Analyzing Student Learning Trajectories in an Introductory Programming MOOC

by

Ayesha R. Bajwa

S.B. Massachusetts Institute of Technology (2018)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author . . . . . .
Department of
Electrical Engineering and Computer Science
May 24, 2019

Certified by . . . . . .
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Certified by . . . . . .
Erik Hemberg
Research Scientist
Thesis Supervisor

Accepted by . . . . . .
Katrina LaCurts
Chairman, Master of Engineering Thesis Committee

Analyzing Student Learning Trajectories in an Introductory Programming MOOC

by

Ayesha R. Bajwa

Submitted to the Department of
Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Understanding student learning and behavior in Massive Open Online Courses (MOOCs) can help us make online learning more beneficial for students. We investigate student learning trajectories on the individual problem level in an MITx MOOC teaching introductory programming in Python, considering simple features of the student and problem as well as more complex keyword occurrence trajectory features associated with student code submissions. Since code is so problem-specific, we develop gold standard solutions for comparison. Anecdotal observations on individual student trajectories reveal distinct behaviors which may correlate with prior experience level. We build models to correlate these trajectories with student characteristics and behaviors of interest, specifically prior experience level and video engagement. Generative modeling allows us to probe the space of submitted solutions and trajectories and explore these correlations.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Acknowledgments

Firstly, thank you to my patient and willing thesis supervisors, Erik Hemberg and Una-May O'Reilly, for their mentorship, funding, feedback, and their faith in me as a student and researcher. I'm very grateful to ALFA for creating an explicitly positive and healthy lab environment for its graduate students.

I'm thankful for the support and collaboration of many others: Ana Bell for her expert insight on teaching 6.00.1x and her patience in working with us, John Ding for some of his processing code, Sagar Biswas for his gold standard solutions, Shashank Srikant for discussing and formulating generative models with me, and high school interns Dhruv Sumathi and Sasha Shashkov for enthusiastically starting to tackle some next steps posed by this thesis. Nicole Hoffman has my gratitude for helping to keep funding logistics stress-free and for the excellent group lunches. I'm also grateful to Srini Devadas, my academic advisor at MIT, for numerous useful discussions on learning and research.

Thank you to ALFA and to the MIT EECS Department for the funding support. Thank you to the EECS Undergraduate Office for assisting with the logistics of the MEng program. Thank you to all the mentors, friends, collaborators, and critics who have made MIT a better place to learn and grow.

Finally, thanks to my favorite people in California: Anupa Bajwa, Eshan Bajwa, Raminder Bajwa, and Vipul Vachharajani for the unwavering encouragement and support from 3,070 miles (4,941 km) away. I'm glad you won't be quite so far away in the near future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent popularity of Massive Open Online Courses (MOOCs) is making it easier to study student learning trajectories – or student learning over the course of solving a single problem or multiple problems – in a fine-grained fashion. In particular, MOOCs that teach students how to program are ubiquitous, making the processes by which students learn to code important considerations for instructional designers. The user interaction data recorded by MOOC platforms such as edX can be represented as features and fed to statistical and machine learning models to find links between student activity and learning outcomes or student characteristics. Previous work has established the importance of "doing" in online courses, as opposed to simply reading and watching [13, 14], as critical to success. One recent paper [19] (on the same course we study) confirms the importance of doing practice exercises over watching video lectures in a dataset from an MITx MOOC hosted on edX: 6.00.1x Introduction to Computer Science and Programming Using Python.

By leveraging the granular, temporal data collected in student activity logs, we can improve our understanding of what kinds of activity or particular trajectories best support learning, and adjust instructional design accordingly. A key focus is the submission history of a particular student for a particular coding problem. By inspecting a student's progression of code submissions, or even just the final submission attempt, we can look for correlations to attributes like student prior experience

level, video engagement, grades, or course completion. Furthermore, we can use this progression to explore *how* students learn, identifying distinct learning behaviors that could generalize to programming and computational thinking MOOCs.

1.1 Related Work

MOOCs in the realm of computational thinking [31] and programming are especially popular, and MOOC student populations tend to be more diverse than traditional classrooms [4, 17]. Previous work indicates that student engagement is key to success, as students who complete course activities are more likely to become certified than those who do not [30]. Additionally, research has shown that measures to personify programming tool feedback can improve the learning experience [16]. The importance of engagement is especially clear in previous work on the "doer effect" [13, 14], showing that the "doing" of practice exercises is more important than reading or watching videos as a predictor of successful outcomes. The presence of this "doer effect" has been validated in the 6.00.1x dataset across student prior experience levels and course topics [19]. We use the same course context to further explore student learning trajectories by inspecting more granular information, such as the history of student submissions to a particular coding problem.

Traditionally, student engagement over time has been tracked at the macro level, over the course of a MOOC. Classification methods have identified distinct clusters of student engagement trajectories within a MOOC, finding that a significant population showed disengagement; these students were engaged at the beginning of the course but had dropped off by the end [12]. Another finding employed clustering to determine student learning behaviors in MOOCs, finding categories of active learners, passive learners, and bystanders based on engagement trajectories over the length of the course [26]. Other work has focused on extracting active learning measures from MOOCs [5]. These findings confirm that students, who enroll in MOOCs for a variety of reasons, exhibit a variety of distinct learning behaviors. Some studies take low-level data such as user events and interactions and turn them data into high-level

information parameters such as effective time spent, total time, use of optional items, and effective correct progress [21]. *How* students arrive at problem solutions can be modeled using the evolution of a student's compiled code for a particular problem, which has shown higher correlation with course outcome than the grade received for that problem does [23].

Yet other work has focused on identifying misconceptions and unhelpful persistence so that the appropriate interventions may be applied. One study used a data-driven approach based on student attempts to identify domain misconceptions from student-tutor interaction logs in an Intelligent Tutoring System [7]. In another study, researchers found that by using word embeddings for unstructured, textual submissions to open-response questions, student misconceptions could be detected with Markov chain Monte Carlo inference [20]. Students who experience unsuccessful persistence may be discouraged and inhibited from learning effectively [24]. These are termed "wheel-spinning" behaviors, since students spend too much time persisting in a way that fails to help them achieve mastery of the material [2]. One study [8] focused on low-level log data for student persistence, in contrast to the usual macro course-level persistence, and identified different learning patterns using clustering. However, learning outcomes did not appear to vary based on these patterns. They also found that wheel-spinning often coexists with productive persistence. A later study used decision trees to distinguish productive from unproductive persistence [10]. These kinds of prior studies motivate us to study learning trajectories and their correlation to particular student behaviors such as video engagement.

A survey paper on automated assessment tools for programming questions between 2006 and 2010 reports that the integration of automated checkers into learning management systems, as well as better security against hackers, is likely to become more prevalent [9]. The integration of automated checkers into MOOC platforms allows for more detailed student activity logging. One study identifies a common cheating behavior detectable on MOOC platforms: the strategy is called CAMEO, or Copying Answers using Multiple Existences Online [22], and refers to a user creating multiple accounts and then using some as "harvester" accounts while copying revealed,

correct answers into a "master" account that may be used to earn a certificate. The authors note that the MOOC platform and data logs enable both the behavior itself and the detection of the behavior. A follow-up study in a science MOOC aiming to characterize the prevalence of CAMEO behavior finds that between 3% and 10% of certificate earners used harvester accounts for their correct answers, with the main motivation for cheating being the ability to earn a certificate [1]. A different case study on cheating behaviors reveals four qualitative categories: overfitting answers to test cases, redirection to reference solutions, problem evasion, and grading statement injections [15].

Clever automatic grading reduces the need for a human instructor to grade programs. Randomization is one approach that can prevent students from hard-coding solutions to specific test cases without solving the general problem. Such tools are especially useful for online courses, and feature representations of code can be used to understand student improvement in coding problems. Traditionally, measures such as semantic similarity [29] or control flow graph similarity [27] between submitted programs and a correct model program have been trialed for automatic grading. Another study used features derived from abstract representations of programs to perform automatic grading, finding that the percentage of test cases passed was usually a better indicator than simple features such as keyword and token counts, but could be improved with the addition of more complex features such as abstract representations that take order and structure into account [25]. To study problem-specific learning trajectories based on code submission history, we begin with a simple feature representation based on Python keyword occurrence count. We begin with these simple features because the goal is to find the simplest representation that allows us to probe the research questions of interest.

The most directly relevant prior studies on modeling code trajectories in introductory programming courses use representations of student code submissions to train state machines [23] and recurrent neural networks [28] to model knowledge representation and learning. If we can synthesize meaningful features and models that reflect student learning based on code submissions in a programming MOOC, we can

move towards automated conceptual progress tracking and feedback at scale. Our approach to modeling first considers the space of code submissions and trajectories given a particular feature representation, utilizing previous work on variational autoencoders [11, 3] to create a generative model capable of parameterizing a latent space learned from student code submissions or trajectories.

1.2 Prior Work on Student Behaviors

We describe some of our earlier results from [19], studying the doer effect on the 6.00.1x dataset, that help motivate the research questions of this thesis. Using mixed effects linear regressions on temporally-binned features associated with *watching* (denoted by video engagement) or *doing* (denoted by finger exercise engagement, or fex) in 6.00.1x, we observe that, in general, practice is more highly correlated with success than watching is correlated with success. This general trend is depicted in Figure 1-1 as a direct comparison of the regression coefficients of watching-related and doing-related student activity features from 6.00.1x data.

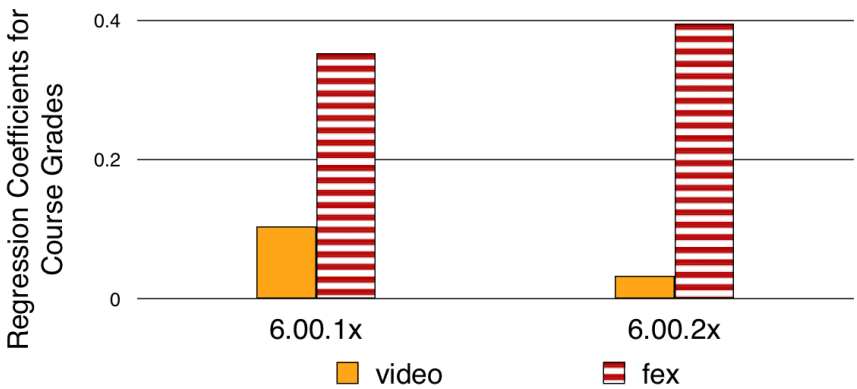

Figure 1-1: **Student behavior: the doer effect.** We see evidence of the doer effect in separate two-feature mixed effects linear regressions for 6.00.1x and 6.00.2x (the following MITx course), since the magnitude of the coefficients associated with doing have higher magnitudes. Only coefficients significant at the p=0.05 level are included.

The doer effect also manifests in a more fine-grained breakdown across student prior experience levels, given in Figure 1-2. We observe the doer effect for all experience levels in the 6.00.1x regression across experience for problem set grades,

since the coefficient of largest magnitude belongs to the "within-unit" fex variable for every experience level. The "within-unit" fex variable ranges from 1.5 to 10 times the magnitude of the other variables associated with a given experience category for 6.00.1x. The direct comparison of video and fex coefficients suggests that doing is clearly more important than watching for students of all experience levels, while the relative importance of watching varies between experience levels.

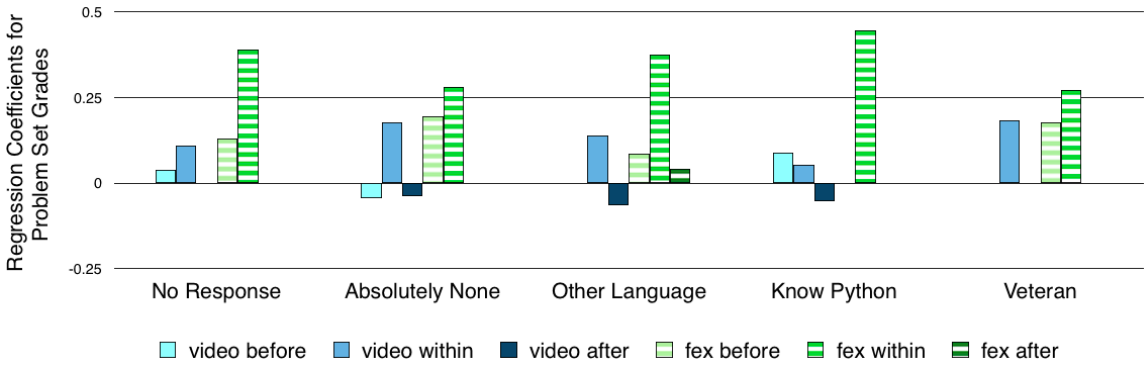

Figure 1-2: **6.00.1x by experience: problem set grades.** Regression coefficients for problem set grades as a function of student activity for 6.00.1x, grouped by student prior experience. The doer effect is apparent, with the "within-unit" finger exercises coefficient having the highest magnitude in every experience category. Only coefficients significant at the p=0.05 level are included.

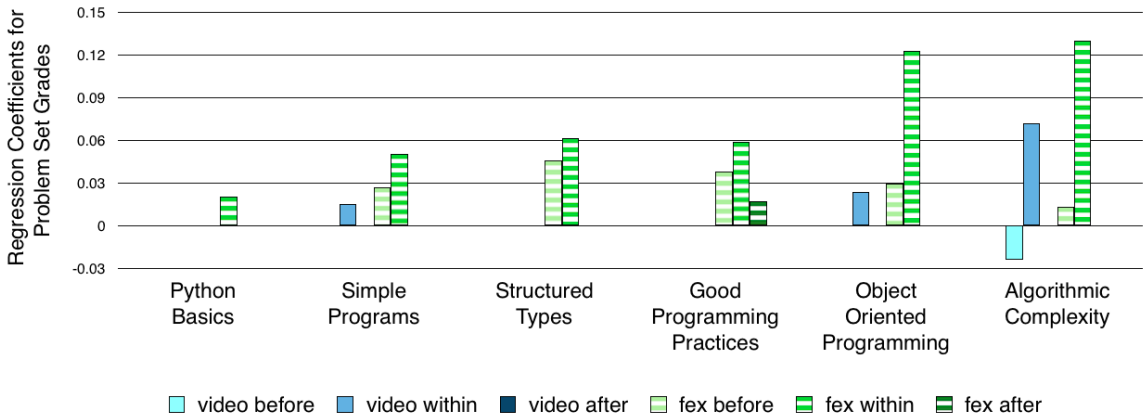

Figure 1-3: **6.00.1x by topic: problem set grades.** Regression coefficients for problem set grades as a function of student activity for 6.00.1x, grouped by unit topic. The "within-unit" finger exercises coefficient has the highest magnitude in every unit, ranging from 1.3 to 10 times the magnitude of the other variables associated with a given unit. Only coefficients significant at the p=0.05 level are included.

Furthermore, when considering a fine-grained breakdown of the doer effect across different topics (units) within 6.00.1x, given in Figure 1-3, we find that student success in particular topics is more correlated with doing than success in other topics is correlated with doing. The "within-unit" fex variable always has largest magnitude, ranging from 1.3 to 10 times the magnitude of the other variables associated with a given unit. An interesting observation is that more conceptual topics within computational thinking, such as algorithmic complexity, appears to oppose the general trend in which video engagement is relatively unimportant. Though the doer effect is still present for the algorithmic complexity topic, since the largest magnitude coefficient belongs to finger exercises, the video features are relatively meaningful as well.

The apparent importance of doing causes us to ask: what kind of doing is important? What behaviors can we learn representations for in the 6.00.1x dataset that shed light on the ways in which students behave? A natural place to observe student behavior is in the graded code submissions in each problem set.

1.3 Research Questions

Our research questions focus on whether considering low-level student activity data in the form of code submission attempts to problem set questions, which we call trajectories, can allow us to predict student outcomes or discover separation by student characteristics (e.g. prior experience level) or behaviors (e.g. video engagement).

Research Question 1. First, we investigate whether simple features such as submission length, number of attempts, and the manually annotated difficulty level of coding problems within graded problem sets are sufficiently correlated to separate the students by grade outcomes. We also consider a breakdown of the simple features over prior experience of students in the course to identify any trends characteristic of groups with a particular experience level.

Research Question 2. Second, we aim to learn student characteristics from more complex features, such as abstract and sequence-oriented representations of code attempts submitted by students. Specifically, are there activity patterns characteristic

of successful or experienced students? How do students who watched a relevant video compare to students who did not, in terms of their submission content and behavior? Using trajectory features, are there groupings of student behaviors that could be found with unsupervised approaches like clustering or generative models? We consider here whether the prior experience of students or video engagement behavior may affect such groupings, and whether these correlations can be revealed through student submission activity.

Research Question 3. Finally, using a combination of simple and complex features, what can we deduce about *how* students learn to code? Are there multiple prototypical learning behaviors that can be inferred from student activity? The broad goal of research like ours is to identify the connections between particular student behaviors or trajectories and learning effectiveness in order to improve course design, develop automated interventions and personalization, and to make online learning more useful for students.

Contributions. In addition to describing the official 6.00.1x edX datasets, we obtain and describe a new dataset of code submission histories that allows us to explore student behavior over time. We visualize the data and make some anecdotal observations that allow us to answer some basic research questions and identify others. We then use a generative model to explore the space of individual submissions and per-problem trajectories.

Thesis Structure. In Chapter 1 we discuss motivation, prior work, and research questions. The 6.00.1x course context and the official edX dataset are described in Chapter 2. Chapter 3 additionally focuses on code submissions and trajectories, describing anecdotal observations in depth and motivating and explaining our feature representations of both submissions and trajectories. In Chapter 4, we give an overview of the generative modeling approach, results, and challenges. Chapter 5 discusses our findings, summarizes potential limitations of our approach, and identifies future directions for this work.

Chapter 2

Dataset Characterization

We consider an aggregated dataset comprised of multiple offerings of 6.00.1x Introduction to Computer Science and Programming Using Python. The data comes from two course terms, 2016 Term 2 and 2017 Term 1. Following previous work [19], we consider the subset of students who complete and earn certification in 6.00.1x. Of the 251,953 students who enrolled in 6.00.1x in these two terms, a total of 3,485 received certification. We consider the behavior of only these certified students since dropout is high and we wish to observe student behavior through the duration of the course.

6.00.1x is divided into six distinct units. Video lectures are followed by optional practice exercises designed for students to practice the material introduced in lectures. The structure and format of graded activities in 6.00.1x is shown in Table 2.1. Only Units 1 through 5 include coding problems on their associated problem sets.

2.1 Distribution of Experience Level

The prior experience survey employed in 6.00.1x gives us a breakdown of entering student experience level in the 2016 Term 2 and 2017 Term 1 data. The survey asks students to self-report any prior familiarity with programming at the beginning of the course: students choose from the categories of *Absolutely None*, *Other Language*, *Know Python*, *Veteran*, or they do not respond.

Table 2.1: 6.00.1x Graded Activities

Assignment	Type
Unit 1: Python Basics	Problem Set
Unit 2: Simple Programs	Problem Set
Midterm Exam	Exam
Unit 3: Structured Types	Problem Set
Unit 4: Good Programming Practices	Problem Set
Unit 5: Object Oriented Programming	Problem Set
Unit 6: Algorithmic Complexity	Problem Set
Final Exam	Exam

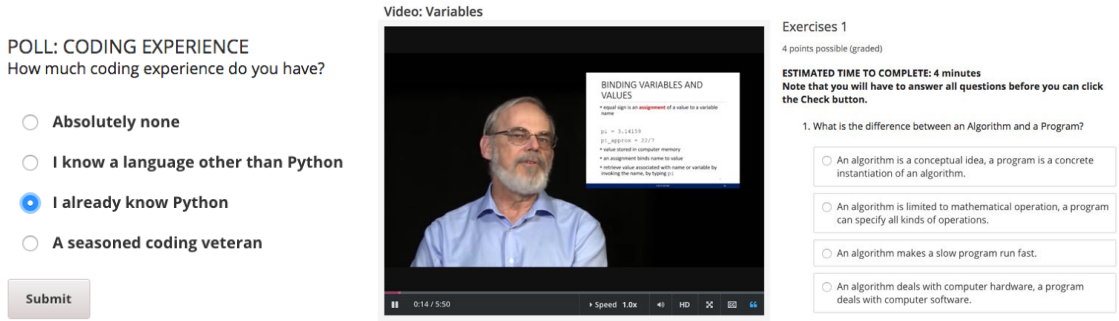

Figure 2-1: **6.00.1x learning affordances.** Students self-identify prior coding experience in an optional pre-course survey (*left*), and the learning affordances of 6.00.1x Introduction to Computer Science and Programming Using Python include video lectures (*center*) and optional exercises following the relevant videos (*right*).

A breakdown of survey results in Figure 2-2 may indicate that prior exposure to coding is correlated to certification in 6.00.1x. Only 853 certified students, or about a quarter, report Absolutely None, and the largest experience category among certificate earners is the Other Language category with 1596 students. The sample of Veterans is relatively small at 75, there are 578 students who indicate Know Python, and 410 students do not respond to the survey.

These proportions by experience level are different from the proportions by experience level for the general population of 6.00.1x enrollees, rather than just certified students [18]. This difference may be an indication that the 6.00.1x material and format favor those with coding experience, or it may simply a consequence of self-selection in the population that enrolls in 6.00.1x with the option for certification.

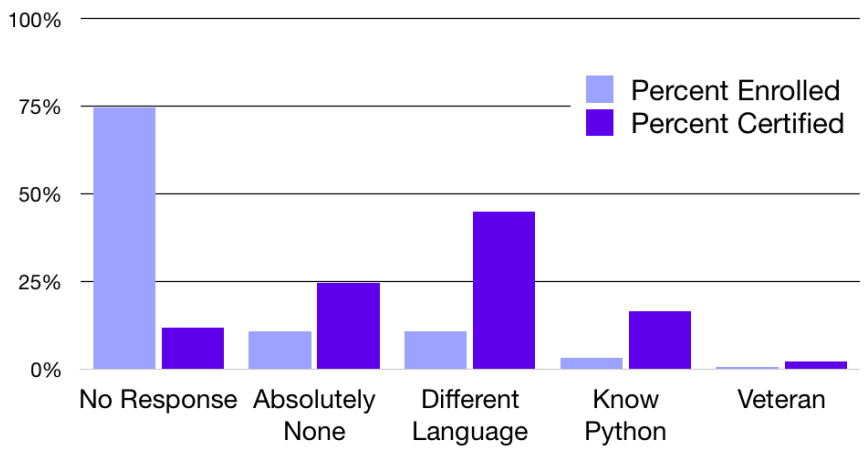

Figure 2-2: **6.00.1x prior coding experience survey.** The 6.00.1x prior experience survey responses show that most students who receive a certificate have some prior coding experience, while a quarter of certified students reported no prior experience. For scale, 251,953 students enrolled in 6.00.1x and 3,485 received certification.

2.2 Selected Coding Problems and Difficulty

In order to study behavior at a fine-grained level, we focus on the 21 coding problems in the graded problem sets of 6.00.1x.

The full list of problems and difficulties are shown in Tables 2.2 and 2.3. Note that while there is a Unit 6: Algorithmic Complexity in 6.00.1x, the associated problem set does not contain any coding problems, so we do not include it in the analysis. In addition to the problem and student features extracted from the edX platform, we consider manually assigned features of problem difficulty along various axes, as assigned by a course instructor (Table 2.2) and by learning science experts (Table 2.3).

The difficulties given in Table 2.2 are assigned by Ana Bell, one of the 6.00.1x instructors. The difficulties range from 1 (easy) to 6 (hardest) [1]. The difficulty is assigned based on the content of the week in question, the implicit assumption being that students have more or less paid attention to previous material; this assumption is likely most valid for those students with no prior coding experience. For example, after a student finished week 1, the difficulty is assigned based on what the student

[1] In 'Pset 4-5 Playing a Hand' students are given pseudocode comments, and they just need to fill it in. It would be a 6 if they were not given that.

Table 2.2: 6.00.1x Coding Problem Instructor Difficulty Ratings

Problem	Task Description	Instructor Rating
1-1	Count Vowels	2
1-2	Count 'bob'	5
1-3	Longest Alphabetical Substring	6
2-1	Paying Debt off in a Year	1
2-2	Paying Debt off in a Year	3
2-3	Bisection Search Makes the Program Faster	6
3-1	Is the Word Guessed	1
3-2	Printing Out the User's Guess	2
3-3	Printing Out all Available Letters	2
3-4	The Game	4
4-1	Word Scores	1
4-2	Dealing with Hands	3
4-3	Valid Words	4
4-4	Hand Length	1
4-5	Playing a Hand	3
4-6	Playing a Game	4
4-7	You and your Computer	6
5-1	Build the Shift Dictionary and Apply Shift	4
5-2	Plaintext Message	3
5-3	Ciphertext Message	4
5-4	Decrypt a Story	2

should have learned in week 1. It would be possible to also assign more conceptual difficulty scores based on the entire course. In that case, questions that have a more algorithmic aspect (such as 2-3) would rate at a higher difficulty than ones that simply have many edge cases to handle (such as 4-7).

Unlike the simple instructor difficulty ratings along a single axis, the learning expert difficulty ratings in Table 2.3 are based on three separate difficulty metrics. The first metric is based on the difficulty of the data structure used within the program and returned from the target function. The second metric is based on the difficulty of deducing the algorithm used to solve the problem. The third metric is based on the difficulty of implementing the algorithm. The scale is 1 (easiest) to 3 (hardest).

Table 2.3: 6.00.1x Coding Problem Learning Expert Difficulty Ratings

Problem	Task Description	Data Structure	Deduce Algorithm	Implement Algorithm
1-1	Count Vowels	1	2	1
1-2	Count 'bob'	1	2	2
1-3	Longest Alphabetical Substring	2	3	3
2-1	Paying Debt off in a Year	1	1	2
2-2	Paying Debt off in a Year	1	2	1
2-3	Bisection Search Makes the Program Faster	2	3	3
3-1	Is the Word Guessed	1	1	1
3-2	Printing Out the User's Guess	1	2	2
3-3	Printing Out all Available Letters	1	1	2
3-4	The Game	2	2	3
4-1	Word Scores	1	2	1
4-2	Dealing with Hands	2	3	2
4-3	Valid Words	2	3	3
4-4	Hand Length	2	1	1
4-5	Playing a Hand	2	2	3
4-6	Playing a Game	2	3	2
4-7	You and your Computer	2	3	3
5-1	Build the Shift Dictionary and Apply Shift	2	3	3
5-2	Plaintext Message	2	3	3
5-3	Ciphertext Message	2	2	3
5-3	Decrypt a Story	0	0	0

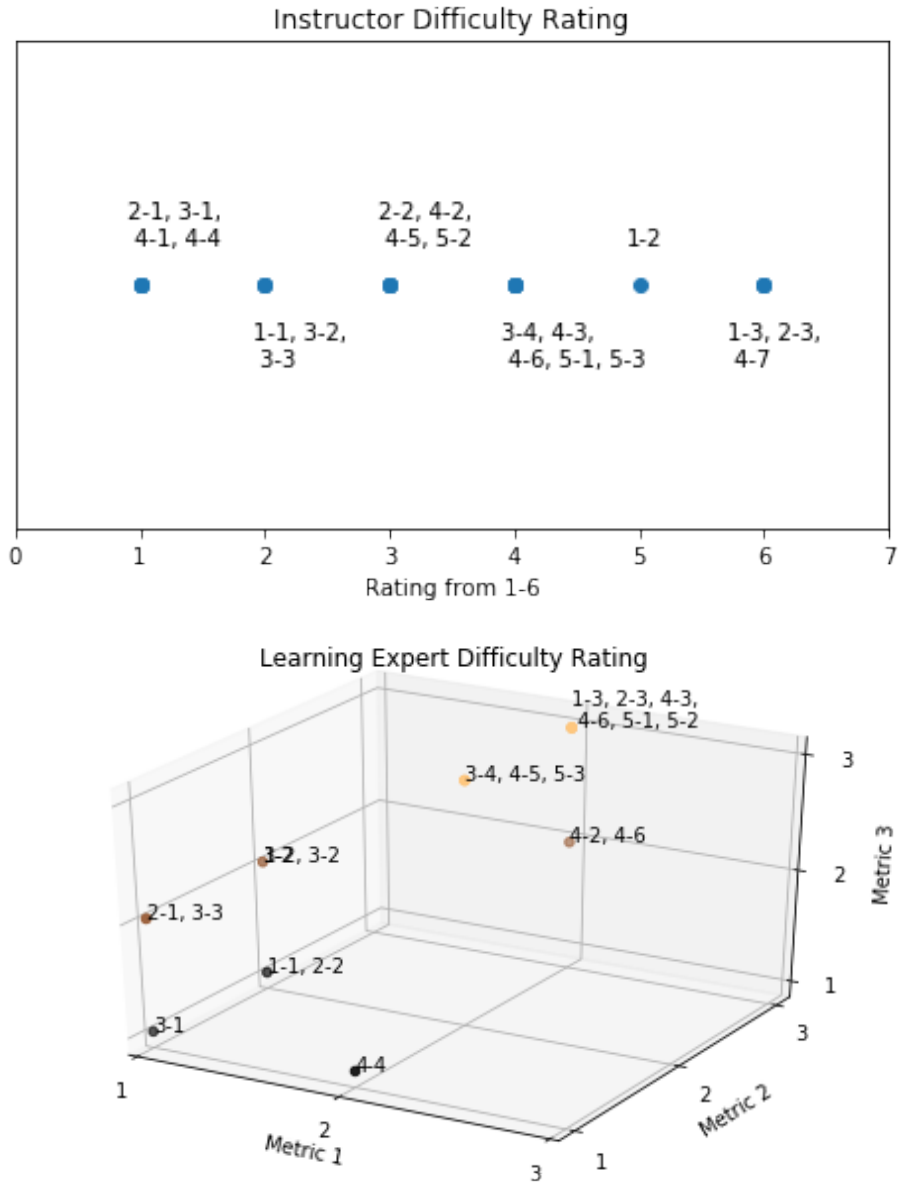

Figure 2-3: **Visualization of manual difficulty features.** We obtained manually assigned difficulty ratings for each of the 21 6.00.1x coding problems from both course instructors (*top*) as well as learning science experts (*bottom*). While course instructors rated overall problem difficulty on a single axis, our learning science collaborators broke down difficulty under three considerations: 1) the data structure, 2) deducing the algorithm, and 3) implementing the algorithm.

2.3 Number of Attempts

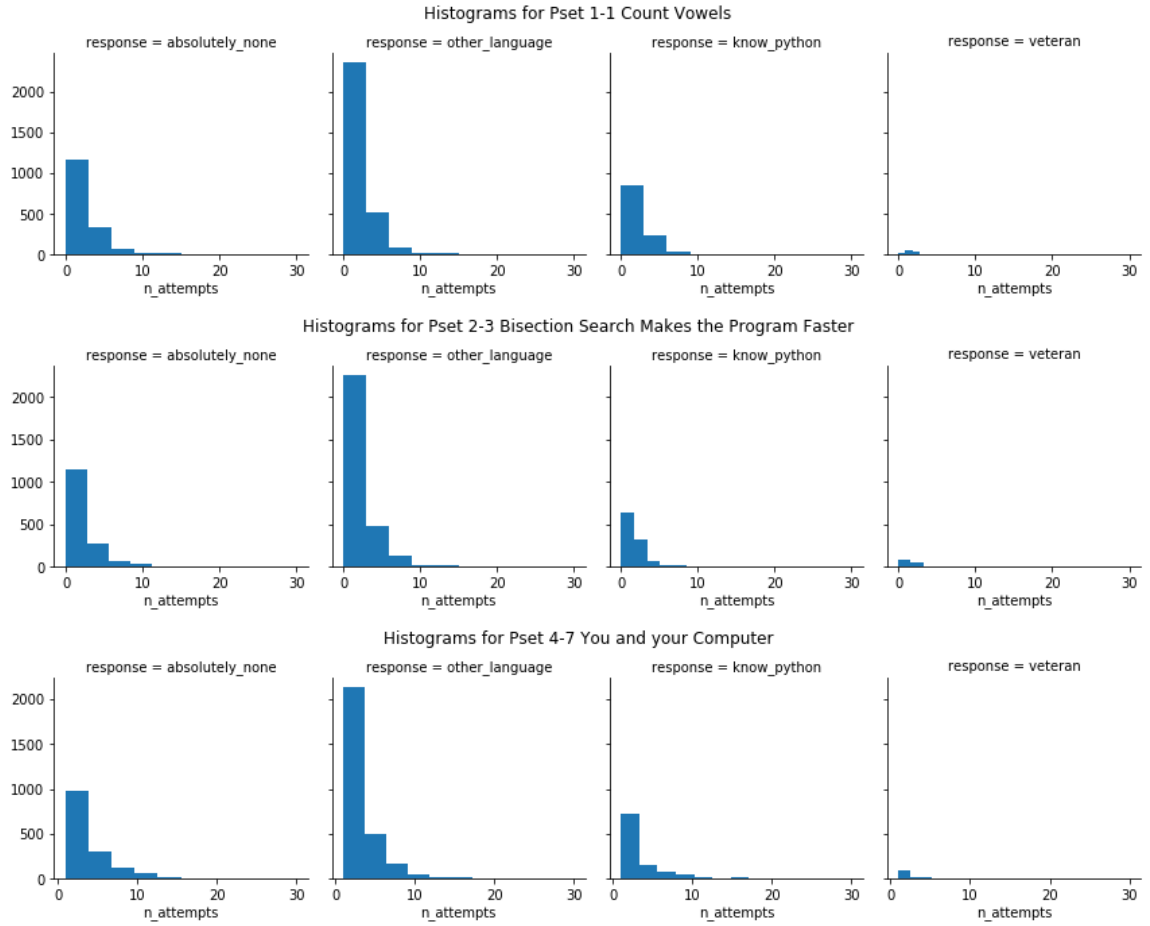

Figure 2-4: **Histogram of number of attempts.** We show histograms of the numbers of attempts students made to selected problems. Students are capped at 30 submissions. Early problems like 1-1 tend to take fewer attempts on average, while long problems like 4-7 take more. The average number of attempts for each of the problems shown are: 2.2 attempts for 1-1, 2.1 attempts for 2-3, and 4.6 attempts for 4-7. Accounting for sample size, the distributions for a given problem are similar across experience level.

We first characterize problems by visualizing the distribution of the number of attempts that certified students, divided by prior experience, took to solve each problem. This relative measure indicates which problems are more difficult; the distribution peaks at a higher number of attempts for problems later in the course. We see that, accounting for sample size, the distributions are similar across experience level.

We focus on three problems: 1-1, 2-3, and 4-7. Observe that in both difficulty ranking schemes, 1-1 is ranked as a relatively easy problem, while 2-3 and 4-7 are

hard problems. We see that difficulty rankings somewhat correlate with the average number of attempts, e.g. 4-7 is hard and has an average of 4.6 attempts per student across our population of certified students. 1-1 has an average of 2.2 attempts. 2-3 is interesting because it is an algorithmic problem (making it difficult by instructor and learning science expert characterizations) yet has a relatively low average number of attempts at 2.1 attempts per student.

2.4 Student and Problem Features

In addition to difficulty ratings, we consider whether simple features are sufficient to separate the students by grade outcomes. These features are shown in Table 2.4.

Table 2.4: List of Simple Features

Feature	Note
number of attempts	per student, per problem
submission length in lines	per student, per problem
submission length in characters	per student, per problem
average number of attempts	per problem
average number of solvers	per problem

Number of attempts is the feature that can give us an idea of how quickly a student solved a problem. However, student behavior varies in how often students resubmit after editing their submissions; some students may be willing to use up more of their 30 attempts by resubmitting after small changes, while others may resubmit only after meticulous edits and local testing. Therefore, we expect this to be a noisy feature with limited influence on grade outcomes. We also hypothesize that the more difficult problems are likely to show more scatter across length and lines features.

Submission length by number of lines uses the relatively common feature of lines of code. We expect this to be a less noisy feature than number of characters, since simply using longer variable names in the same code would imply more characters. However,

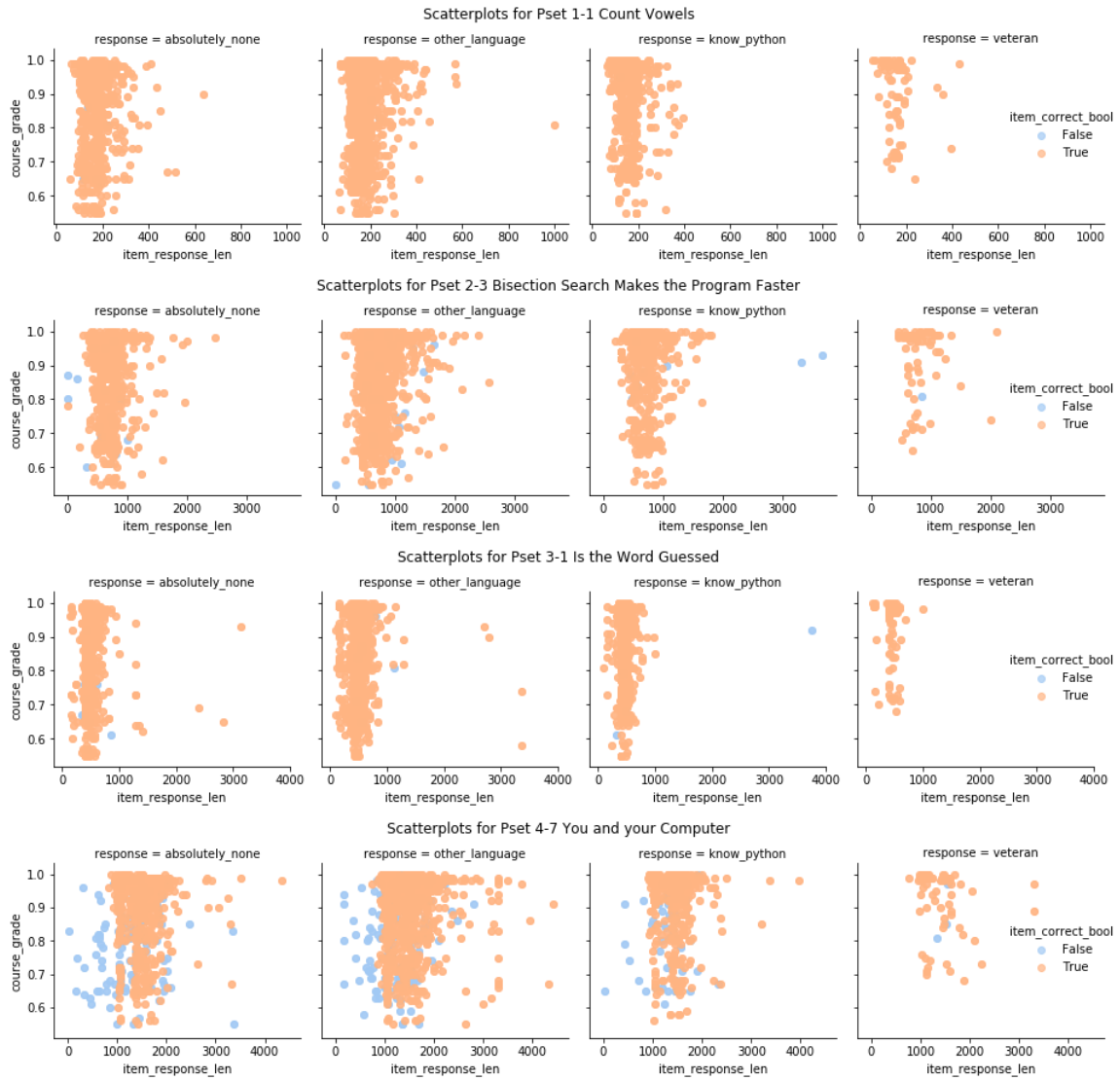

Figure 2-5: **Final submission length feature.** Looking at how the response length feature (*item response length*) of student final submissions correlates with course grade, the means do not vary significantly across experience level. We observe more scatter for longer and more difficult problems like 4-7. The difficulty of 2-3 is largely algorithmic, and we do not see the length vary as much. The easier problems, 1-1 and 3-1, have less scatter. We also observe that more students answer 4-7 incorrectly, especially those with shorter solutions.

we hypothesize that number of lines is still less predictive than more abstracted features, such as Python keyword features or features derived from an Abstract Syntax Tree (AST) may be. The number of lines is a particularly interesting feature because of its prevalence: the "ballpark" number of lines of code written is a common question in industry hiring, though it may not be distinguishing or predictive.

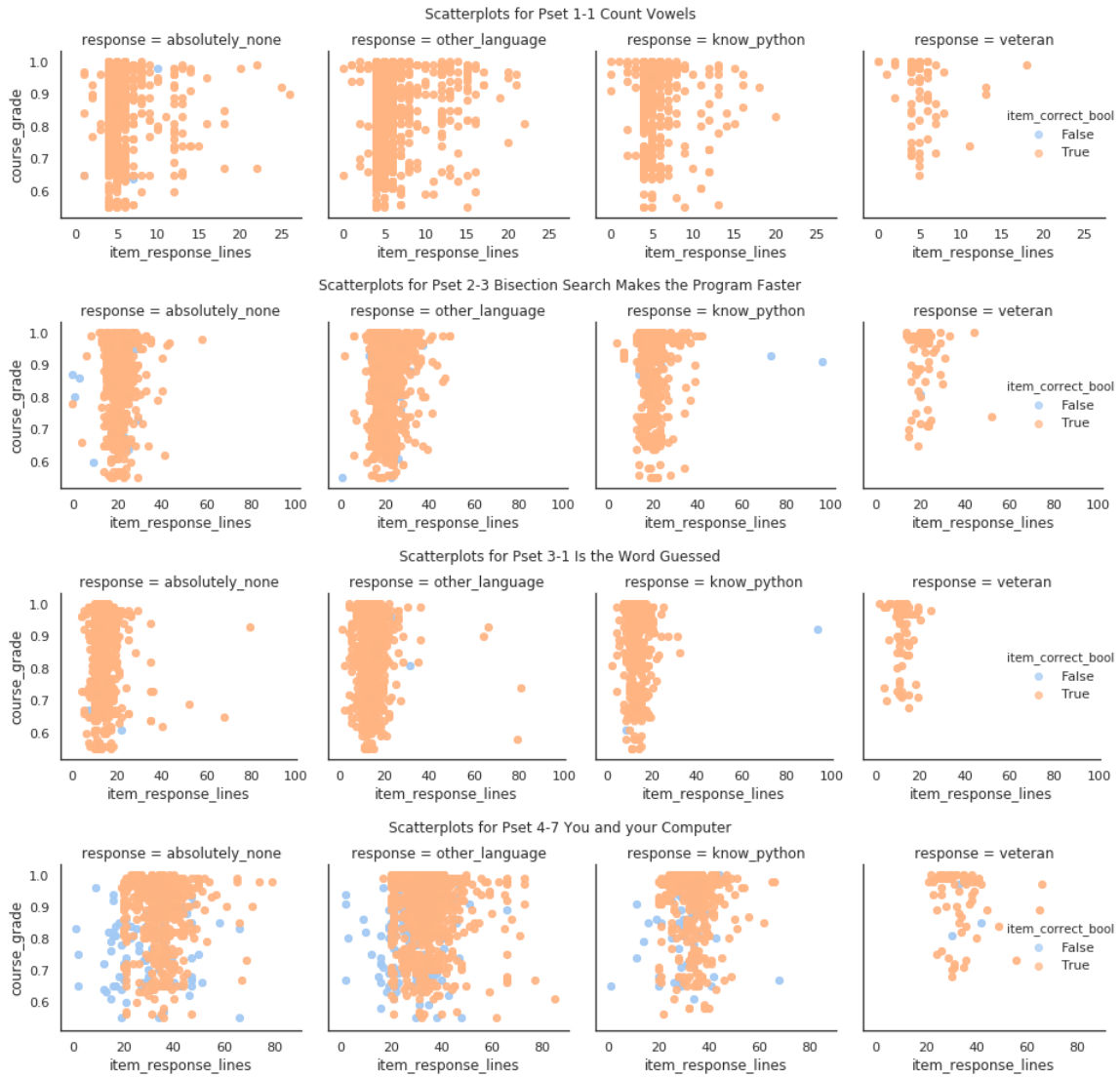

Figure 2-6: **Final submission number of lines feature.** Looking at how the number of response lines feature (*item response lines*, which we expect to mirror the response length feature) of student final submissions correlates with course grade, the means do not vary significantly across experience level, as before. We observe a surprising amount of scatter for 1-1, which we hypothesize is due to a large number of suboptimal solutions. As before, we observe that more students answer 4-7 incorrectly, especially those with shorter solutions.

Though the means are more or less the same across experience level, there appears to be significant variation in Figures 2-5 and 2-6, though it may be due to sample size effects rather than differences due to prior experience. The small pool of coding veterans shows less scatter than the other categories and is more distinguishable from them as a whole (though many of the individual data points still overlap).

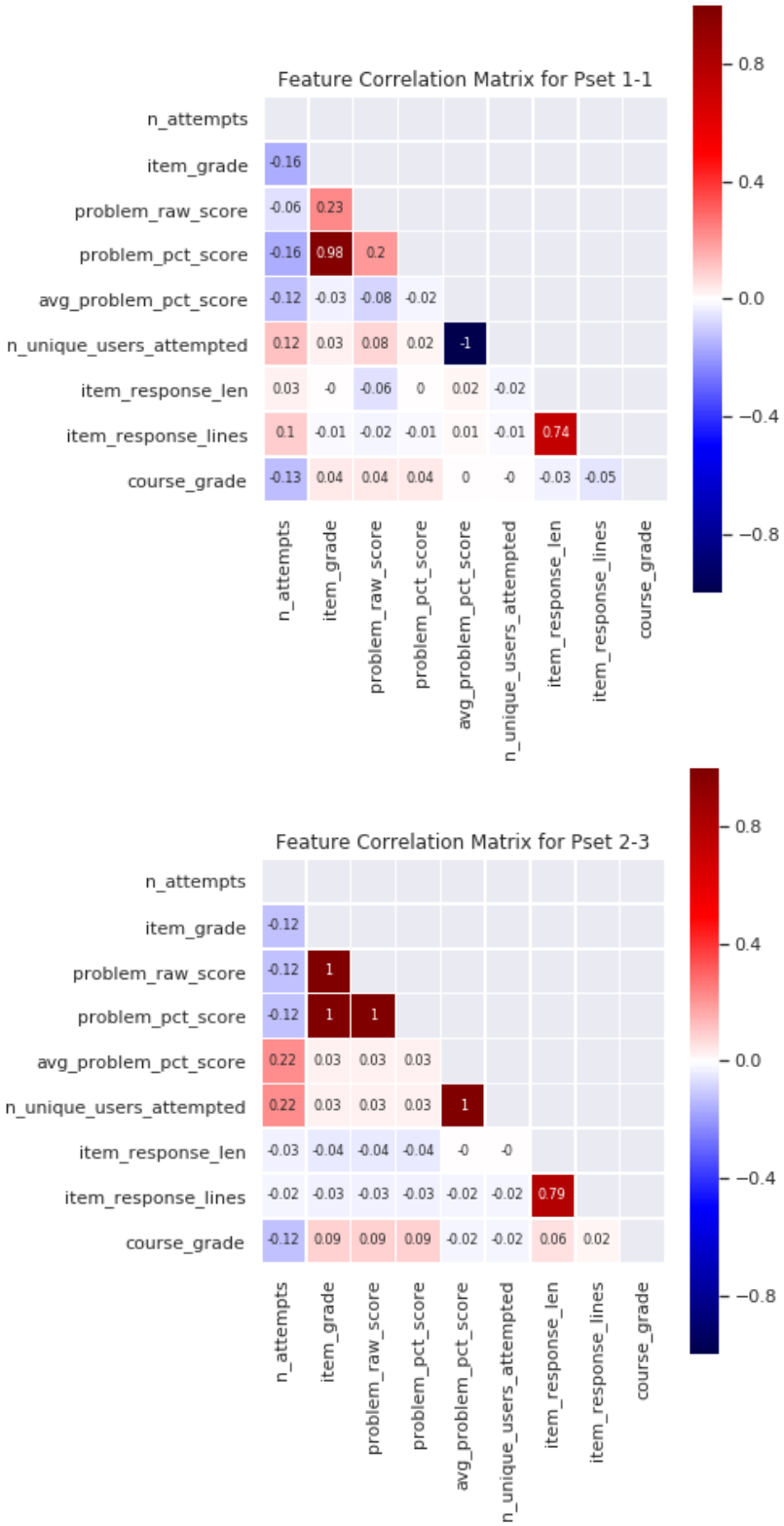

Figure 2-7: **Simple feature correlations.** We create heatmaps of simple feature correlation (Pearson correlation) for all the problems, with 1-1 and 2-3 shown here. None of the correlations are particularly strong unless they are directly dependent, such as *problem raw score* or *problem percent score* and *item grade* or *response lines* and *response length*.

In contrast to number of attempts, length, and lines, which are per student and per problem, the average number of attempts and average number of solvers are features particular to each problem only. In this sense, they are similar to the manual difficulty ratings for each problem.

We obtain scatterplots similar to Figures 2-5 and 2-6 for the other simple features, none of which show distinct behaviors between experience levels. We then aim to get a sense of whether any of these simple features may be used to predict grade outcomes as a baseline. While these visualizations of simple feature categories for all students and across experience levels seems to agree with the intuition provided by course instructors and learning experts, these features alone appear insufficient to distinguish or predict grade outcomes. Looking at feature correlations in Figure 2-7, we find that none of these features, as is, correlates strongly enough with course grade outcome for it to be a good predictor of student performance. We omit directly dependent features, such as *problem raw score* and *problem percent score* or *response lines* and *response length*, when looking for useful, potentially predictive correlations. This finding holds across levels of prior experience. Also note that *difficulty ana* is the instructor difficulty rating, and the learning expert difficulty ratings are *difficulty hk m1* for the data structure difficulty, *difficulty hk m2* for the difficulty of deducing the algorithm, and *difficulty hk m3* for the difficulty of implementing the algorithm. It is also interesting to note that for 1-1, the *average problem percent score* is negatively correlated with the *number of unique users attempted*, while the two features are positively correlated for 2-3.

In this chapter, we explored the official 6.00.1x edX dataset and found that the simple features we consider are generally insufficient to predict grades or separate the students by experience level. In the next chapter, we consider more complex trajectory features and what they can tell us about student characteristics and behavior.

Chapter 3

Trajectory Features

In this section, we no longer consider the prediction of student grade outcomes. Rather, we ask the more general question of what we can learn about students given their history of submitted code on problem sets. We refer to these code submission histories as *trajectories*, focusing on particular problem trajectories for different students and in aggregate, though we briefly consider student trajectories across multiple problems. Examples of what we could infer about student characteristics include identifying their prior experience level or whether a student watched the video most relevant to the problem.

In classrooms, instructors teaching students how to code have the ability to monitor progress and provide feedback through regular interaction. There is generally no analogous tracing of learning progression in programming MOOCs, hindering the ability of MOOC platforms to provide automated feedback at scale. Furthermore, programming MOOCs contain open-ended coding problems, which have more complex feature representations than other types of problems (e.g. multiple choice questions). Furthermore, students arriving at similar answers have varying code submission histories that represent distinct conceptual struggles. Moving beyond the simple features considered in earlier chapters, we explore features for every certified student's history of code submissions (rather than just the final submission) to specific coding problems in 6.00.1x and measure similarity to gold standard solutions that we develop.

Table 3.1: 6.00.1x Student Submission Attempts by Experience

Experience (Count)	Total Attempts	Attempts to Count Ratio
Absolutely None (853)	56,331	66.0
Other (1,569)	89,157	56.8
Python (578)	33,641	58.2
Veteran (75)	3,700	49.3
No Response (410)	24,099	58.7
All (3,485)	206,928	59.4

We seek to understand whether students who succeed in the course reach solutions similar to these instructor-intended gold standard solutions, in terms of the concepts and mechanisms they contain. We also explore what feature representations are sufficient for code submission history, since they are directly applicable to the development of automated tutors for progress tracking.

Table 3.1 shows the count of certified students by experience and gives for each experience category: 1) the total number of submission attempts for that category and 2) the attempts to student count ratio for each experience category (note that this is not a distribution but a quotient). Observe that the Absolutely None experience category has the highest attempts per student ratio at 66.0, while the Veteran experience category has the lowest ratio at 49.3 attempts per student. This observation is unsurprising if we expect veteran coders to solve problems with fewer submissions on average than novices.

Specific Research Questions. How many students achieve correct answers without using concepts the instructors intended for them to practice? Course instructors hypothesize that the students with no prior coding experience are the ones whose answers come closest to the intended solutions. We develop distance metrics to quantify this disparity across the 6.00.1x student population by experience level. Do students learn to conform to the instructors' intentions as the course progresses, and does prior experience have correlations with student behavior? Are we able to

see which students watched the most relevant video based on their submissions to a problem?

3.1 Scraping User Submission Histories

Table 3.2: 6.00.1x Coding Problem Scraping Statistics

Problem	Task Description	Total Subs.	Cert. Students Attempted	% Cert. Attempted
1-1	Count Vowels	7284	3257	93.5
1-2	Count 'bob'	7104	3232	92.7
1-3	Longest Alphabetical Substring	10964	3163	90.8
2-1	Paying Debt off in a Year	8917	3260	93.5
2-2	Paying Debt off in a Year	7155	3214	92.2
2-3	Bisection Search	6564	3171	91
3-1	Is the Word Guessed	8612	3321	95.3
3-2	Printing Out the User's Guess	6557	3306	94.9
3-3	Printing Out all Available Letters	6236	3301	94.7
3-4	The Game	20615	3265	93.7
4-1	Word Scores	6125	3300	94.7
4-2	Dealing with Hands	6432	3291	94.4
4-3	Valid Words	7965	3279	94.1
4-4	Hand Length	5410	3282	94.2
4-5	Playing a Hand	19013	3239	92.9
4-6	Playing a Game	10760	3150	90.4
4-7	You and your Computer	14015	3036	87.1
5-1	Build Shift Dictionary & Apply Shift	15282	3040	87.2
5-2	Plaintext Message	11909	2864	82.2
5-3	Ciphertext Message	12827	2766	79.4
5-3	Decrypt a Story	7182	2686	77.1

The official edX 6.00.1x database contains only student final attempts. To curate a dataset of student submission histories to coding problems, we utilize the edX instructor interface, which allows instructors to view student submission histories by username in the browser. We wrote an automated scraping tool to obtain the raw code and tutor response history, which includes timestamps, correctness, and some test case information in addition to code submissions. We summarize the scraped submission history data in Table 3.2.

Table 3.3: Most Relevant Videos for Problems 1-1, 2-3, and 4-7.

Problem	Video
1-1: Count Vowels	Control Flow (Unit 1)
2-3: Bisection Search	Bisection Search (Unit 2)
4-7: You & Your Computer	Exceptions as Control Flow (Unit 4)

Notice a drop-off in the percentage of certified students attempting problems later in the course in Table 3.2. Course instructors believe this drop-off is due to the 55% overall grade cutoff that allows a student to gain 6.00.1x certification. If a student received full credit on all problems until the middle of Problem Set 4, the student would have already earned the passing grade for certification and could stop there.

3.2 Most Relevant Videos

The teaching philosophy of 6.00.1x uses analogy to help students write their own code. For example, the Bisection Search video translates quite directly to solution code for 2-3: Bisection Search Makes the Program Faster. We hypothesize that for these problems, students who did watch the relevant video have solutions closer to each other and to the gold standard than students who did not watch the relevant video.

The course does not specify an exact one-to-one mapping for which video lecture is relevant to which problem, especially since later problems often combine multiple concepts from previous video lectures. However, we create an approximate one-to-one mapping of the *most* relevant or central video lecture corresponding to three coding problems of interest: 1-1, 2-3, and 4-7, in order to probe whether students with distinct video engagement behaviors (i.e. watching or not watching the video) show distinct behaviors in submitted code as well. The video-problem correspondences we choose are given in Table 3.3.

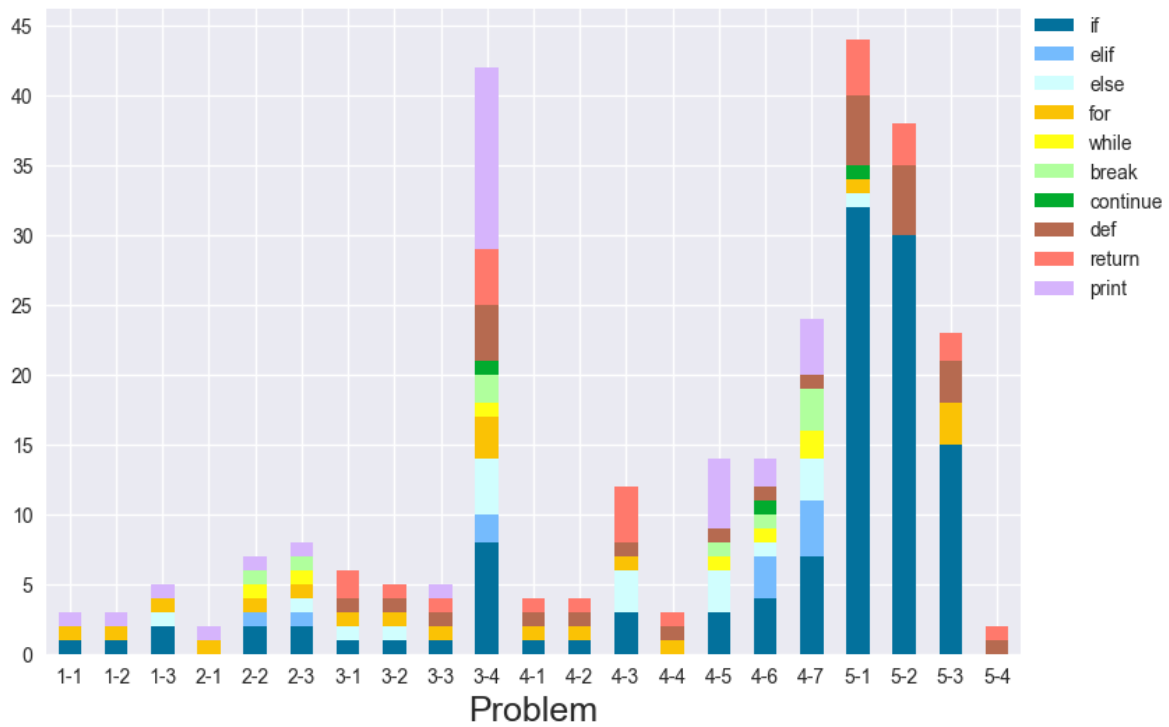

Figure 3-1: **Gold standard solutions.** We create gold standard example solutions for 6.00.1x graded coding problems based on what the instructors intend and expect students to write. Importantly, these gold standard solutions contain all key aspects of the problem as they were covered in preceding material. Gold standard solutions are visualized here in terms of keyword occurrence features.

3.3 Keyword Occurrence Features

Keyword occurrence features are similar in principle to bag-of-words features with stopwords removed; we simply restrict the allowed vocabulary to 10 Python keywords. These keyword occurrence counts are visualized in stacked bar charts. For the level of Python used in 6.00.1x, the 10 keywords are *if*, *elif*, *else*, *for*, *while*, *break*, *continue*, *def*, *return*, and *print*. The motivation for starting with these simple count-based features is that we could establish a baseline from which to quantitatively measure improvement, if any, from using features which account for order or logical dependencies, such as abstract syntax trees, API calls, and control-flow.

Gold Standard Solutions. There are no official 6.00.1x instructor solutions. We develop gold standard solutions that contain all key aspects of the problem as they were covered in preceding material; therefore, these gold standard solutions are based

on what the instructors intend and expect engaged students to write. We visualize these gold standard solutions in Figure 3-1. We aim to compare student submissions to these known gold standards and quantify the similarity between them.

Similarity Measures. We compare submissions for a problem using keyword occurrences and Pearson correlation[1] as a distance measure. These similarities are visualized in heatmaps, as in Figures 3-2, 3-3, 3-4, and 3-5, and give a quantitative measure of each submission's similarity to the instructor-intended solution. Probing whether students who passed test cases also achieved solutions similar to the gold standard – suggesting they have included key concepts in their own code submissions – helps us understand whether students are properly learning the material taught.

3.4 Anecdotal Findings

Inspecting code trajectories for individual problems and students gives us a sense of the behaviors we can observe anecdotally in something as simple as the keyword occurrence features. We describe some here and later investigate whether it is possible to detect or recover these behaviors in the aggregate dataset.

We show examples of student trajectories for three problems. Course instructors believe these problems represent the types of graded coding problems appearing in 6.00.1x. 1-1: Count Vowels is the simplest problem; the standard solutions uses exactly one *if* statement, *for* loop, and *print* statement. 2-3: Bisection Search to Make the Problem Faster is the most algorithmic problem, since users must implement bisection (binary) search from scratch. 4-7: You and Your Computer relies on complex control flow to handle edge cases and requires students to combine code from previous problems. We compare student trajectories to gold standard solutions (Figure 3-1), using a quantitative distance measure (Pearson correlation) for code similarity.

[1]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

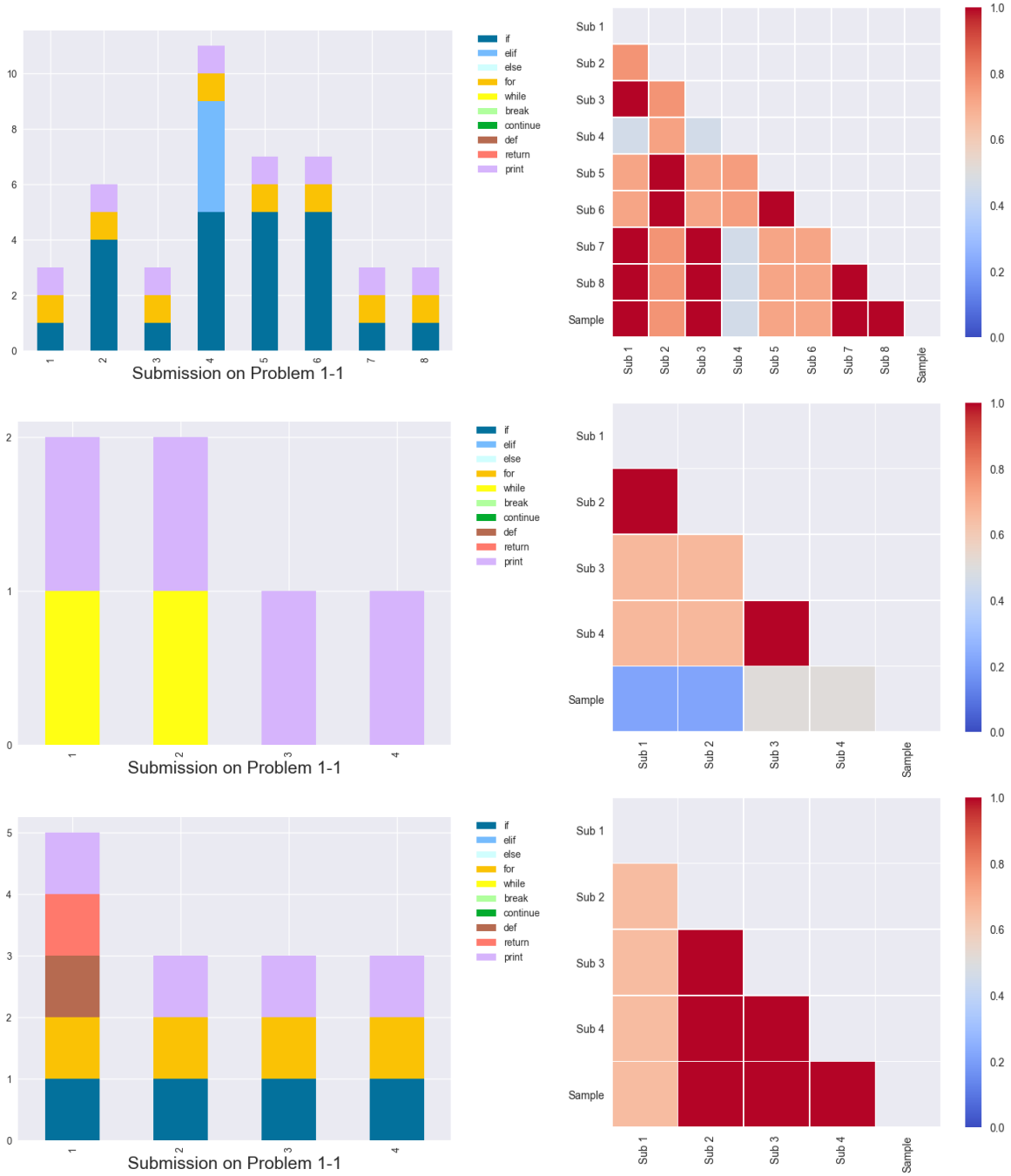

Figure 3-2: **Anecdotal 1-1 trajectories for different students.** We compare trajectories of three different students with varying prior experience who achieve correct answers on 1-1: Count Vowels. *Top*: A student with no coding experience makes more, seemingly-erratic attempts but ultimately comes close to the gold standard solution. *Middle*: This student's solution is unlike the gold standard. The student indicated experience with another language and used the Python string count function even though it was not taught or necessary to use. *Bottom:* This student, who indicated prior experience with Python, initially included a function definition that was later removed.

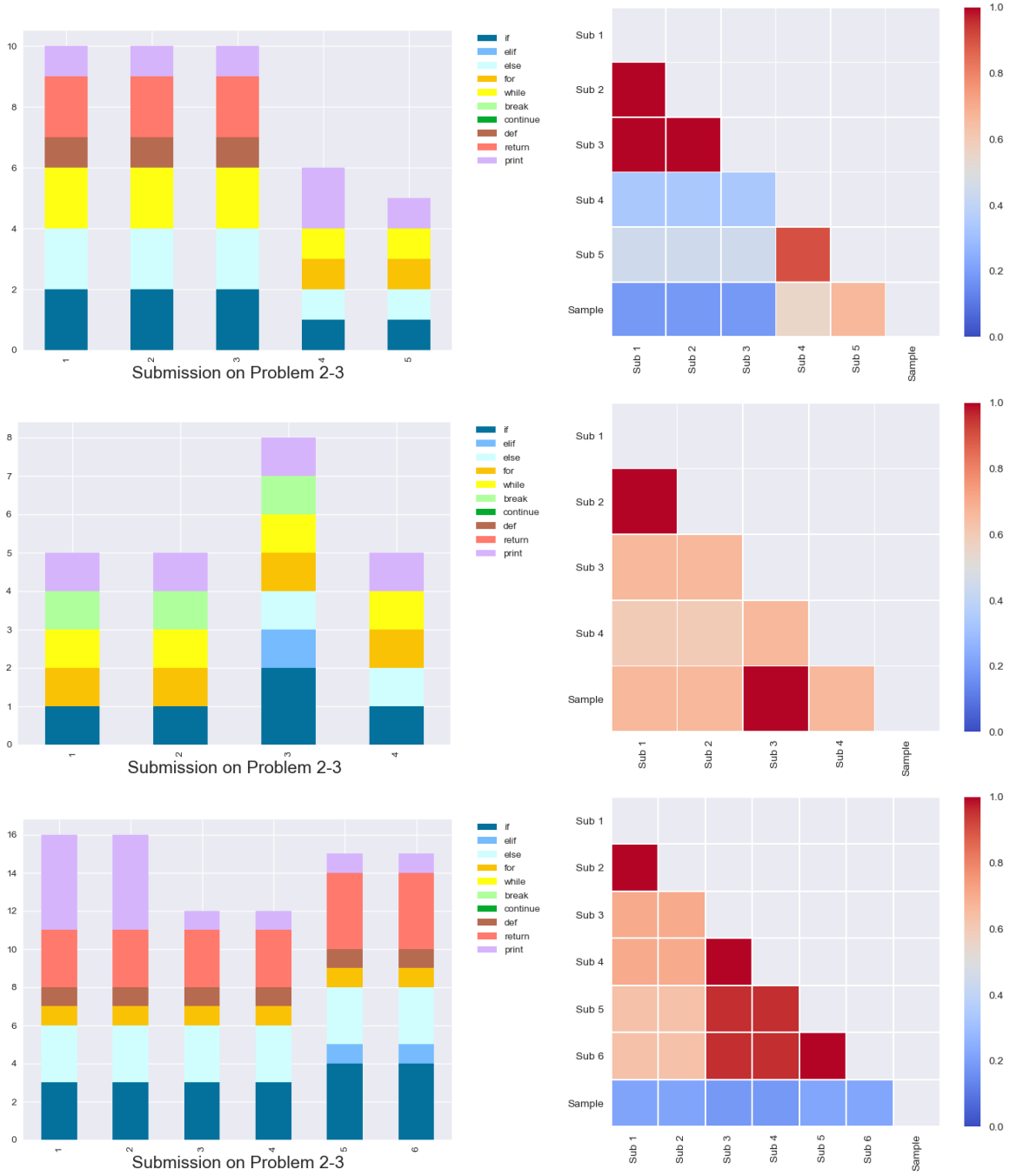

Figure 3-3: **Anecdotal 2-3 trajectories for different students.** We compare trajectories of three different students with varying prior experience who achieve correct answers on 2-3: Bisection Search. *Top:* A student with no prior coding experience starts by using a function definition, then removes it and uses a for loop. *Middle:* A student who indicated experience in another language changes the conditionals used and omits the break statement. *Bottom:* A student who knows Python makes submissions very different from the gold standard by using function definitions.

We visualize student code trajectories on a *per student, per problem* basis (Figures

3-2, 3-3, 3-4, and 3-5), using keyword occurrence features in comparative bar plots and heatmaps based on Pearson correlation as a measure of closeness. We expect that keyword occurrence features are not sufficient to capture the algorithmic aspects of 2-3 (Note that in Figure 3-5, the last set of the student's 2-3 submissions look identical), though they may be sufficient for a straightforward problem such as 1-1.

1-1 is a very simple problem for which the gold standard solution is just a few lines, but student submissions vary greatly as in Figure 3-2. While the average number of attempts is just 2.2 (see Table 2-4), looking at slightly longer submission histories makes this variety even clearer. Figure 3-2 shows keyword occurrence trajectories for a novice, a student experienced with another language, and a student experienced with Python. One interesting observation is that the student with knowledge of another language uses a built-in Python string count function[2], suggesting that those experienced in other languages may look for Python analogues to concepts they have seen elsewhere. In contrast, the students with no prior experience and the students who know Python may be more likely to come up with solutions close to the gold standard. However, Figure 3-2 illustrates the *ways* in which novices and experienced coders may differ in their understanding. We could reasonably interpret the experienced student's trajectory (bottom) as having to change conceptually valid code to fit the testing format, while the novice student's trajectory (top) represents actual conceptual advances. The novice also shows erratic behavior in the form of code growth and shrinkage.

The intuitions we see for 1-1 are slightly less clear in the case of 2-3. By this point in the course, even novice programmers have been exposed to function definitions (during Unit 2 material), and we see a novice initially using a function definition that is later removed in Figure 3-3. We also observe in Figure 3-3 that the student who knows Python writes a very different solution compared to the gold standard (using function definitions and just a *for*, rather than a *for* and a *while*), while the student with other language experience ends up with a solution similar to the novice's.

[2]https://docs.python.org/3/library/stdtypes.html?highlight=count#str.count

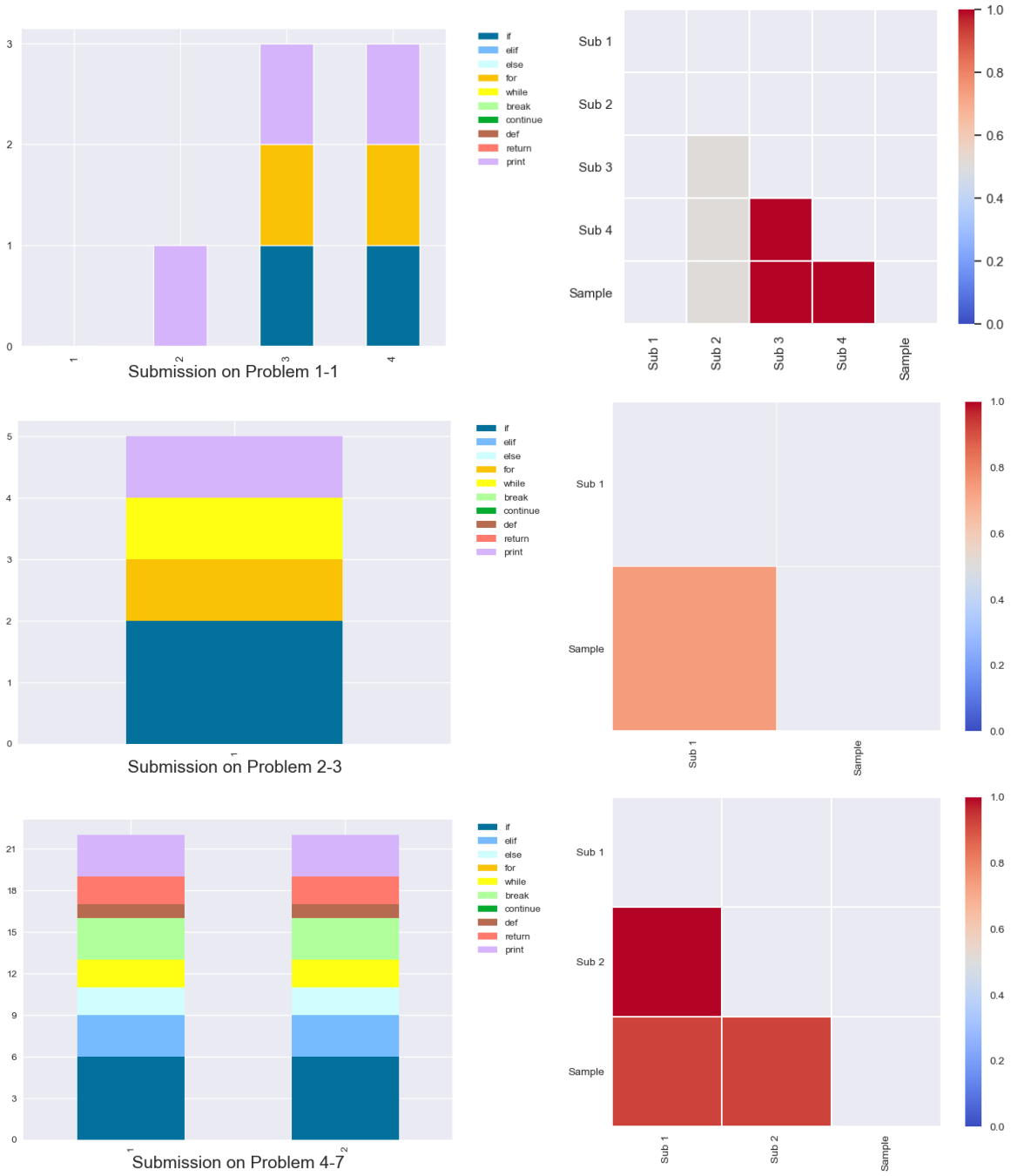

Figure 3-4: **Anecdotal course trajectory for one "absolutely none" student.** A student with no prior coding experience has trajectories to correct solutions for 1-1: Count Vowels (*top*), 2-3: Bisection Search (*middle*), and 4-7: You and Your Computer (*bottom*). This student has very few submissions to the harder and later problems compared to the first problem, indicating that the student may have started using a local environment to test after the first problem, and ultimately comes very close to the gold standard in all cases.

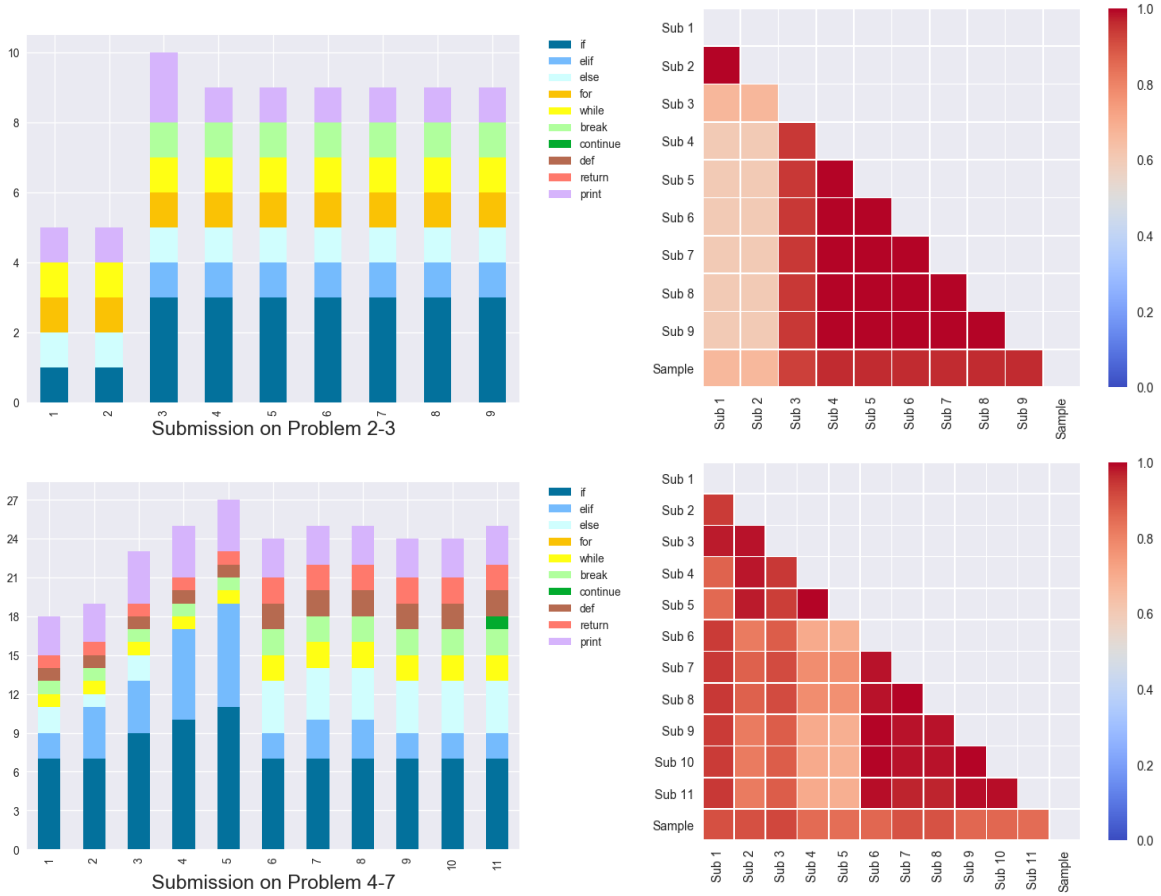

Figure 3-5: **Anecdotal course trajectory for one "know Python" student.** This Know Python student has trajectories to correct solutions for 2-3: Bisection Search (*top*) and 4-7: You and Your Computer (*bottom*). The 2-3 solution approaches the gold standard, and the 4-7 solution diverges only slightly from it.

We briefly consider student trajectories over multiple problems, as in Figures 3-4 and 3-5, since many general motivations behind researching learning trajectories pertain to assisting or personalizing a MOOC for individual students. The student in Figure 3-4 with no prior coding experience comes close to the gold standard in all cases, and in fact reduces the number of submission attempts in later problems. Figure 3-4 also suggests that the student may have switched to using a local environment to test rather than using the tutor as a test environment. The student in Figure 3-5 has prior knowledge of Python and comes close to the 2-3 gold standard solution (100% similarity in the keyword occurrence feature space) and relatively close to the 4-7 gold standard solution. Figure 3-5 also suggests code shrinkage or refactoring over time.

3.5 Final Submission Similarity vs. Correctness

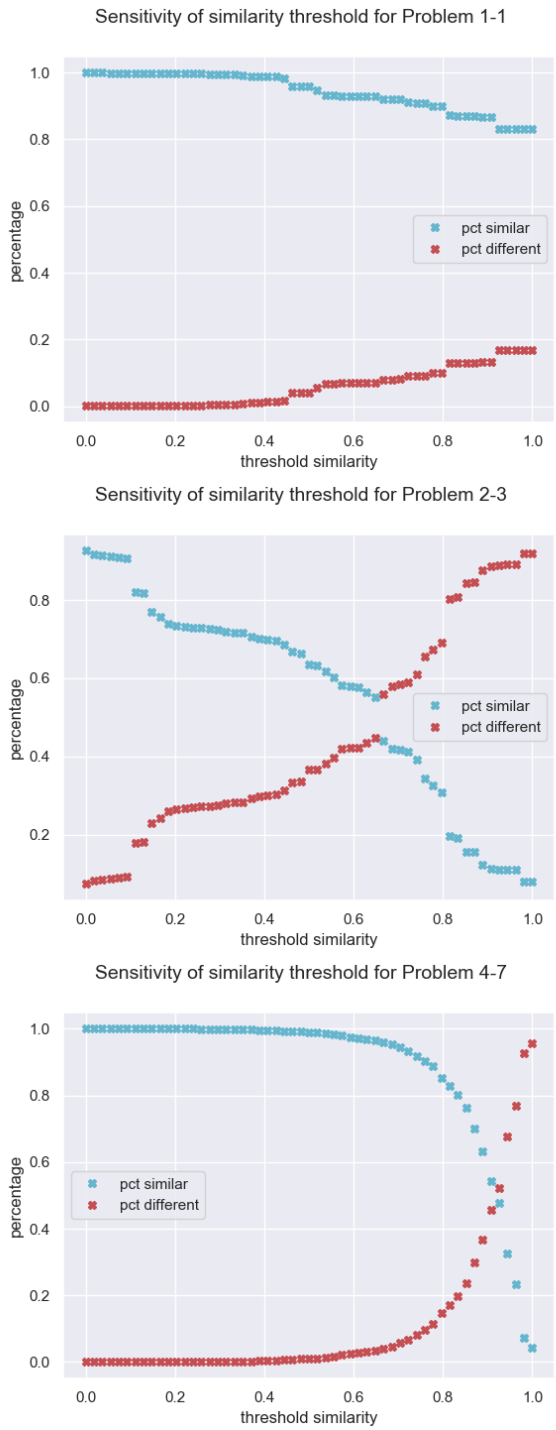

Figure 3-6: **Sensitivity sweeps for 1-1, 2-3, and 4-7.** These sweeps show the percentage of student final submissions that are similar to the gold standard keyword occurrence features for thresholds between 0% and 100% similarity for 1-1, 2-3, and 4-7. Observe that the shape of the curve is different for each problem.

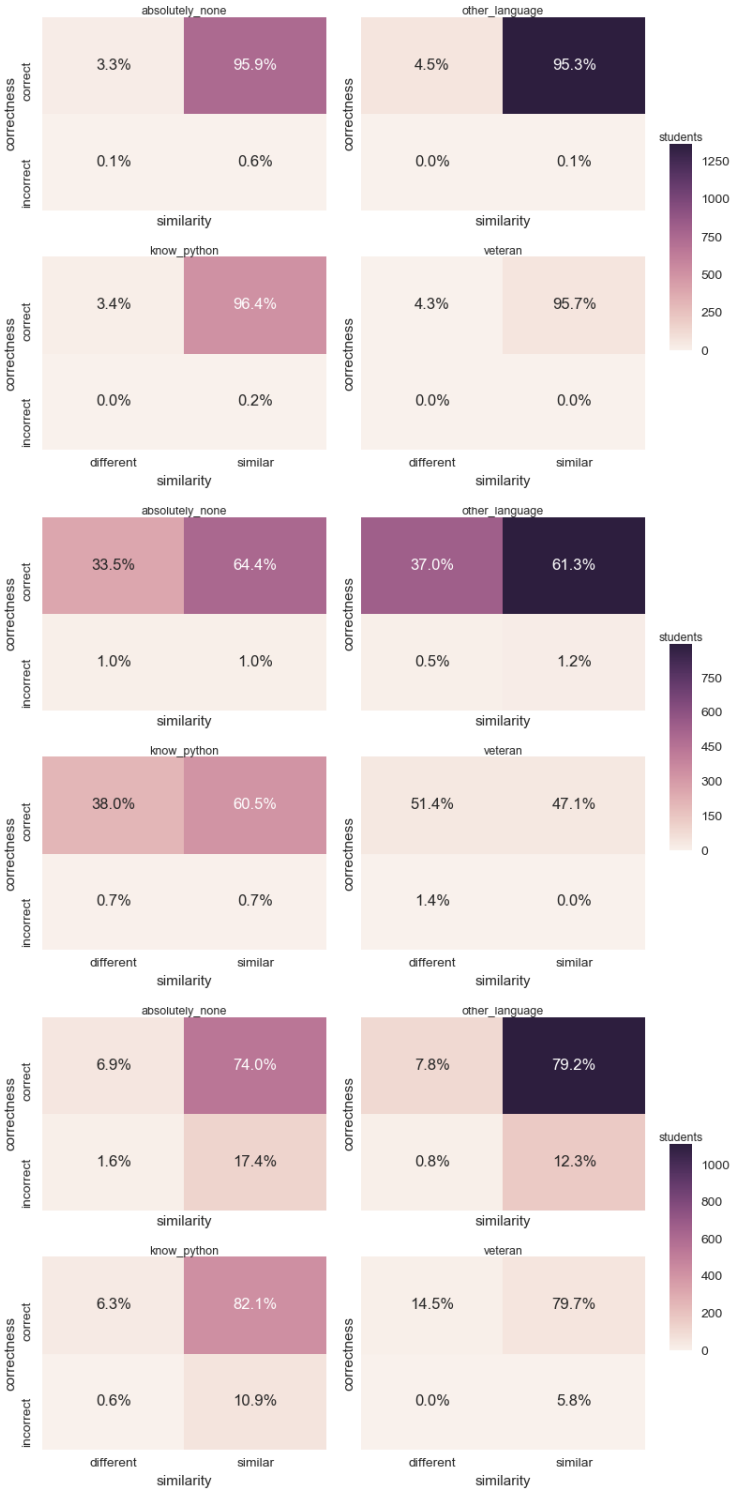

Figure 3-7: **Correctness-similarity counts by experience level.** We bin students based on *correctness* (a boolean of passing all test cases) and *similarity* (a boolean of whether the solution is close to the gold standard given a threshold) of their final submission for problems 1-1 (*top*), 2-3 (*middle*), and 4-7 (*bottom*). Note that we use a similarity threshold of 50% for 1-1 and 2-3 but 75% for 4-7. We omit students who did not respond to the survey.

We next focus on the similarity of student final submissions to the gold standard for each problem. The question of where to draw the threshold of "similar" or "different" in terms of keyword occurrence features and Pearson correlation likely has a different answer for each problem. To determine similarity thresholds, we run a threshold sweep for individual problems. This sweep helps us determine, given the keyword occurrence features, at what point a large number of students converge or diverge to the gold standard. As we see in Figure 3-6, this point – as well as the shape of the similarity curve – can vary depending on the problem. Both 1-1 and 4-7 show similarity up to a high threshold, but the shape of the curve is different, resulting in the intercepts at a 100% similarity threshold being 80% of 1-1 students (meaning 80% of student final submissions have exactly the same keyword occurrence vector as the gold standard) and essentially 0% of 4-7 students (meaning no significant number of students produced solutions with keyword occurrence vectors very similar to the gold standard). Note that while the percentage of students with 100% similarity to the 2-3 gold standard is small, it is nonzero.

Based on these per-problem thresholds, we subdivide students into four categories for each problem based on *correctness* (a boolean of passing all test cases) and *similarity* (a boolean of whether the solution is close to the gold standard given a threshold) of their final submission. Correctness-similarity counts in Figure 3-7 show that for 1-1, the vast majority of students (over 95% for each experience level) achieve very similar solutions to the gold standard, despite the large difference in the total number of students within each experience level. Observe that the similarity threshold of 50% in Figure 3-7 would correctly distinguish the user trajectories in Figure 3-2.

Students of different experience levels appear to achieve significantly different similarities to the gold standard on some problems. Another conclusion from the varying similarity threshold sweeps is that the space of possible solutions is smaller for problems such as 1-1, and larger for others such as 4-7. The quantitative similarities to the gold standard, as in Figures 3-6 and 3-7, could be contextualized within this space of solutions. The next chapter focuses on utilizing generative modeling to characterize this space of code submissions and trajectories.

Chapter 4

Generative Model

A generative model allows us to characterize the space of code submissions and trajectories in our dataset while also giving us the ability to generate new samples representing submissions and trajectories. From a learning design perspective, we could map student characteristics or behaviors to parts of this space. We utilize a variational autoencoder (VAE) [11, 6, 3] to explore the latent space and its relationship to student characteristics or behavior. The input space of keyword occurrence vectors is relatively small at 10×1 for a single submission, or 300×1 for a trajectory.

To allow us to generate samples from the latent space, we assume it has the form of a Gaussian distribution parameterized by vectors $\mu(z)$ and $\sigma(z)$. The latent space itself is the posterior $p_\theta(z|x)$ of the input x, meaning latent representation z is the output of the *encoder* portion of the network with input x. The *decoder* network then reconstructs vectors representing code submissions or trajectories given a particular latent representation z. The decoding model is $p_\theta(x|z)$. The encoder and decoder are multilayer perceptron (MLP) networks with ReLU activations.

4.1 Variational Autoencoder Setup and Architecture

We summarize the inputs, outputs, estimation, and losses as proposed by Kingma and Welling [11]. Let θ be the MLP parameters and ϕ be the variational parameters;

the benefit of the VAE setup is that we can learn θ and ϕ concurrently.

Since the posterior $p_\theta(z|x)$ is intractable, we instead compute the variational approximation $q_\phi(z|x)$ (called the recognition model). This $q_\phi(z|x)$ represents the encoder, producing a distribution over values of z from which x could have been generated. We assume the latent space prior has the form of an isotropic unit gaussian $\mathcal{N}(z; 0, I)$, such that the log of the approximate posterior given a single input x is

$$\log q_\phi(z|x) = \log \mathcal{N}(z; \mu, \sigma^2 I) \tag{4.1}$$

We sample from this posterior $z^{(i,l)} \sim q_\phi(z|x^{(i)})$ using

$$z^{(i,l)} = g_\phi(x^{(i)}, \varepsilon^{(l)}) = \mu^{(i)} + \sigma^{(i)} \odot \varepsilon^{(l)} \tag{4.2}$$

where $\varepsilon^{(l)} \sim \mathcal{N}(0, I)$. The notation in [11] uses $x^{(i)}$ rather than x to denote a single input, but they are the same.

The decoder is then $p_\theta(x|z)$, producing a distribution over x given a particular z. One key concept that makes it possible to compute $q_\phi(z|x)$ is the reparameterization trick introducing the differentiable transform $g_\phi(x^{(i)}, \varepsilon^{(l)})$ in [11].

The estimated (decoded) output is $p_\theta(x|z)$, and it is the expectation of this quantity, or $E_{z\sim q_\phi(z|x)}\Big[\log p_\theta(x|z)\Big]$, that is considered in computing the loss.

We train the VAE using a combination of reconstruction loss (mean squared error, or MSE) given the true input and decoded output and the Kullback-Leibler divergence (KL divergence)[1] of $q_\phi(z|x)$ and $p_\theta(z)$, using the closed-form

$$-D_{KL}(q_\phi(z|x)||p_\theta(z)) = -\frac{1}{2}\sum(1 + log(\sigma^2) - \mu^2 - \sigma^2) \tag{4.3}$$

where μ is $\mu(z)$ and σ is $\sigma(z)$. The motivation for including KL divergence in the loss function is to learn a well-formed latent space; it can be thought of as a form of regularization that assumes the prior $p_\theta(z)$ on the latent space is a good one.

[1] We obtain the closed form by choosing strategic conjugate priors over z that allow integration to a closed-form, as in [11] (Appendix B).

Single submissions. To characterize the space of sample solutions as single submissions in the keyword occurrence feature space, we utilize a VAE, shown in Figure 4-1. By parametrizing the latent space between the encoder and decoder, the VAE can, through sampling, generate new keyword occurrence feature vectors representing code submissions not in the original dataset. Figure 4-2 gives a detailed description of the layers and network dimensions for the single-submission case.

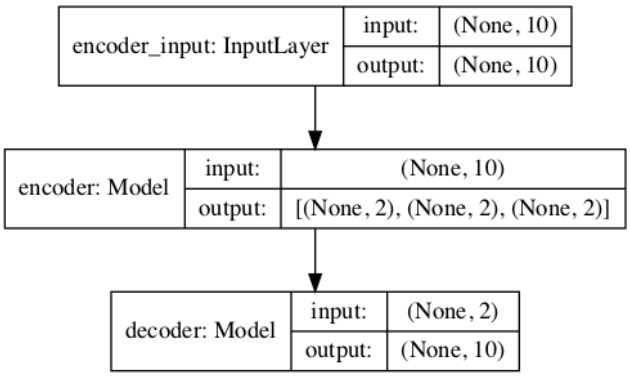

Figure 4-1: **Single-submission VAE architecture summary.** The encoder network (encoder: Model) maps 10 dimensional keyword occurrence vectors (encoder_input: InputLayer) representing individual submissions to a parameterized 2D latent space. The decoder network (decoder: Model) then reconstructs 10D keyword occurrence vectors from the latent space variables. Hidden layers are feedforward, fully-connected with ReLU activations.

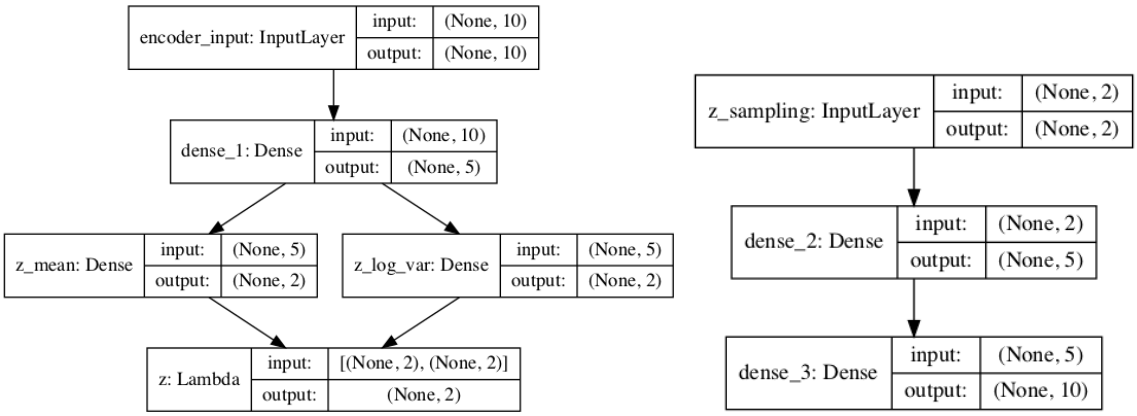

Figure 4-2: **Single-submission encoder and decoder.** *Left:* The encoder portion of the VAE maps 10 dimensional keyword occurrence vectors representing individual submissions to a 2D latent space parameterized by $\mu(z)$ and $\sigma(z)$. *Right:* The decoder reconstructs 10D keyword occurrence vectors from the latent space variables. The hidden layers are feedforward, fully-connected with ReLU activations.

Trajectories. We also train an architecture that takes in a flattened trajectory as input, shown in Figure 4-3 (analogous to Figure 4-1). The trajectory matrices are represented by 10×30 matrices, or 300×1 vectors once flattened. Therefore this architecture has many more units to transform the input into a 2D latent space. A more detailed view of the encoder and decoder are given in Figure 4-4 (analogous to Figure 4-2). In this case, the latent space being learned between the encoder and decoder represents the trajectory space rather than the space of individual submissions.

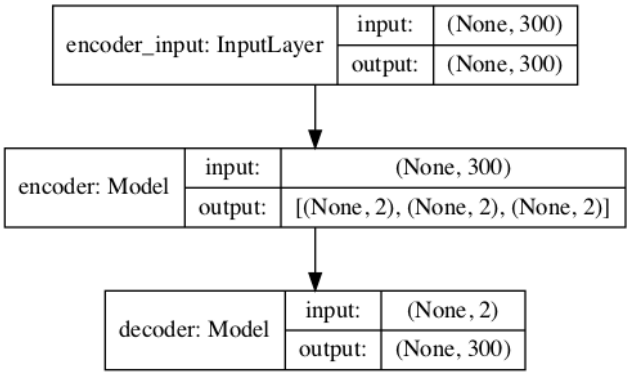

Figure 4-3: **Flattened trajectory VAE architecture summary.** The encoder (encoder: Model) maps 300×1 keyword occurrence vectors (encoder_input: InputLayer) representing trajectories to a parameterized 2D latent space. The decoder (decoder: Model) then reconstructs 300×1 keyword occurrence vectors from the latent space variables. These vectors tend to be sparse since students rarely use up to 30 submissions. Hidden layers are feedforward, fully-connected with ReLU activations.

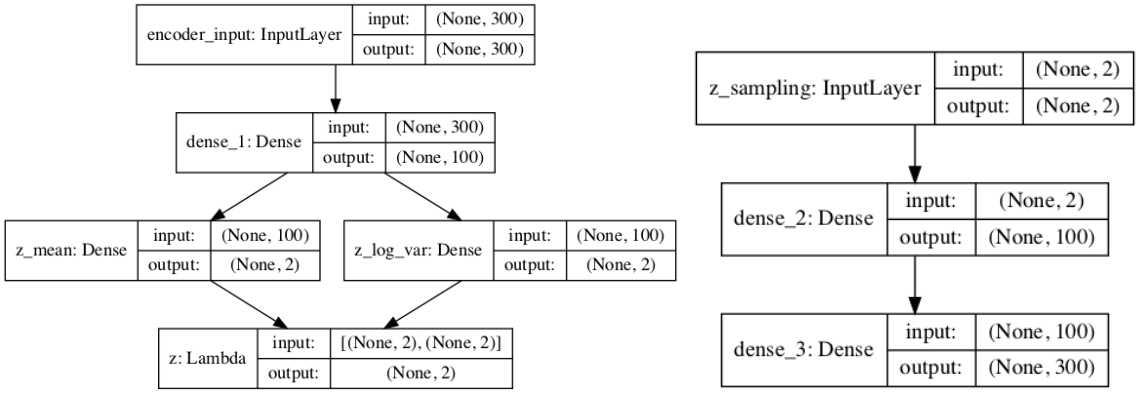

Figure 4-4: **Flattened trajectory encoder and decoder.** *Left:* The encoder maps 300×1 keyword occurrence vectors representing trajectories to a 2D latent space parameterized by $\mu(z)$ and $\sigma(z)$. *Right:* The decoder reconstructs 300×1 keyword occurrence trajectory vectors from the latent space variables. The hidden layers are feedforward, fully-connected with ReLU activations.

4.2 Training and Validation

Single submissions. We train the VAE to simply reconstruct keyword occurrence inputs for a given problem, then sample the 2D latent space between the encoder and decoder using the learned parameters $\mu(z)$ and $\sigma(z)$. We consistently achieve <0.01 training loss and <0.01 validation loss when training on 1-1 submission data, suggesting the model is sufficiently complex to capture the keyword occurrence data, yet does not overfit to the training set. We show training loss, composed of reconstruction loss (mean squared error, or MSE) and KL divergence, along with validation loss, in Figure 4-5.

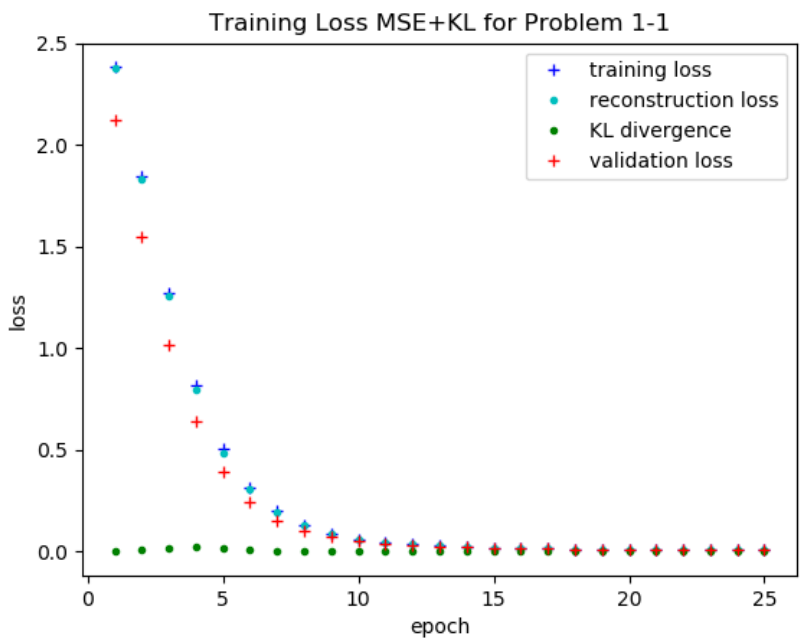

Figure 4-5: **VAE single submission training loss.** The VAE loss function is composed of the reconstruction loss (mean squared error, or MSE) and KL divergence. We plot total training loss with its reconstruction and KL components as well as total validation loss for each epoch of training. We consistently achieve <0.01 training and validation loss on 1-1 data with 25 epochs of training.

We achieve similar results when training the VAE on all the submissions for a given problem and on only the *final* submissions for the same problem. Note that this form of input data does not encode trajectories, but still allows us to probe the

space of sample solutions.

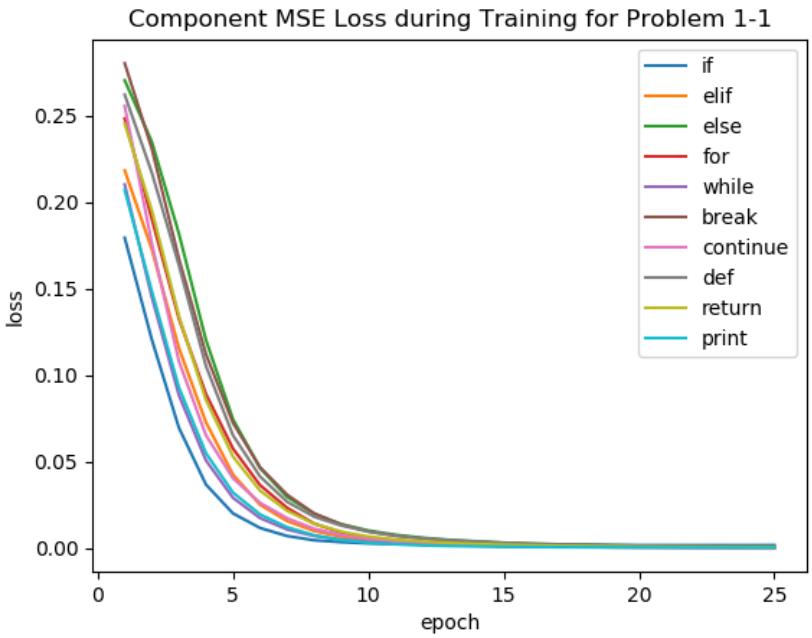

Figure 4-6: **VAE single submission component-wise training loss.** We show the component-wise reconstruction loss (MSE) for each of the Python keywords.

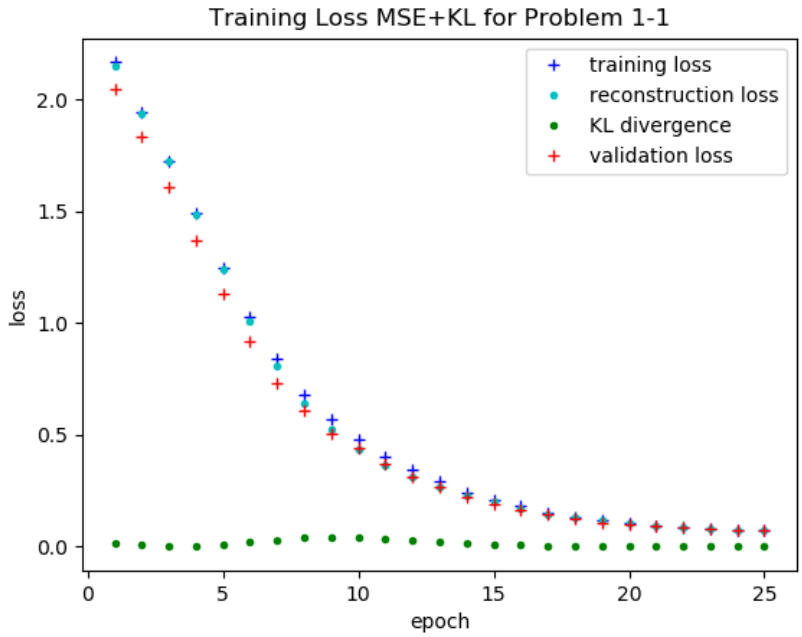

Figure 4-7: **VAE single submission training loss (final submission only).**

Trajectories. To incorporate the notion of trajectories into our modeling of the VAE latent space, we train the larger VAE to reconstruct trajectory data, where each input is a 300x1 vector (and often very sparse). Though the training loss (where only the reconstruction loss is significant) is a little higher, around 0.01 rather than strictly below it, we observe convergence as before.

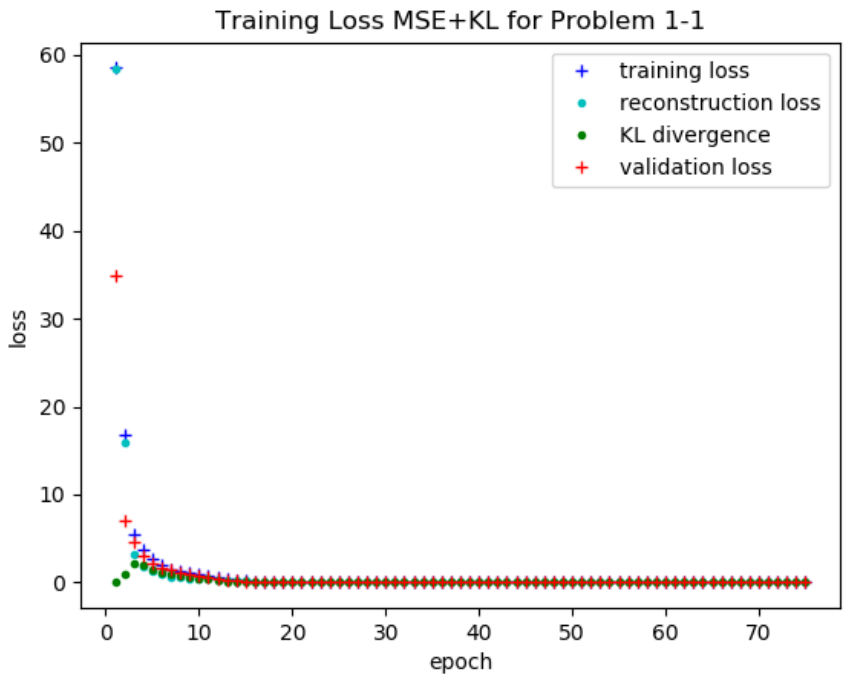

Figure 4-8: **VAE flattened trajectory training loss.** The VAE loss combines the reconstruction loss (MSE) and KL divergence. Total training loss is shown with reconstruction and KL components as well as total validation loss. We consistently achieve around 0.01 training and validation loss on 1-1 data with 75 epochs of training.

The training and validation losses we see for 1-1 are similar to what we obtain for other problems. These observations make us reasonably confident that the model is expressive enough for the keyword occurrence feature vectors yet does not overfit. We next wish to visualize the latent space as a whole, rather than as individual points, and map portions of the latent space to student video engagement or experience level.

4.3 Visualizing the Latent Space

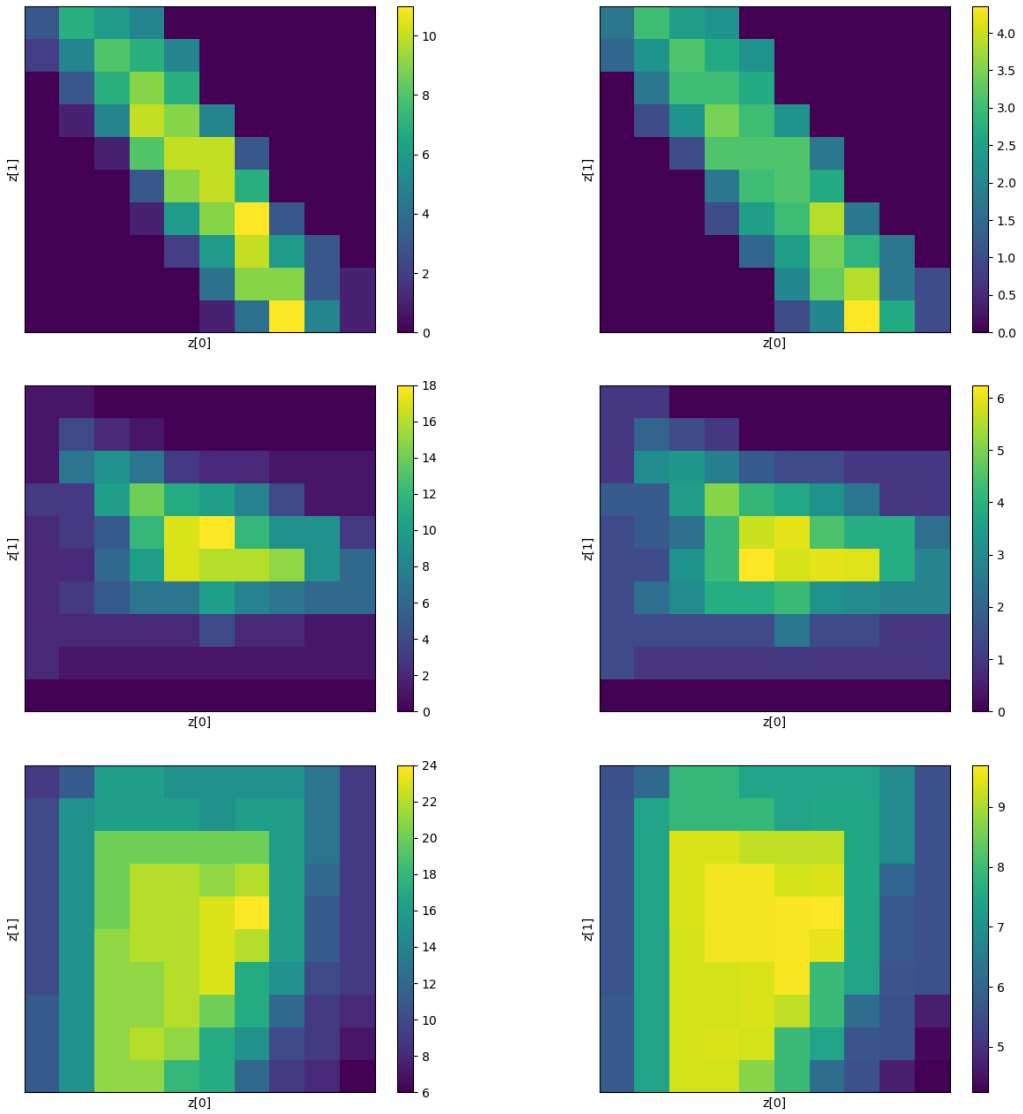

Figure 4-9: **Visualization of the single submission latent space in L1 and L2 norms for 1-1, 2-3, and 4-7.** The L1 (*left*) and L2 (*right*) plots look similar, but the scales are different. Observe that the latent space for 4-7 contains more regions of large L1 and L2 norms.

In the case of a standard image dataset such as MNIST[2], we know that the information we would like to predict is contained in the images. See Appendix B for the kinds of results we obtain from the latent space of a VAE trained on MNIST. In the case of

[2]https://keras.io/datasets/#mnist-database-of-handwritten-digits

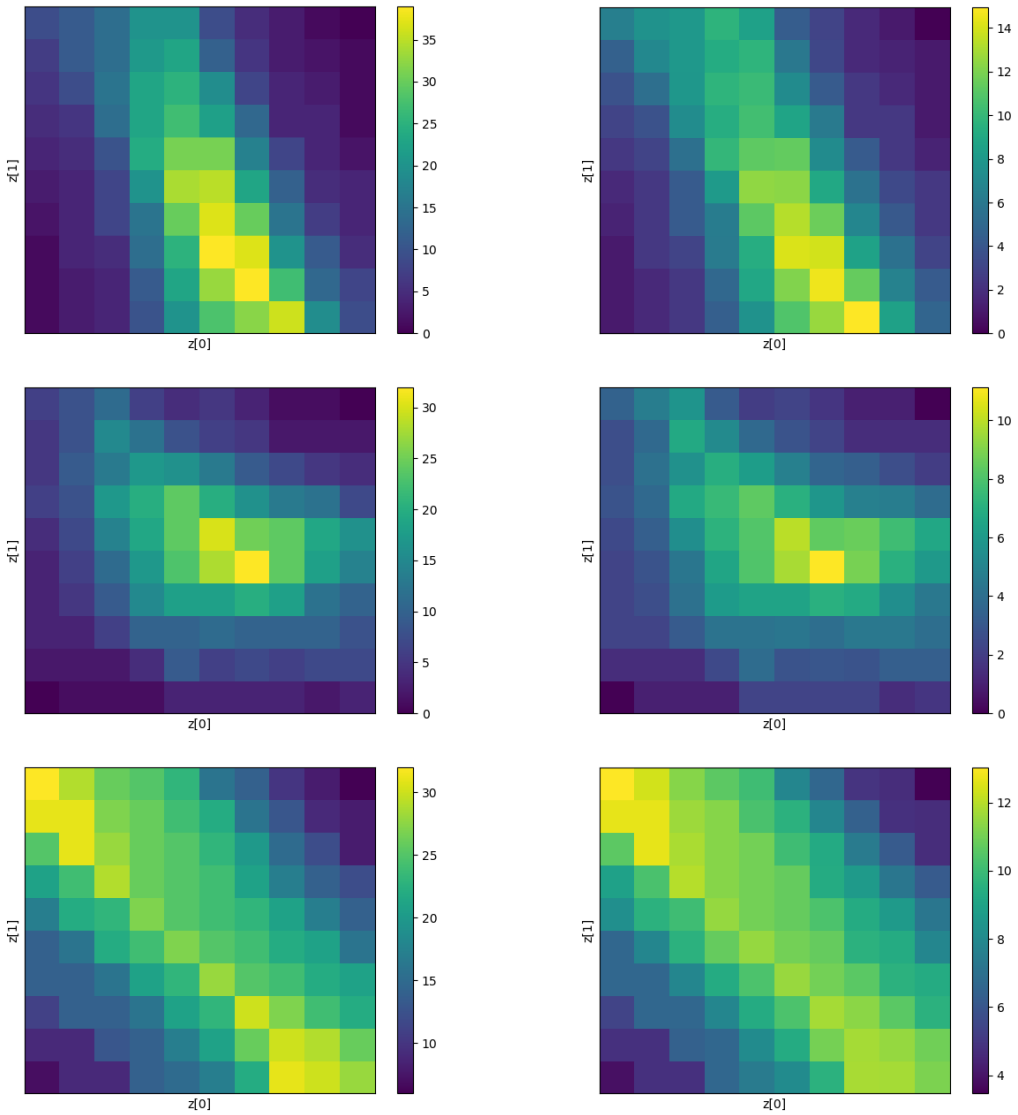

Figure 4-10: **Visualization of the *final* submission latent space in L1 and L2 norms for 1-1 (*top*), 2-3 (*middle*), and 4-7 (*bottom*).** As above, the L1 (*left*) and L2 (*right*) plots look similar but have different scales. The 4-7 final submission latent space contains more regions of large L1 and L2 norms.

student code and trajectory features, we aim to test the assumption that our feature representations of code and trajectories convey sufficient information about student characteristics and behavior, such as prior experience level or whether the student engaged with the video.

We visualize the latent space using the L1 and L2 norms of the reconstructed vectors representing code and trajectories that the decoder generates from sampled

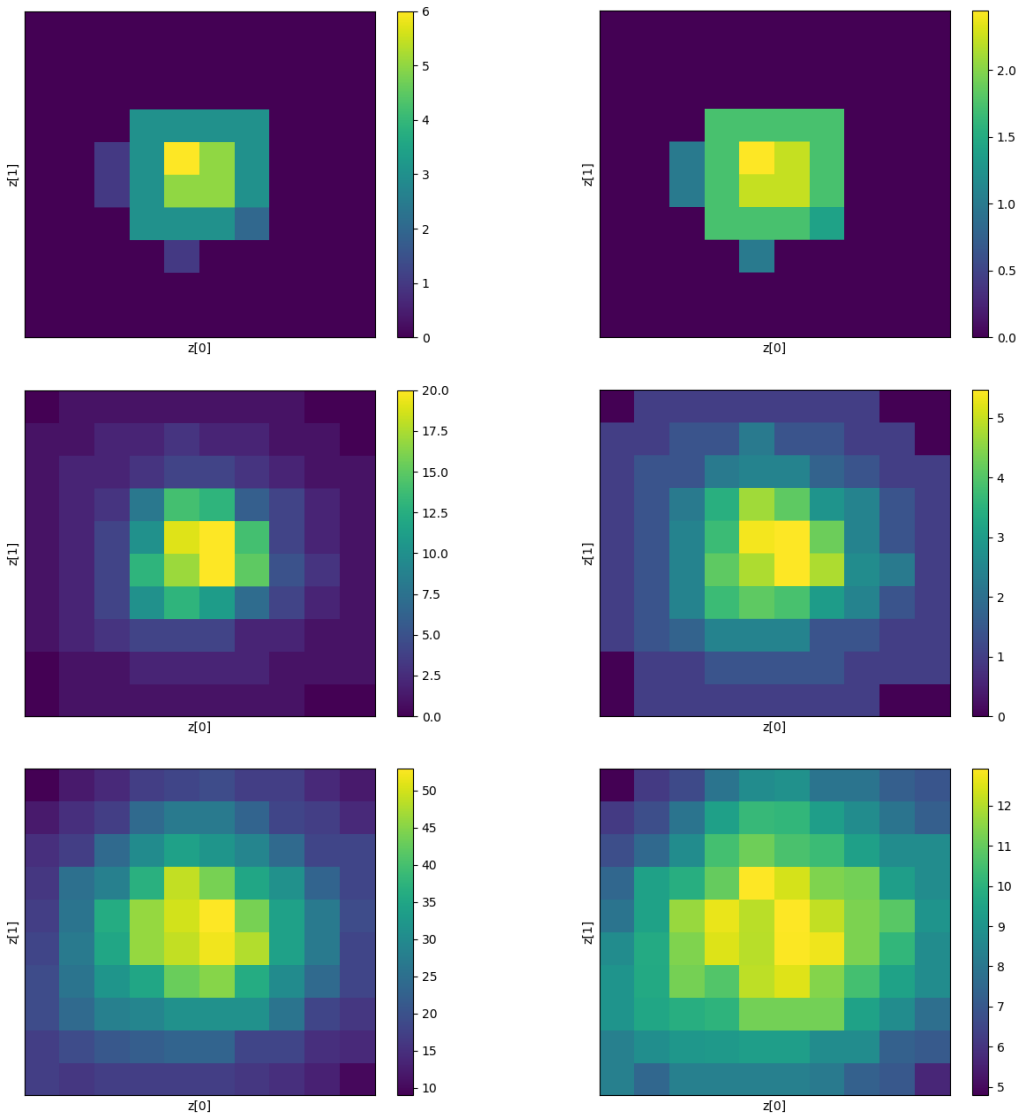

Figure 4-11: **Visualization of the trajectory latent space in L1 and L2 norms for 1-1 (*top*), 2-3 (*middle*), and 4-7 (*bottom*).** Interestingly, the trajectory latent space appears more symmetric and consistent across problems. This may be due to the high concentration of students with relatively sparse flattened trajectory vectors (few submissions).

points in the latent space. Each visualization is created by sampling across a grid of means in the 2 latent space dimensions.

Longer problems which occur later in the course have a larger range of L1 and L2 norms; this is particularly obvious in the L1 norm. Further, the space looks to have gradual change along both latent space variables axes, with one or two areas at which the maximum norm value occurs.

4.4 Relationship to Features of Interest

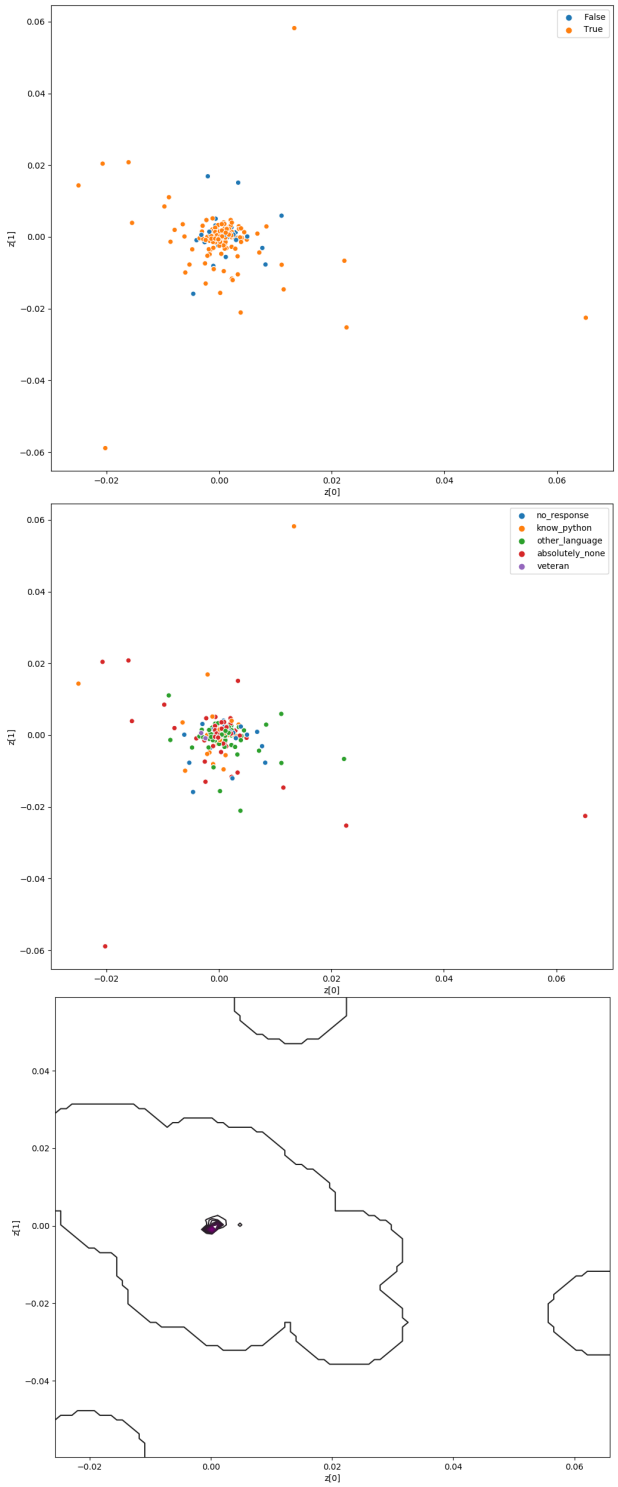

Figure 4-12: **Video engaged and experience level scatterplots, with density plot, in the latent space for 1-1.** We do not observe clear separation by video engagement (for the most relevant video: Control Flow, see Table 3.3) or experience level in the space for 1-1. The kernel density is very concentrated.

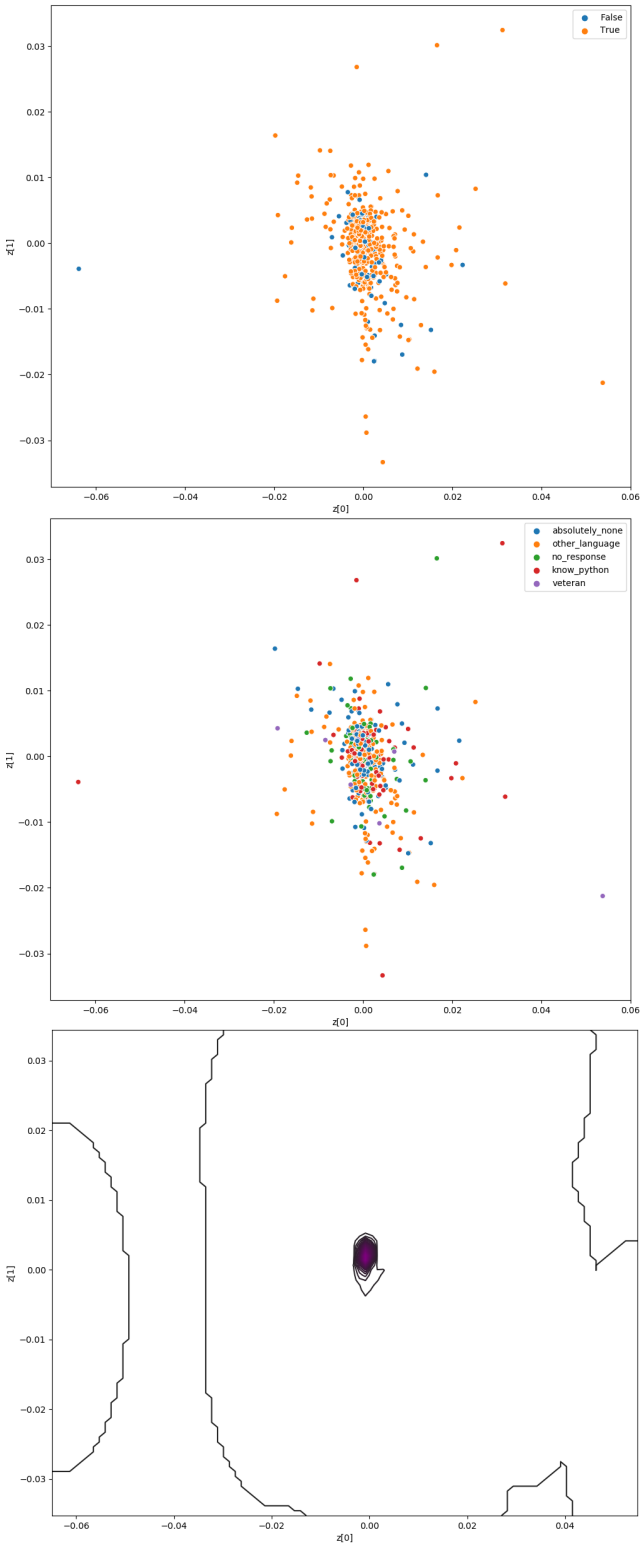

Figure 4-13: **Video engaged and experience level scatterplots, with density plot, in the latent space for 2-3.** As before for 1-1, we do not observe clear separation by video engagement (for the most relevant video: Bisection Search, see Table 3.3) or experience level in the space for 2-3. The kernel density is again very concentrated.

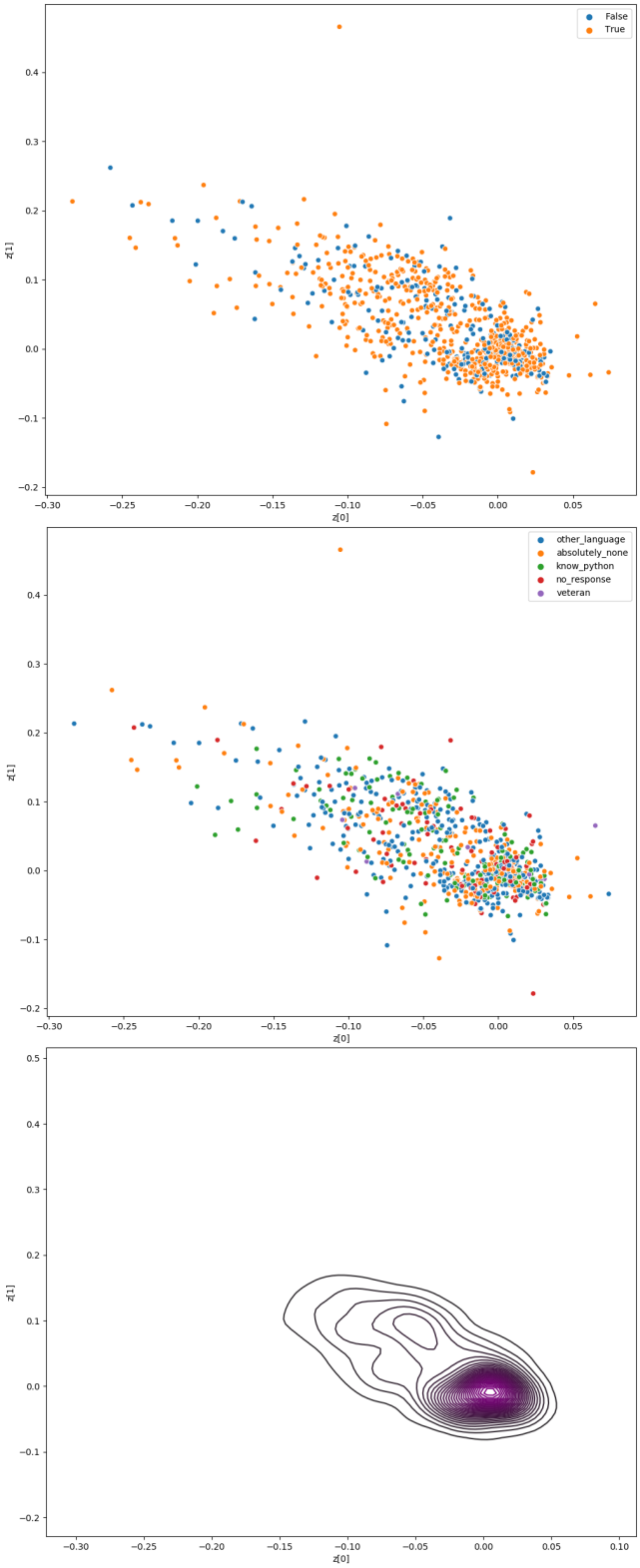

Figure 4-14: **Video engaged and experience level scatterplots, with density plot, in the latent space for 4-7.** As before, we do not observe clear separation by video engagement (for the most relevant video: Exceptions as Control Flow, see Table 3.3) or experience level in the space for 4-7. We do, however, see better spread in the kernel density plot.

We visualize each student trajectory in the latent space in Figures 4-12, 4-13, and 4-14, with students distinguished by video engagement (a boolean representing engagement with the most relevant video) or prior experience level. We do not see the kind of clustering observed for the MNIST dataset (see Appendix B).

We also report r^2 values for the correlation of the trajectory VAE's latent variables z_0 and z_1 to student video engagement and experience level. Very small significance values indicate that there is no clear correlation between the two latent variables and student video engagement behavior or the student prior experience level characteristic.

Table 4.1: r^2 Values for Trajectory VAE Latent Variable Correlations.

Problem	z_0 r^2 for Video	z_0 r^2 for Experience	z_1 r^2 for Video	z_1 r^2 for Experience
1-1	-0.009	-0.029	-0.021	0.041
2-3	0.016	0.022	0.047	-0.041
4-7	0.017	-0.008	-0.011	0.034

There are at least two reasons we may not observe a correlation between student attributes and trajectories. The first reason is a lack of data. To train on the MNIST dataset in a similar manner, we would have a total of 70,000 flattened $(28 \times 28) \times 1$ vectors, while for trajectories we have only 2,686 to 3,321 of the $(30 \times 10) \times 1$ flattened vectors for each problem (see Table 3.2). The second reason, which we suspect is more important, is that the keyword occurrence features are an insufficient description of student code trajectories, preventing us from discovering meaningful correlations between student trajectories and student behavior or experience. In terms of separating students, keyword occurrence does not sufficiently spread out trajectories, resulting in very concentrated kernel density plots.

We find that while is it possible to train generative models to represent student submissions and trajectories in a 2D latent space, it is difficult to relate the learned latent variables to student characteristics or behaviors that we observe anecdotally. We believe a more complex feature representation for code would bring us closer to the goal of characterizing the latent space.

Chapter 5

Conclusion and Future Work

We summarize some of the main results of this thesis, discuss limitations, and describe the future directions and extensions of this work.

Firstly, we obtained a new dataset of code submission histories that allows us to understand student behavior over the course of solving each problem. We also described this trajectory dataset and created ways to visualize student behavior anecdotally in terms of keyword occurrence features. Secondly, we defined research questions for this dataset based on the anecdotal behavior we observe for individual students or subsets of students. These keyword occurrence features are sufficient to probe some simple research questions, including how much student coding behaviors differ from the gold standard on aggregate. Trajectory features seem to illuminate differences across experience level – with respect to the gold standard – better than simple features such as number of attempts. Lastly, we have built and trained generative models to explore the space of individual submissions and per-problem trajectories. However, we were not able to find strong correlations between student video engagement behavior or prior experience level and the latent space variables of the trained models, suggesting that features more complex than keyword occurrence counts are required.

One limitation is that the self-identified experience level labels are noisy, since students with similar prior experience may reasonably self-identify in different cat-

egories (i.e. Know Python vs. Veteran). Students within the largest experience category (Other Language) likely vary in what languages they have seen, how similar those languages are to Python, and how similar their previous experiences were to problems in 6.00.1x. Further analysis of anecdotal trajectories, with a focus on code growth and shrinkage, could help us categorize the students in other ways.

We have not considered cheating behaviors. As previously discussed, CAMEO [22] is one such behavior, since nothing prevents users from creating multiple accounts. However, since we focus on coding problems within problem sets, to which official answers are not provided, it would be less straightforward to leverage a harvesting account. It would be possible to use multiple accounts to circumvent the limit of 30 attempts. Furthermore, course instructors have indicated that many solutions to these problems exist online and are not difficult to find. Solutions to 6.00.1x problems exist in personal GitHub accounts, in tutorial blog posts, and in forums; we do not account for the prevalence of these online solutions in our analysis. Identifying common cheating behaviors could change our analysis. For example, we could compare student submissions against the submissions of other students in addition to the gold standard and use timestamp information to identify likely duplication. We could also curate a set of solutions occurring in online forums and check submissions against these.

Finally, we are potentially limited by the dataset size in our ability to use generative models effectively, as well as the simplicity of the keyword occurrence features, which hinders informative separation in the latent space (especially for easier problems with less variation in submissions). Promising future directions include complex features such as abstract syntax tree (AST), control flow graph (CFG), and API call-based feature representations for code. These features account for order and structure in a way that keyword occurrence features do not. We could also try expanding the keyword features to include booleans and variables, without taking order into account. Since we believe the generative model would be more successful with a larger dataset, we could scrape and parse student submission histories from additional offerings of 6.00.1x to grow the dataset. Another method of obtaining a larger dataset would be to consider all students, rather than just those receiving certification.

Appendix A

Repository Structure

```
...
│ data/
│ │ bigquery_csv/ - CSV of official 6.00.1x database tables
│ │ features/ - directory for aggregated official 6.00.1x feature data
│ │ pickle/ - Pickle files of scraped submission trajectories
│ │ csv/ - CSV of submission trajectories
│ │ keyword_occurrence/ - CSV of trajectory keyword occurrence features
│ docs/ - contains paper submissions, posters, thesis
│ scripts/
  │ extract/ - extract data from database tables and to JSON files
  │ featurize/ - process standard 6.00.1x data, do data exploration
  │ scrape/ - scrape submission histories by usernames from 6.00.1x offerings
  │ models/ - test
```

Extraction. The Python notebooks and function files in `scripts/extract` can be used to download JSON data files from Google BigQuery and Google Cloud Storage and convert them to CSV files. The CSV files may then be used with `pandas`.

Making features and data exploration. Files in `scripts/featurize` allow us to process the raw data from the standard 6.00.1x dataset and create feature files for data exploration and preliminary modeling. Notebooks are available for data exploration and plotting, as well as for adding manual features such as difficulty rating.

Scraping, parsing, and visualizing trajectory data. The directory `scripts/scrape` contains scripts to scrape student submission histories from 6.00.1x offerings on MITx, parse through the scraped submissions, create keyword occurrence features, visualize trajectories, and investigate correctness-similarity counts. Bash scripts make running parsing, making features, and plotting for all problems more efficient.

Generative models. The directory `scripts/models` contains scripts to build variational autoencoder models and visualize the network architectures, training plots, and latent space. Both single submission and trajectory models are included.

Appendix B

Variational Autoencoder Latent Space for MNIST

To help build intuition about what we can learn from the latent space using a variational autoencoder, we show examples of a VAE trained on the MNIST dataset[1], a standard image dataset of handwritten digits. The images are 28x28 pixels, and the full dataset contains 70,000 labeled samples of digits between 0 and 9 inclusive.

It is easy to visualize the generated digit images that result from sampling a grid across the latent space, as in Figure B-1. These images are not in the original labeled MNIST dataset since they are generated samples. Notice that there is gradual change along each axis in Figure B-1. For example, the generated digits are most 0-like at large z_0 values and z_1 values close to 0, while the generated digits are most 7-like at large values of z_1 and small values of z_0.

We also observe distinct clustering behavior of the latent space means for each input labeled MNIST digit, as in Figure B-2. Though a continuous color bar is shown, the MNIST labels are the integers 0 through 9.

[1] https://blog.keras.io/building-autoencoders-in-keras.html

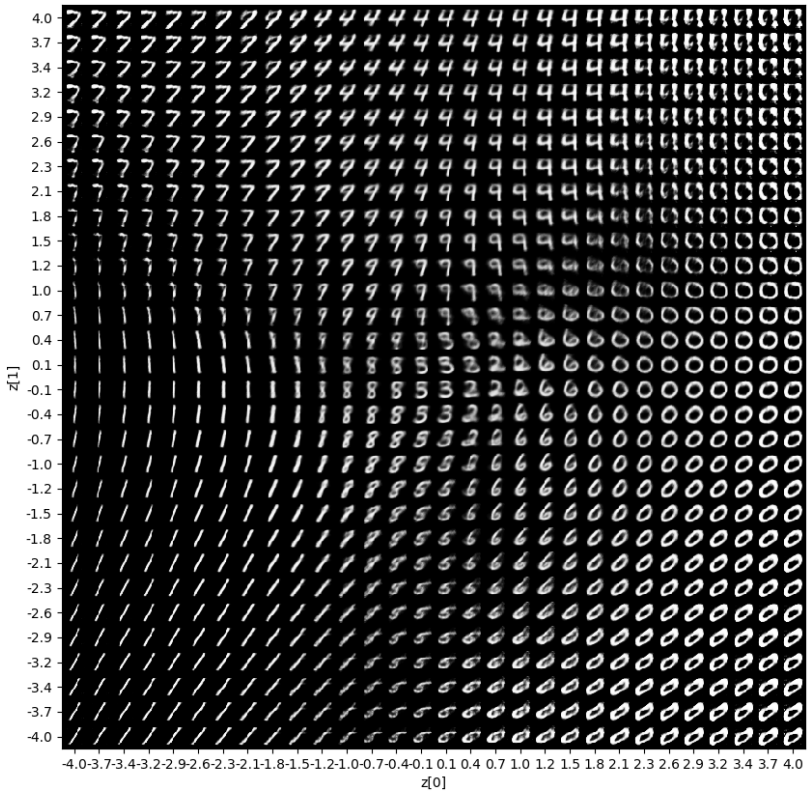

Figure B-1: **Digits generated from sampling a grid over the VAE latent space for MNIST images.**

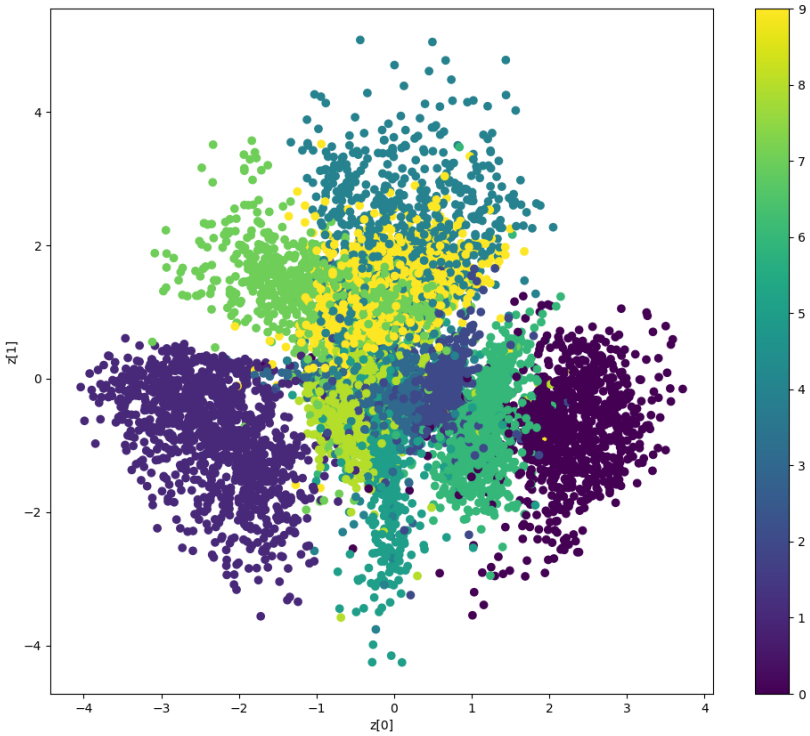

Figure B-2: **Means corresponding to MNIST digits in the VAE latent space.**

Bibliography

[1] Giora Alexandron, José A. Ruipérez-Valiente, Zhongzhou Chen, Pedro J. Muñoz-Merino, and David E. Pritchard. Copying@scale: Using harvesting accounts for collecting correct answers in a mooc. *Computers and Education*, 108:96 – 114, 2017.

[2] Joseph E. Beck and Yue Gong. Wheel-spinning: Students who fail to master a skill. In H. Chad Lane, Kalina Yacef, Jack Mostow, and Philip Pavlik, editors, *Artificial Intelligence in Education*, pages 431–440, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] François Chollet et al. Keras. https://keras.io, 2015.

[4] Jennifer DeBoer, Glenda Stump, Lori Breslow, and Daniel Seaton. Diversity in mooc students' backgrounds and behaviors in relationship to performance in 6.002x. In *Proceedings of the Sixth International Conference of MIT's Learning International Networks Consortium (LINC)*, 06 2013.

[5] Nicholas Diana, Michael Eagle, John C Stamper, and Kenneth R Koedinger. Extracting measures of active learning and student self-regulated learning strategies from mooc data. In *EDM*, pages 583–584, 2016.

[6] Carl Doersch. Tutorial on variational autoencoders, 2016. cite arxiv:1606.05908.

[7] Myse Elmadani, Moffat Mathews, and Antonija Mitrovic. Data-driven misconception discovery in constraint-based intelligent tutoring systems. In *Workshop Proceedings of the 20th International Conference on Computers in Education*, 11 2012.

[8] Ying Fang, Yonghong Jade Xu, Benjamin Nye, Arthur Graesser, Philip Pavlik, and Xiangen Hu. Online Learning Persistence and Academic Achievement. In *Proceedings of Educational Data Mining (EDM) 2017*, pages 312 – 317, Wuhan, China, June 2017. EDM 2017.

[9] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 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, pages 86–93, New York, NY, USA, 2010. ACM.

[10] Shimin Kai, Cristina Heffernan, Ma. Victoria Almeda, Neil Heffernan, and Ryan Baker. Decision tree modeling of wheel-spinning and productive persistence in skill builders. *Journal of Educational Data Mining*, 05 2018.

[11] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[12] René F. Kizilcec, Chris Piech, and Emily Schneider. Deconstructing disengagement: Analyzing learner subpopulations in massive open online courses. In *Proceedings of the Third International Conference on Learning Analytics and Knowledge*, LAK '13, pages 170–179, New York, NY, USA, 2013. ACM.

[13] Kenneth R Koedinger, Jihee Kim, Julianna Zhuxin Jia, Elizabeth A McLaughlin, and Norman L Bier. Learning is not a spectator sport: Doing is better than watching for learning from a MOOC. In *Proceedings of the second (2015) ACM conference on Learning @ Scale*, pages 111–120. ACM, 2015.

[14] Kenneth R Koedinger, Elizabeth A McLaughlin, Julianna Zhuxin Jia, and Norman L Bier. Is the doer effect a causal relationship?: how can we tell and why it's important. In *Proceedings of the Sixth International Conference on Learning Analytics & Knowledge*, pages 388–397. ACM, 2016.

[15] Nane Kratzke. Smart like a fox: How clever students trick dumb automated programming assignment assessment systems, 01 2019.

[16] Michael J Lee and Andrew J Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, pages 109–116. ACM, 2011.

[17] Caroline M. Hoxby. The economics of online postsecondary education: Moocs, nonselective education, and highly selective education. *The American Economic Review*, 104, 05 2014.

[18] Jitesh Maiyuran. Understanding the doer effect for computational subjects with moocs. Master's thesis, Massachusetts Institute of Technology, 77 Massachusetts Avenue, 6 2018.

[19] Jitesh Maiyuran, Ayesha Bajwa, Ana Bell, Erik Hemberg, and Una-May O'Reilly. Confirming and refining the doer effect in computational thinking MOOCs. In submission.

[20] Joshua J. Michalenko, Andrew S. Lan, and Richard G. Baraniuk. Data-mining textual responses to uncover misconception patterns. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 245–248, New York, NY, USA, 2017. ACM.

[21] Pedro J. Muñoz Merino, José A. Ruipérez Valiente, and Carlos Delgado Kloos. Inferring higher level learning information from low level data for the khan academy platform. In *Proceedings of the Third International Conference on Learning Analytics and Knowledge*, LAK '13, pages 112–116, New York, NY, USA, 2013. ACM.

[22] Curtis G. Northcutt, Andrew D. Ho, and Isaac L. Chuang. Detecting and preventing "multiple-account" cheating in massive open online courses. *Comput. Educ.*, 100(C):71–80, September 2016.

[23] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[24] Grzegorz Sedek and Mirosław Kofta. When cognitive exertion does not yield cognitive gain: Toward an informational explanation of learned helplessness. *Journal of personality and social psychology*, 58:729–43, 05 1990.

[25] Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1887–1896, New York, NY, USA, 2014. ACM.

[26] Shu-Fen Tseng, Yen-Wei Tsao, Liang-Chih Yu, Chien-Lung Chan, and K. Robert Lai. Who will pass? analyzing learner behaviors in MOOCs. *Research and Practice in Technology Enhanced Learning*, 11(1), apr 2016.

[27] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Inf. Softw. Technol.*, 55(6):1004–1016, June 2013.

[28] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. Learning to represent student knowledge on programming exercises using deep learning. In *EDM*, 2017.

[29] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Inf. Softw. Technol.*, 49(2):99–107, February 2007.

[30] Julia Wilkowski, Amit Deutsch, and Daniel M. Russell. Student skill and goal achievement in the mapping with google MOOC. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, pages 3–10, New York, NY, USA, 2014. ACM.

[31] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006.