Interlude: Assessment

Part of our skepticism about mechanized models of assessment is a basic, yet overlooked, fact: simple code can reinforce existing systemic inequities. Here we offer actual, runnable Python code (targeted at programming novices) that implements an adaptive assessment for a grade-school mathematics student. The way to wade through this code—regardless of your programming proficiency—is to notice how basic, core decisions (that a programmer might not even consider!) will generate inequitable outcomes for students. Assessment inequity does not require malevolence; we'd hazard that almost everyone in the field of education means well. Though building an adaptive assessment may sound complicated, it does not have to be. It is our hope that even nonprogrammers will be able to read the following program code and see points at which good intentions generate problematic situations.

Matthew teaches a class on computational research in education: an adaptive assessment like the ones here can be built by a relatively novice student for a week's homework. Like everything in the book, building an assessment in which bias is exposed and which leverages or at least considers notions of equity in the structure of the assessment is not necessarily harder. Let's say you start with a simple math problem. In Canvas (a system used in classes at both Stanford

and UW–Madison), instructors with no programming background can create a problem generator. The following text offers such a tool in Python for a basic arithmetic class (if this sample looks intimidating, the language that follows "#" is a step-by-step guide explaining what each line does and how).

```
# we will use randomness in this code, so we need to "import"
# what's called a "library" - you can look up any number of these
# at python.org - there are hundreds.

import random

# def declares a function in python
# a function is something that can be called repeatedly
# calling addend() returns an integer between 0 and 9

def simple_addend():
    # random.choice() takes a list and returns a random element
    # range(N) returns a list of integers from 0 to N
    return random.choice(range(10))
x = simple_addend() # random integer, eg 5
y = simple_addend() # random integer, eg 7

# int() takes a string (text) or number (such as 3.442)
#    and tries to give you the integer of that
#    for example, int(2.332) == 2
# input("hello!") prints that text (here: hello!)
#    and then waits for user input
# int(input(text)) turns the user input into
#    its best guess of an integer
```

Comments start with a #. Anything on the line after a "#" is not run by the computer. It is only for humans reading the code.

A *function* can be run repeatedly. Typically, just as in mathematics (e.g., f(x) = 2x), a function takes *parameters* and *returns a value* (in this example, x is the parameter in f(x) = 2x and, say, 6 would be "returned" when x is 3).

Python has built-in functions—such as int(), which we describe to the left. You can also write your own—such as "simple_addend()" just above.

```
response = int(input('\n{} + {} =?\n'.format(x,y)))
if response == (x+y):
   print("correct")
else:
   print("incorrect")
```

```
>>> 4 + 7 =?
user> 11
>>> correct
>>> 9 + 8 =?
user> 17
>>> correct
```

This is what it looks like when you "run" the code above. The running of the code generates the language signified by >>>; user input is denoted *user>*; and Python's responses are denoted >>>. You can run this code at the links included with this book or at the website (nothing is required other than a browser; nothing is installed).

If we add a "difficulty" parameter, we can make the math problems randomly harder or easier.

```
# this time, addend() takes a parameter, difficulty
def addend(difficulty):
   # we never want difficulty below 1,
   # so we always take the maximum of
   # the two options: difficulty and 1
   difficulty = max(1,difficulty)

   # we multiply difficulty by our random
   # integer, likely making it
   # a more difficult number to add
   return difficulty * random.choice(range(10))
```

Here we are defining a function we have named *addend* which takes a *parameter* that we have decided to call *difficulty*. When it is called, it generates a random number that is simply the parameter *difficulty* times a random number that is either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. (The first line of code after the comment just limits the difficulty to 1 or larger.)

```
# difficulty starts at 3
# since our range above is 0–9, addends now
# range from 0–27
difficulty = 3
x = addend(difficulty)
y = addend(difficulty)

response = int(input('\n{} + {} =?\n'.format(x,y)))

if response == (x+y):
   # if correct, we make things harder for next time by
   # adding one to the difficulty
   difficulty += 1
   print("correct")
else:
   # if incorrect, we make things easier for next time by
   # subtracting one from the difficulty
   difficulty -= 1
   print("incorrect")
print("difficulty is now {}.".format(difficulty))
```

We then generate two random numbers with *addend* (which are different every time the program is run) and that we have creatively and memorably named *x* and *y*.

We take *input* from the user, and we respond by acknowledging the response and updating the difficulty.

```
>>> 12 + 3 =?
user> 22
>>> incorrect
>>> difficulty is now 2.
>>> 21 + 6 =?
user> 32
>>> incorrect
>>> difficulty is now 2.
```

This is what it looks like when it is run. Again, *>>>* is what Python produces, and *user>* is what the user types.

That process is not particularly tricky. The process becomes something only slightly trickier when the problem generated has to be adaptive.

In some sense, we can call it a day. We have created an adaptive assessment. The more problems that a student gets right, the harder the following problems become, and the teacher gets a "max

difficulty" score. Not only that, but because the tool doesn't penalize you for getting a question wrong (instead making the next question is relatively easier), you are incentivized to keep going without having to worry about working on problems that may be a bit too hard. Sounds good, right? So, what is wrong with this?

The biggest sticking point is that very few problems scale in the way this generator is designed. The problems that do scale rarely assess understanding meaningfully. For that we need "features."

As we saw in many of the cases in this chapter, the decision of what counts as a *feature* (e.g., a measurable aspect, an axis, a column, a dimension) is extremely fraught. As we will see through the following code examples, "feature engineering" is a salient way that bias and control are baked into the algorithms that shape modern schooling. Despite the cutting edge-allure of AI, the cleaning and quantizing of data is a job that is tedious, tiresome, and relegated to the status of rote labor. Much of the feature engineering on massively successful machine learning in the marketplace is done by novices and interns.

Going back to our example again, we create something simple but include two features:

```
def addend(difficulty_range):
    # random.choice() takes a list and returns a random element
    # range(N) returns a list of integers from 0 to N
    return difficulty_range * random.choice(range(10))
```

The *addend* function is the same, but now instead of varying the difficulty only through the number, we are adding a new function we have called *arithmetic_operation* that also takes the *difficulty* parameter. We are proposing that subtraction is inherently more difficult than addition and that we can generate a tricky two-feature problem with this same *difficulty* parameter.

```
# this function returns a function (like in math)
def arithmetic_operation(difficulty):
    if difficulty < 3:
        # at low difficulty, addition
        return ("+",lambda x,y: x+y)
    if difficulty < 5:
        # at higher difficulty, subtraction
        return ("-",lambda x,y: x-y)
    else:
        # highest difficulty is multiplication
        return ("*",lambda x,y: x*y)

def two_feature_problem(difficulty):
    x = addend(difficulty)
    y = addend(difficulty)

    # calling arithmetic_operation() returns the name
    # so we can print it out to the user
    # and the function, so we can do the calculation
    # ourselves to check their answer
    op_name, op_function = arithmetic_operation(difficulty)
    response = int(input('{} {} {} =?'.format(x,op_name,y)))
    if response == op_function(x,y):
        difficulty += 1
        print("correct")
    else:
        difficulty -= 1
        print("incorrect")
    print("difficulty is now {}.".format(difficulty))

# we have changed the difficulty
# so we need to tell the program
# outside the function the new difficulty
return difficulty
```

```
difficulty = 4

# do this 3 times
for i in range(3):
    # since this changes every time, we set it equal
    # to the returned value of the
    # two_feature_problem function
    print() # separate the lines
    difficulty = two_feature_problem(difficulty)
```

```
>>> 0 - 28 =?
user> -28
>>> correct
>>> difficulty is now 5.
>>> 25 * 30 =?
user> 3
>>> incorrect
>>> difficulty is now 4.
>>> 36 - 24 =?
user> 12
>>> correct
>>> difficulty is now 5.
>>> 12 - 4 =?
user> 3
>>> incorrect
>>> difficulty is now 3.
>>> 15 - 9 =?
user> 3
>>> incorrect
>>> difficulty is now 2.
>>> 12 + 8 =?
user> 0
>>> incorrect
>>> difficulty is now 1.
```

We can see here that the problems become meaningfully more difficult as you succeed at them and easier when you fail. For the example, we input that 25 * 30 is 3, but we have calculators and know that it is secretly an entirely different number. Sadly, that number, like so many, is unknowable and/or 750.

These features introduce a new, more fundamental question: Should operation and range scale difficulty in the same way? This short code segment masks the equity implications of the scaling: different students will excel at different trajectories through these problems. For some students, addition of negative numbers will halt their progress; others might stop at the multiplication of multidigit numbers. It may be that there is a systematicity to these different trajectories. Considering the ways these would play out in the real world, it is incumbent on programmers to account for different trajectories.

Teachers use tools every day that carry inherent assumptions or biases, and, because these assumptions might be buried within the implicit logic of simple code (e.g., like the preceding code), they go unexamined. It's not even clear that anyone involved with many of the systems would be aware of the need to look for systematic problems of this sort. Raising awareness that even simple code like this contains inherent systematic inequity (often residing in the IH quadrant of AnSpec) is not enough. One possible best practice would be to agree that adaptive software offers some way to expose assumptions. Although it does not "fix" the inequality baked into an assessment, it at least makes such inequality obvious. Unfortunately, this either requires the user (in this case, teachers who already have plenty of other demands on their schedules) to dedicate their time to learning how to do critical analysis of code, or requires us to create some sort of language for exposing and expressing these biases in ways that do not require teachers get another master's degree. Turning the preceding code into an assessment only requires keeping track of the assessment implicit in the adaptivity:

```
def analyze_incorrect(user_answer, x1, x2, op_function):
    possible_operations = [lambda x,y: x+y,
                           lambda x,y: x-y,
                           lambda x,y: x*y]
    correct_answer = op_function(x1,x2)
```

```
    # if they simply flipped the order or used another operation
    # lower the difficulty of the operation
    if user_answer in map(lambda op: op(x1,x2), possible_operations) or
    user_answer in map(lambda op: op(x2,x1), possible_operations):
        print("are you sure you did the right operation?")
        return (0,-1)
    # don't change range, make operation less difficult
    # or if they were within 10% of the correct answer
    # then change the range, but don't change the operation
    elif abs(user_answer - correct_answer) <= abs(0.1 * correct_answer):
        print("are you sure you did the calculation correctly?")
        return (-1,0)
    # otherwise, no guesses! difficulty for both is lowered by one
    else:
        return (-1,-1)

difficulty_range = 3
difficulty_operation = 3
max_range_difficulty = 0
max_operation_difficulty = 0

for i in range(5):
    x = addend(difficulty_range)
    y = addend(difficulty_range)
    op_name,op_function = arithmetic_operation(difficulty_operation)
    response = int(input('\n{} {} {} =?\n'.format(x,op_name,y)))
    if response == op_function(x,y):
        difficulty_range += 1
        difficulty_operation += 1
        print("correct")
    else:
        delta_difficulty_range, delta_difficulty_operation =
        analyze_incorrect(response,x,y,op_function)
        difficulty_range += delta_difficulty_range
        difficulty_operation -= delta_difficulty_operation
        print("incorrect")
    print("range difficulty is now {}.".format(difficulty_range))
    print("operation difficulty is now {}.".format(difficulty_operation))
```

```
    max_range_difficulty = max(max_range_difficulty,difficulty_range)
    max_operation_difficulty = max(max_operation_difficulty,difficulty_operation)
print("user score (max range difficulty): {}".format(max_range_difficulty))
print("user score (max operation difficulty): {}".format(max_operation_difficulty))

# ** a sample run **
>>> 24 - 3 =?
user> 21
>>> correct
>>> range difficulty is now 4.
>>> operation difficulty is now 4.
>>> 32 - 20 =?
user> 12
>>> correct
>>> range difficulty is now 5.
>>> operation difficulty is now 5.
>>> 25 * 35 =?
user> 875
>>> correct
>>> range difficulty is now 6.
>>> operation difficulty is now 6.
>>> 0 * 48 =?
user> 0
>>> correct
>>> range difficulty is now 7.
>>> operation difficulty is now 7.
>>> 21 * 21 =?
user> 2
>>> incorrect
>>> range difficulty is now 6.
>>> operation difficulty is now 8.
>>> user score (max range difficulty): 7
>>> user score (max operation difficulty): 8
>>> 21 - 3 =?
user> 0
>>> incorrect
```

```
>>> range difficulty is now 2.
>>> operation difficulty is now 4.
>>> 12 - 16 =?
user> 0
>>> incorrect
>>> range difficulty is now 1.
>>> operation difficulty is now 5.
>>> 9 * 1 =?
user> 0
>>> incorrect
>>> range difficulty is now 0.
>>> operation difficulty is now 6.
>>> 0 * 0 =?
user> 0
>>> correct
>>> range difficulty is now 1.
>>> operation difficulty is now 7.
>>> 3 * 9 =?
user> 0
>>> incorrect
>>> range difficulty is now 0.
>>> operation difficulty is now 8.
>>> user score (max range difficulty): 2
>>> user score (max operation difficulty): 8
```

We can see how quickly our adaptive math tutor elides into a sort of de facto (if not de jure) assessment. On the one hand, this assessment is in many ways superior to a classic math test: it does not punish failure, it allows exploration, it is impossible to cheat the system (as the numbers are randomized), and it does not report your trajectory, so to speak. On the other hand, it may make implicit what is understood to be explicit about a test: that it is an evaluation, that it is not expected to relate to lived practice, that it is an exercise for assessment rather than learning. Unless that distinction between the two is

understood by all parties, it is simply another potential opportunity to rank students unnecessarily. There is no expectation that, simply because it is adaptive, it is authentic to any meaningful tasks. That is a dangerous leap, but one that many people seem to make. We may just be reproducing many of the worst elements of school, faster. And yet, here we are splashing in the possibilities of Python. Perhaps we might produce new school elements, generate new pathways for learning, support, and assessment. What might you tweak in your work with code and with the contexts of your scholarship?

The Left Hand of Data

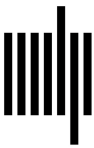
Designing Education Data for Justice

By: Matthew Berland, Antero Garcia

Citation:

The Left Hand of Data: Designing Education Data for Justice
By: Matthew Berland, Antero Garcia
DOI: 10.7551/mitpress/14381.001.0001
ISBN (electronic): 9780262377645
Publisher: The MIT Press
Published: 2024

The open access edition of this book was made possible by generous funding and support from MIT Press Direct to Open



The MIT Press

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Stone Serif and Stone Sans by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data

Names: Berland, Matthew, author. | Garcia, Antero, author.
Title: The left hand of data : designing education data for justice / Matthew Berland and Antero Garcia.
Description: Cambridge, Massachusetts : The MIT Press, [2024] | Includes bibliographical references and index.
Identifiers: LCCN 2023030088 (print) | LCCN 2023030089 (ebook) | ISBN 9780262547529 (paperback) | ISBN 9780262377652 (epub) | ISBN 9780262377645 (pdf)
Subjects: LCSH: Education—Data processing. | Education—Research. | Educational evaluation.
Classification: LCC LB1028.43 .B45 2024 (print) | LCC LB1028.43 (ebook) | DDC 370.285—dc23/eng/20230718
LC record available at https://lccn.loc.gov/2023030088
LC ebook record available at https://lccn.loc.gov/2023030089