingly available to educators [33]. We currently represent timestamped code using the git abstraction: we customized the Eclipse IDE students use to automatically save student code at compile time (as described in [31]). However, *Pensieve* depends neither on the use of Eclipse nor Java in the classroom; it merely requires that student work be cached over time in a timestamped code folder. Our tool can work at a coarser granularity, such as on assignments that require students to commit periodically to an online storage platform like GitHub. The tool can also operate on setups like those on Code.org, which save snapshots whenever students run their code. We utilize the timestamped code repository to analyze both student code progress and timing information. In this section, we discuss interacting with the *Pensieve* tool for a single file in the timestamped code folder.

4.1.1 Assignment timeline. The leftmost panel of the tool (Figure 2) displays all captured snapshots for a given student file. Each entry in the timeline contains the snapshot index (in temporal order), the amount of time spent so far on this file, break time, and an optional color key indicating functional progress (discussed more in Section 4.1.4). Time spent on the assignment is calculated as an aggregate of relative timing information between this snapshot and the previous one. We mark break time in parentheses, where a break occurs when two consecutive snapshots' timestamps differ by more than 10 minutes. By browsing the timeline, an instructor can infer where students took substantial breaks to get a cursory glance of which snapshots would be worth closer attention.

4.1.2 Current snapshot information. Clicking on a snapshot entry in the Assignment timeline panel displays that snapshot's code in the center panel with appropriate syntax highlighting. Other than syntax and comment highlighting, the center panel is plaintext, allowing an instructor to select and copy code if additional verification is needed. The top-right panel is used to display this code's output when compiled and run; if the code has a compile or runtime error, nothing is shown. The example assignment shown in Figure 2 draws a pyramid-like object using Java's ACM library. It is important to note that Pensive is not running snapshot code live; the compiling and running of any student code is done in a preprocessing step. The details of designing such a preprocessor are discussed more in Section 4.1.4.

4.1.3 Workflow graphs. The bottom-right panel contains time series of various file metrics to visualize student progress and style over time. The SourceLength graph shown in Figure 2 displays code length (in green) and comment length (in yellow) in number of characters, as well as a red, vertical indicator of the currently selected snapshot's metrics. Another graph displays indentation errors over time—which are allowed in Java but are indicative of messy code—and a third graph displays custom metadata from the timestamped code directory; for example, our repositories also record the number of code runs per snapshot. Instructors can easily toggle between these time series, which are intended to highlight student work patterns, such as when they started thinking about good style and indentation, or whether the majority of their work and code changes were concentrated at the end of the timeline.

4.1.4 Snapshot functionality. The top-right panel visualizes the output of the currently selected code, if there were no compile or runtime errors. All outputs for all snapshots are run and saved prior to downloading and running *Pensieve* in a preprocessing step. In our implementation, this is performed on the course servers after students submit the assignment and before instructors begin grading. The preprocessor compiles and run each snapshot in each file; if there are no errors, then the output is saved into a separate folder. The *Pensieve* program then interfaces with the folder of outputs via a JSON metadata lookup, which associates the snapshot (represented by a (Unix Epoch timestamp, original filename) tuple) with the following features: a flag for compile error, a flag for runtime error, the output file path, and a milestone metric, of which the latter two are only valid if both error flags are off. Any additional metadata per snapshot can also be saved in this JSON lookup file.

In our example assignment, we mainly use graphics exercises based on the Java ACM library. As a result, output file paths point to saved output PNG files. However, one could easily run and save console program output as plain-text log-files, which can be displayed in the top-right panel of *Pensieve* with minor modifications. Milestone metrics are computed based on each snapshot's output; if there are unit tests for the assignment, this can simply be the number of passed unit tests. For our image files, we had an image classifier that determined a milestone number for each snapshot [42]. The milestone metric per snapshot is displayed as a small square color indicator in the left timeline panel; runtime and compile errors are assigned separate color indicators.

4.2 Classroom use

Pensieve is split into phases (Figure 1): first, a student works in their own environment (a lab computer, personal laptop, or in-browser app), which saves timestamped snapshots of their code progress. The student then submits their work to a database, where optional preprocessing occurs; for example, to generate and save snapshot output for later review. Then, the teacher downloads and reviews the student code submissions using *Pensieve*, and finally discusses the code with the student in a classroom.

In our classroom, eight students are assigned to a teaching assistant for the duration of the quarter; the teaching assistant's weekly responsibilities are to teach discussion sections and grade assignment work. The graders download the student code from the course database, typically a few days prior to the grading deadline, which is a week after the student submission deadline. At the time of grader download, *Pensieve* is already included as part of each student folder. Any preprocessing to generate visual program output or console output logs must be included in the student folders prior to download; this is done to ensure that the grader can run *Pensieve* as a lightweight tool on their own computers. The preprocessing step to generate and save graphical output on three separate code files for a 400-student class took about half a day. In the absence of a course database or preprocessor, a teacher can simply drop the *Pensieve* JAR into the assignment directory on the students' computer. Once the JAR is within the correct directory, teachers and students can navigate the timestamped code repository and discuss coding tips and misconceptions.

Teachers in our classroom provide assignment feedback to their students in two ways: the functional and style grades produced from a fixed rubric, and an *interactive grading* (IG) session [37], where the teacher and student sit down for a one-on-one, 15-minute session to discuss the student's code, any misconceptions, and tips to improve for future assignments. Teachers can use *Pensieve* during these IG sessions to identify, for example, places where students struggled with concepts or showed good style habits, and to gain a holistic view of how the student thought through the assignment. Grading is done independently from IG session outcomes; the purpose of the IG session is to maintain a conversation throughout the quarter so that students are actively reflecting on their coding progress.

In addition to the teacher-facing version of *Pensieve*, we provide a reduced version of the tool to the students so that they can look at their prior work and revert back to a previous version if needed *as they work on the assignment*; our tool at this point functions more as a welcoming, light version control software for students.

5 EXPERIENCE

At the time of writing this report, *Pensieve* has been used in two terms: Winter 2018 and Spring 2018. In the first term we refined the tool and teacher training procedures. In the second term (which we will refer to as the **Experience Term**), we evaluated the impact of the tool for the 207 students who took CS1.

We sought to evaluate the impact of our tool in three ways: (1) a formal qualitative analysis of how useful instructors found the tool, (2) an official university survey on how useful students found the tool, and (3) quantitative measures of performance on exams, time to complete assignments and honor code violations. All evaluations

Table 1: (a) Positive teacher feedback; (b) ways to improve.

(a)

% Agree	Item
90	Insights from *Pensieve* lead to "more actionable and specific assistance"
70	*Pensieve* "is helpful for showing instructors which students might need extra help"
–	Data on how long it takes students to complete assignments are "especially valuable"

(b)

% Agree	Item
80	Want an option "to show only major changes in code" to better prioritize information and feedback
75	Want more "student-facing features"
–	"All of the data *Pensieve* presents is helpful but can be overwhelming"

tell a consistent story of educational benefit that we expected to see given the improved pedagogical benefits.

5.1 Student and Teacher Perceptions

At the end of the Experience Term, we asked students to provide a Likert rating for the statement, "It was useful to see the process of how I learned using *Pensieve*," as part of their formal course evaluation. Students on average strongly agreed ($\mu = 4.6/5, \sigma = 0.6$). Furthermore, students gave a higher rating for the quality of course instruction and feedback ($\mu = 4.8, \sigma = 0.4$, where 5 = Excellent, and 4 = Good) than in previous terms ($\mu = 4.6, \sigma = 0.4$) for the Experience Term instructor. Response rate was 70%.

To measure how useful teaching assistants found our tool, we requested an external facilitator to conduct a small group instructional diagnosis (SGID) [10]. External evaluators met with 31 teaching assistants who used Pensieve in the Experience Term to discuss (1) whether using the tool improved the teachers' understanding of student learning process in CS, (2) if there were ways to improve the tool, and (3) if there were ways to improve how the tool was used in the classroom. The evaluators solicited feedback and consolidated major sentiments. To minimize bias, the instructor and researchers were not present during the SGID. The teaching assistants articulated that they found the tool to be useful, despite it requiring more work on their part (Table 1). The feedback session suggests that more teacher training or better highlighting within the tool would improve the teacher experience.

5.2 Learning Analysis

In addition to measuring the perception of students and teachers, we also place importance on quantifiable improvements in learning outcomes. Under our hypothesis that *Pensieve* had a substantial impact on student learning, we expected to see a notable change in student performance.

All measures are between the Experience Term ($N = 207$ students) where we deployed *Pensieve* and a Baseline Term with the same instructor, lectures and assignments, but without using the *Pensieve* tool ($N = 498$ students). The students in the Baseline Term had significantly **more** CS background than in the Experience Term. 37.4% of students in the Baseline Term reported > 10 hours of programming experience, vs 27.7% of students in the Experience Term. The Experience Term was 52% female vs 51% female in the baseline.

In the Experience Term, we provided our tool to the entire class (as opposed to running a randomized control trial), thus it is not possible to report on the causal impact of using *Pensieve*. While we could not observe causality, we can observe correlation. The co-occurrence of using the tool and notable learning improvement adds weight to our belief that the tool has a positive impact. To assess the co-occurrence of learning gains, we measured changes in (1) time spent on assignments after the *Pensieve*-assisted interactive grading assignment (2) exam performance and (3) plagiarism.

Assignment time: We used student assignment completion time as a pre-post measure of their ability to program. For each of Assignments 1, 2, and 3, we used students' intermediate progress records to calculate how long the assignment took (Note: students received the *Pensieve* "intervention" just before the due date for Assignment 2). The Experience Term saw a significant decrease in the number of hours spent on Assignment 3, the assignment due after the intervention (from an average completion time of 7.0 hours down to 6.3 hours, in Figure 3a). This decrease was particularly large given that in the Experience Term, students were slower on average to complete assignments *prior* to the Pensieve intervention. To account for this difference we calculate how long students in the Experience Term took to complete Assignment 3 (X_3) compared to their expected completion time had they been in the Baseline Term, given how long they spent on Assignment 1 ($\hat{X_3}|X_1$):

$$\begin{aligned}\text{Decrease in Assn 3 time} &= E[\hat{X_3}|X_1] - E[X_3] \\ &= 48 \text{ mins } (p < 0.0001)\end{aligned}$$

We observe an increase in Assignment 3 grades between the Baseline and Experience Term, but the change is not significant ($\delta =$ 3.2pp, $p = 0.07$). Most students in the Baseline Term already received high grades ($\mu = 98\%$ in Baseline, $\mu = 101\%$ in Experience).

Exam ability: On its own, the ability to complete an assignment faster does not indicate that students have learned more. Another measure is the difference in how well students performed on the class midterm (given just after the due date for Assignment 3). Since the exams were not identical we evaluated the difficulty (on a consistent scale) of each exam question and used Item Response Theory [17] to calculate a midterm "ability" score for each student, which would be comparable across quarters. We define a student's score for each question on the midterm exam as $S_{i,j} = n_j \cdot \sigma(a_i - d_j)$ where $S_{i,j}$ is the score of student i on question j, n_j is the number of points on the question, d_j is the difficulty of the question, and a_i is the ability of the student. We can then reverse calculate ability (a_i) given their observed score and problem difficulty. The difference in exam ability between terms is shown in Figure 3b. Student midterm abilities, measured on a scale from 0 to 10 increased from an average of 6.9 to an average of 7.6 ($p < 0.0001$).

Plagiarism: One of the theorized impacts of using Pensieve is that it would create a culture where plagiarism would be less prevalent: it is much harder to cheat if you are going to be presented with your process. For Assignment 3, trajectory-based plagiarism

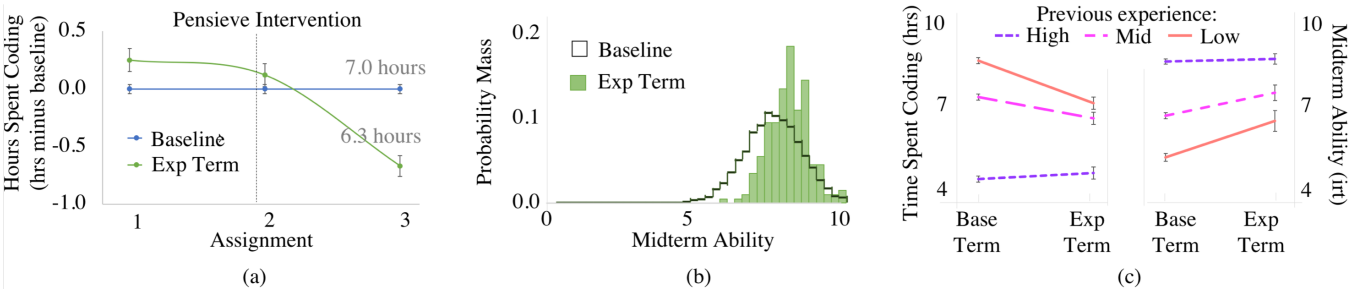

Figure 3: The difference in performance between the Baseline Term and the Experience (Exp) Term, where we deployed Pensieve. (a) Change in time to complete assignments. (b) Change in midterm ability. (c) Changes broken down for students with High, Mid and Low levels of initial CS background for: (left) time to complete assignment 3 and (right) midterm ability. All error bars are one standard error of the mean.

detection (TMOSS) shows a small decrease (from 4.3% of the class down to 3.9%) compared against previously published numbers from the same course [43].

Who benefits: Given that background knowledge outweighs many other factors for predicting student performance in CS1, we disaggregated our assignment time results and exam ability results based on the prior knowledge of students. We split students into three equal-sized terciles (Low, Mid, High) based on a background statistic, computed as a weighted sum of normalized: reported hours of experience (50%), Assignment 1 grade (30%) and Assignment 1 work time in hours (20%). Based on these splits we can see that the benefits of Pensieve were mainly experienced by the Low tercile (Figure 3c). The Mid tercile also had significant improvements; however the top tercile had negligible changes in midterm ability and assignment work time between the Baseline Term and the Experience Term.

Despite these promising results with *Pensieve*, there are many uncontrollable, confounding factors. In particular, teaching assistants are not required to use our tool, and it is highly likely that those who effectively use the tool are more effective teachers to begin with. Instead of taking our analysis as conclusive proof of the efficacy of *Pensieve*, it is better to see these results as a positive indication that such a tool improves the student learning experience.

6 BEST PRACTICES

In this section, we discuss best practices of *Pensieve* that were gleaned from our continued use of the tool and the conversations from the SGID sessions with teaching assistants.

When to use *Pensieve*: With the goal of developing student metacognition at an early stage, deploy *Pensieve* for the first few programming projects in a course. This enables teachers to identify process errors early on and recommend good programming practices, and develop a student's metacognition.

How to use *Pensieve*: *Pensieve* can be deployed in many ways. We believe that it is most valuable when teacher and student sit down together and use the tool to facilitate discussion. This type of conversation helps a teacher identify struggling students early on, so that they can be monitored and helped through the course. In our experience, keeping these short sessions ungraded helps to create an environment where students can ask questions about their own learning. Alternatively, the tool could be used for remote feedback, though we expect the missing human conversation will limit the impact on student learning.

Teacher training: Since our goal was to use *Pensieve* in student-teacher interactions, it is important to train teachers to use this tool effectively. Instruct teachers to foster conversations with students about problem solving techniques. We specifically focused on teaching teachers how to talk about top down "decomposition" and "iterative testing" [13]. We recommend teachers use the graphs to efficiently find the most "important" snapshots of progress—snapshots that illustrate conceptual changes.

Student messaging: *Pensieve* works best with fine-grained code snapshots of student process; naturally this brings up questions of student privacy. When presenting the tool to students, we made clear that our intentions were to help students learn as much as possible. Just like in other classes, the more a student shows their work the more helpful feedback a teacher can give. Similarly, when introducing *Pensieve* to students, it is an opportunity to explain that plagiarism is much more obvious when a teacher is looking at one's progress.

Currently, Pensieve is most valuable with a human teacher in the feedback process. There are many potential extensions to the tool that would add clearer information extraction and reduce teachers' burden, such as highlighting notable trends on the Workflow graphs or code diffs between snapshots. Still, it is an important first step in providing more formative, metacognitive feedback to students, and paves the way for better automated feedback in the future.

7 CONCLUSION

Perhaps one of the most important things *Pensieve* brings to the table is the ability to view progress in a tool that is useful for both the teacher *and* the student. This is a way to begin the conversation on metacognition—and a way to bring a human aspect back to programming despite the size of classrooms. Given how much time is needed to learn how to code, early feedback for beginning programmers is a highly efficient use of scarce resources, even for classrooms who do not have a low student-teacher ratio.

Students and teaching assistants had overwhelmingly positive feedback about using *Pensieve* in the classroom, and the courses where it was deployed had significant learning improvements. We invite you to use and improve on this novel, open-source way to give feedback on coding process.

REFERENCES

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

[2] Lorin W. Anderson and David R. Krathwohl (Eds.). 2001. *A Taxonomy for Learning, Teaching, and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives* (2 ed.). Allyn & Bacon, New York.

[3] Randy Elliot Bennett. 2011. Formative assessment: A critical review. *Assessment in Education: Principles, Policy & Practice* 18, 1 (2011), 5–25.

[4] Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the Role of Self-Regulated Learning on Introductory Programming Performance. In *Proceedings of the first international workshop on Computing education research (ICER '05)*. ACM, New York, NY, USA, 81–86. https://doi.org/10.1145/1089786.1089794

[5] Paul Black and Dylan Wiliam. 2009. Developing the theory of formative assessment. *Educational Assessment, Evaluation and Accountability(formerly: Journal of Personnel Evaluation in Education)* 21, 1 (23 Jan 2009), 5. https://doi.org/10.1007/s11092-008-9068-5

[6] David Boud and Elizabeth Molloy. 2013. Rethinking models of feedback for learning: the challenge of design. *Assessment & Evaluation in Higher Education* 38, 6 (2013), 698–712. https://doi.org/10.1080/02602938.2012.691462

[7] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. 2000. *How People Learn*. National Academy Press.

[8] Tracey Bretag. 2013. Challenges in Addressing Plagiarism in Education. *PLoS Medicine* 10, 12 (2013).

[9] David Carless, Diane Salter, Min Yang, and Joy Lam. 2011. Developing sustainable feedback practices. *Studies in Higher Education* 36, 4 (2011), 395–407. https://doi.org/10.1080/03075071003642449

[10] D. Joseph Clark and Mark V. Redmond. 1982. Small group instructional diagnosis. ERIC.

[11] Curtis Clifton, Lisa C. Kaczmarczyk, and Michael Mrozek. 2007. Subverting the Fundamentals Sequence: Using Version Control to Enhance Course Management. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*. ACM, 86–90.

[12] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. 2010. Manipulating Mindset to Positively Influence Introductory Programming Performance. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, New York, NY, USA, 431–435. https://doi.org/10.1145/1734263.1734409

[13] Nell B. Dale. 2006. Most difficult topics in CS1: results of an online survey of educators. *ACM SIGCSE Bulletin* 38, 2 (2006), 49–53.

[14] Thanasis Daradoumis, Roxana Bassi, Fatos Xhafa, and Santi Caballé. 2013. A Review on Massive E-Learning (MOOC) Design, Delivery and Assessment. *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (2013), 208–213.

[15] Carol Dweck. 2008. *Mindsets and Math/Science Achievement*. New York: Carnegie Corporation of New York, Institute for Advanced Study, Commission on Mathematics and Science Education, New York, NY, USA.

[16] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. *SIGCSE Bull.* 40, 3 (June 2008), 328–328. https://doi.org/10.1145/1597849.1384371

[17] Susan E. Embretson and Steven P. Reise. 2013. *Item response theory*. Psychology Press.

[18] Anneli Eteläpelto. 1993. Metacognition and the Expertise of Computer Program Comprehension. *Scandinavian Journal of Educational Research* 37, 3 (1993), 243–254. https://doi.org/10.1080/0031383930370305

[19] Peter Ferguson. 2011. Student perceptions of quality feedback in teacher education. *Assessment & Evaluation in Higher Education* 36, 1 (2011), 51–62. https://doi.org/10.1080/02602930903197883

[20] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. 2007. Developing a Computer Science-specific Learning Taxonomy. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '07)*. ACM, New York, NY, USA, 152–170. https://doi.org/10.1145/1345443.1345438

[21] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[22] Charles Juwah, Debra Macfarlane-Dick, Bob Matthew, David Nicol, David Ross, and Brenda Smith. 2004. Enhancing student learning through effective formative feedback. *The Higher Education Academy* 140 (2004).

[23] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/2899415.2899422

[24] Oren Laadan, Jason Nieh, and Nicolas Viennot. 2010. Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. In *Proceedings of the 41st SIGCSE technical symposium on Computer science education*. ACM, 480–484.

[25] Joseph Lawrance, Seikyung Jung, and Charles Wiseman. 2013. Git on the cloud in the classroom. In *Proceedings of the 44th ACM technical symposium on Computer science education*. ACM, 639–644.

[26] Michael J. Lee and Andrew J. Ko. 2011. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/2016911.2016934

[27] Ying Liu, Eleni Stroulia, Kenny Wong, and Daniel German. 2004. CVS historical information to understand how students develop software. In *26th International Conference on Software Engineering*. International Workshop on Mining Software Repositories (MSR), 32–36.

[28] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[29] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[30] David J. Nicol and Debra Macfarlan-Dick. 2006. Formative assessment and self-regulated learning: a model and seven principles of good feedback practice. *Studies in Higher Education* 31, 2 (2006), 199–218. https://doi.org/10.1080/03075070600572090

[31] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 153–160.

[32] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/3230977.3230981

[33] Thomas W. Price, Neil C.C. Brown, Chris Piech, and Kelly Rivers. 2017. Sharing and Using Programming Log Data (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 729–729. https://doi.org/10.1145/3017680.3022366

[34] Sarah Quinton and Teresa Smallbone. 2010. Feeding forward: using feedback to promote student reflection and learning—a teaching model. *Innovations in Education and Teaching International* 47, 1 (2010), 125–135. https://doi.org/10.1080/14703290903525911

[35] Karen L. Reid and Gregory V. Wilson. 2005. Learning by Doing: Introducing Version Control as a Way to Manage Student Assignments. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*. ACM, 272–276.

[36] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (01 Mar 2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[37] Eric Roberts, John Lilly, and Bryan Rollins. 1995. Using Undergraduates as Teaching Assistants in Introductory Programming Courses: An Update on the Stanford Experience. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*. ACM, 48–52.

[38] Kimberly Tanner. 2012. Promoting Student Metacognition. *CBE Life Sciences Education* 11, 2 (2012), 113–120.

[39] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. 2008. Bloom's Taxonomy for CS Assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78 (ACE '08)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 155–161. http://dl.acm.org/citation.cfm?id=1379249.1379265

[40] Peter Vamplew and Julian Dermoudy. 2005. An Anti-Plagiarism Editor for Software Development Courses. In *Proceedings of the 7th Australasian Conference on Computing Education*. 83–90.

[41] Fabienne M. Van der Kleij, Remco C. W. Feskens, and Theo J. H. M. Eggen. 2015. Effects of Feedback in a Computer-Based Learning Environment on Students' Learning Outcomes: A Meta-Analysis. *Review of Educational Research* 85, 4 (2015), 475–511. https://doi.org/10.3102/0034654314564881

[42] Lisa Yan, Nick McKeown, and Chris Piech. 2019. The PyramidSnapshot Challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3287324.3287386

[43] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 110–115. https://doi.org/10.1145/3159450.3159490