Introduction to Algorithms and Machine Learning

from Sorting to Strategic Agents

Justin Skycak

First edition.

www.justinmath.com

Preamble: The Story of Math Academy's Eurisko Sequence

This book was written to support Eurisko, an advanced math and computer science elective course sequence within the Math Academy program at Pasadena High School. During its operation from 2020 to 2023, Eurisko was the most advanced high school math/CS sequence in the USA. ("Eurisko" is Greek for "I discover," and is the namesake of an AI system from the 1980s that won a particular game competition twice in a row, even when the rules were changed in an attempt to handicap it.)

Eurisko's courses were presented at a level of intensity comparable to those offered at elite technical universities, and students wrote all their code from scratch before they were allowed to import external libraries. This was possible because students had already learned a large amount of college-level math through Math Academy, including multivariable calculus, linear algebra, and differential equations by the end of 10th grade.

The first Eurisko course was inspired by MIT's Introduction to Computer Science and went far beyond it. In addition to implementing canonical data structures and algorithms (sorting, searching, graph traversals), students wrote their own machine learning algorithms from scratch (polynomial and logistic regression, k-nearest neighbors, k-means clustering, parameter fitting via gradient descent).

In subsequent courses, students implemented more advanced machine learning algorithms such as decision trees and neural networks. They also reproduced academic research papers in artificial intelligence leading up to Blondie24, an AI computer program that taught itself to play checkers. At the same time, they worked together to implement Space Empires, an extremely complex board game that pushed their large-scale project skills (object-oriented design, version control, etc.) to the limit. The ultimate goal was to create artificially intelligent Space Empires players, drawing inspiration from techniques used in Blondie24.

This book would not have been written if the Eurisko program had not existed, and Eurisko would not have been possible without the collaboration of Jason Roberts, founder of Math Academy. Eurisko started in June 2020 when Jason asked me to teach his 15-year-old son Colby some serious computer science over the summer. He pulled in some of Colby's classmates who had the necessary mathematical background, and we put together a summer computer science group that met three times a week with about 10 hours of problem sets each week.

All problem sets required students to write code individually, from scratch, in Python. They weren't allowed to use external libraries.

Instead, they had to build everything themselves. They were allowed to collaborate at a high level, discussing different approaches to solving the problems, but every student had to write up every problem set on their own.

To our surprise, the students progressed even faster than we could have possibly expected:

- At the start of June, they didn't know how to write helper functions. Even something as simple as checking if a string was a palindrome was not trivial to them.
- By the start of July, they had built a matrix class and a gradient descent optimizer from scratch. The matrix class included methods for matrix arithmetic as well as standard linear algebra procedures like row reduction, determinants, and inverses.
- By the start of August, they had built a regression library on top of their matrix class and gradient descent optimizer. The library included polynomial, logistic, and multiple linear regressors with interaction terms.
- On top of all this, they also implemented standard algorithms for sorting arrays and traversing graphs. And per Jason's suggestion, to get some systems programming experience, they also created a simple version of the Space Empires board game and played against each other by programming and submitting autonomous strategies.

At the end of the summer, Eurisko was fortunate to receive funding for an official high school class through a partnership with App

Academy. Jason recruited a second cohort of incoming 10th graders, and – through an extreme feat of class management – I managed to continue supporting the first cohort while simultaneously launching the second cohort within the same class period.

The class setup during the 2020-21 year was incredibly tricky. School was fully remote due to the pandemic, and each cohort only had two hours of class time over a group video call each week. If a student got stuck, they had to ask for help by posting a message on Slack, and the message had to be descriptive enough that I or another classmate could quickly understand their question and respond briefly without burning much time. This setup would be a steep learning curve for any junior developer, much less a high school underclassman with little to no prior coding experience – but we made do, and those students who muscled through it emerged with an incredible amount of self-sufficiency and debugging ability. (Not to mention, we did all this on school-issued Chromebooks using free online development environments like Replit and Gitpod.)

Later years became significantly smoother as I refined the curriculum and school returned to in-person classes. During the 2021-22 year, most problem sets contained written tutorials descriptive enough that I only had to give a traditional lesson once per week. Over summer 2022 I further refined and organized these tutorials into book chapters, and during the 2022-23 year, the Eurisko classes were almost entirely self-service. Students read the assigned chapter on their own and completed practice problems. In class, instead of giving a traditional lesson, I answered questions and helped students debug their code.

In 2021, Jason shared a wild idea to have Eurisko students reproduce the Blondie24 research papers and use that as inspiration to create artificially intelligent Space Empires players. This became our vision for the ultimate capstone project. The first Eurisko cohort came within striking distance but ultimately ran out of time because they only had two full years of Eurisko instead of three. But the second cohort managed to reproduce one to two of the three papers in the Blondie24 research program and use this as inspiration for evolving combat strategies within Space Empires.

The second cohort's final year (2022-23) also happened to be the final year of the Eurisko program due to my relocation. All together, there were 16 students who stayed for the duration of the program.

- *Cohort 1 (Summer 2020 - Spring 2022):* David Gieselman, George Meza, Riley Paddock, Colby Roberts, Elijah Tarr
- *Cohort 2 (Fall 2020 - Spring 2023):* Maia Dimas, Justin Hong, Cayden Lau, Anton Perez, William Wallius, Charlie Weinberger
- *Cohort 3 (Fall 2021 - Spring 2023):* Celeste Acosta, Elias Gee, Benjamin Park, Jeffrey Smithwick
- *Cohort 4 (Fall 2022 - Spring 2023):* Matteo Paz

Finally, a sincere thank you to Sanjana Kulkarni for her thoughtful suggestions and diligent proofreading of this book.

Contents

Part I

Hello World

1. Some Short Introductory Coding Exercises

Let's get started on some introductory coding exercises. It's assumed that you've had some basic exposure to programming and that you know about variables, functions, if statements, for loops, while statements, arrays, strings, and dictionaries. Ideally, you've learned some object-oriented programming (classes, attributes, methods) as well. We will speak in terms of Python, but the general ideas are applicable to any programming language.

It's important to understand that in coding, you should always try to do as much as you can on your own, using Google as a resource if needed. Don't know what a word means? Google it. Don't know how to run a file from the command line? Google it. Don't know how to get the second character of a string? Google it. Our goal is to provide just enough scaffolding to give you direction while avoiding hand-holding.

Exercises

Write the following functions from scratch. Don't use any external libraries or any built-in functions that allow you to bypass the use of for loops, while loops, or if statements in nontrivial ways. In particular, don't use `Set` or `Counter`. But you can use primitives like `len()` and `Array.append`.

1. `check_if_symmetric(string)`
   Return `True` if the input string is symmetric (i.e. a palindrome), and `False` if not. You can ignore capitalization.

2. `convert_to_numbers(string)`
   Return an array of numbers corresponding to letters in a string where space = 0, a = 1, b = 2, and so on. For example, `convert_to_numbers('a cat')` should return `[1,0,3,1,20]`.

3. `convert_to_letters(string)`
   This is the inverse of `convert_to_numbers`. For example, `convert_to_letters([1,0,3,1,20])` should return `'a cat'`.

4. `get_intersection(array1, array2)`
   Return an array consisting of the elements that are in both `array1` and `array2`. There should not be any repeated elements in the output array.

5. `get_union(array1, array2)`
   Return an array consisting of the elements that are in either

`array1` or `array2`. Again, there should not be any repeated elements in the output array.

6. `count_characters(string)`
   Count the number of each character in a string and return the counts in a dictionary. Lowercase and uppercase letters should not be treated differently. For example, `count_characters('A cat!!!')` should return the dictionary `{ 'a': 2, 'c': 1, 't': 1, ' ': 1, '!': 3 }`.

7. `is_prime(N)`
   Check whether an input integer $N > 1$ is prime by checking whether there exists some $n \in \{2, 3, 4, \ldots, \lfloor N/2 \rfloor\}$ such that n divides N. (The $\lfloor\ \rfloor$ symbol denotes the *floor function*.)

Tests

In another file, write a variety of tests for each function. Two example tests are shown below for `check_if_symmetric`. You can use these, but you should also write more tests that cover all the edge-cases you can think of. (For example, an empty string is symmetric, and the string `'!ab123 4 321ba!'` is also symmetric.)

```
from hello_world import check_if_symmetric

tests = [
    {
        'function': check_if_symmetric,
        'input': 'racecar',
        'output': True
    },
    {
        'function': check_if_symmetric,
        'input': 'batman',
        'output': False
    }
]

num_successes = 0
num_failures = 0

for test in tests:
    function = test['function']
    test_input = test['input']
    desired_output = test['output']
    actual_output = function(test_input)

    if actual_output == desired_output:
        num_successes += 1
    else:
        num_failures += 1
        function_name = function.__name__
        print('')
        print(f'{function_name} failed on input {
            test_input}');
        print(f'\tActual output: {actual_output}')
        print(f'\tDesired output: {desired_output}')

print(f'Testing complete: {num_successes} successes and
    {num_failures} failures.')
```

Debugging

Run your tests and fix any failures. If you struggle with anything, *don't* ask others for help until you've made a thorough effort to debug the issue on your own. Here are some debugging tips:

1. *Print out everything.* Within the function that you're debugging, print out every manipulation that your code makes, even if you don't think it's making a mistake there. Bugs often show up in places you don't expect. Also, don't just print out the values of variables – it will help you avoid confusion if you print the names of the variables as well.

2. *Identify the first discrepancy.* Manually work out what the printouts should be, and then look for the first instance where the actual printouts deviate from what you're expecting.

3. *If the discrepancy involves a helper function, then isolate the issue in a separate file and return to step* 1. If the issue is occurring when you're passing inputs into a helper function, then create a separate file where all you do is pass those inputs into the helper function, and go back to step 1 (print out everything) with the helper function. This way, you can focus entirely on the helper function without worrying about any other pieces of code interacting with it.

4. *If the discrepancy does not involve a helper function and you still can't figure out what's going wrong, then ask for help.* Be sure that you can explain what you've done to debug the issue so far, including the furthest point back to which you've traced the

bug. Keep your debugging print statements in your function and make sure the printouts are organized so that another person can follow along with them without knowing all the nitty-gritty details of your code.

Code Quality

Your code should be clean and work in general. If you have thorough test coverage, you can be fairly confident that your code works in general. To be confident that your code is clean, do the following:

- *Run your code through a linter* (like pep8online) and fix all the issues.
- *Use proper cases.* In Python, variables, functions, and files use `snake_case` (all lowercase with underscores separating words), while classes use `PascalCase` (no spaces with each separate word capitalized).
- *Make sure that your variable names are clear and appropriate.* Variables and classes should be nouns, while functions (including methods) should be verbs. Additionally, names should be descriptive. It's okay to make a name multiple words long if you need. For example, `calc_prob()` is much better than `prob()` or `cp()`.

2. Converting Between Binary, Decimal, and Hexadecimal

We are used to representing numbers using ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This is called the **decimal** number system.

However, there are other number systems that use more or fewer characters. For example, the **binary** or **base-**2 number system uses two characters (0 and 1), and the hexadecimal or **base-**16 number system uses sixteen characters (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, where $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, $F = 15$).

Counting Demonstration

Below is a table that illustrates how to count from zero to thirty-two in these different number systems. Each column corresponds to a different number system, and each row shows how the same quantity is represented in the three different number systems.

The key pattern to notice is that you always increment the character in the rightmost digit, unless it has reached the last character. In that case, you replace it with the first character and then increment the digit directly to the left.

Decimal	Binary	Hexadecimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Decimal	Binary	Hexadecimal
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F
32	100000	20

Converting from Base-b to Decimal

In general, if you have a number consisting of digits $x_n x_{n-1} \ldots x_1 x_0$ in base-b, then you can convert it to decimal using the following formula:

$$x_n \cdot b^n + x_{n-1} \cdot b^{n-1} + \ldots + x_1 \cdot b^1 + x_0 \cdot b^0$$

For instance, the number 11010 in binary (base-2) corresponds to the following number in decimal:

$$\begin{aligned}
&1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= 26
\end{aligned}$$

Likewise, the number 3B07F in hexadecimal (base-16) corresponds to the following number in decimal:

$$\begin{aligned}
&3 \cdot 16^4 + \text{B} \cdot 16^3 + 0 \cdot 16^2 + 7 \cdot 16^1 + \text{F} \cdot 16^0 \\
&3 \cdot 16^4 + 11 \cdot 16^3 + 0 \cdot 16^2 + 7 \cdot 16^1 + 15 \cdot 16^0 \\
&= 241791
\end{aligned}$$

Converting from Decimal to Base-b

On the other hand, if you have a decimal number and you want to convert it to base-b, then you can repeatedly compute the highest power of b (call it b^n) that's less than or equal to your decimal number, and then subtract off the largest multiple of b^n that's less than or equal to your decimal number.

For example, to convert the decimal number 26 to binary (base-2), we do the following:

1. The largest power of 2 that's less than or equal to 26 is $2^4 = 16$. The largest multiple of 16 that's less than or equal to 26 is $1 \cdot 16 = 16$. Subtracting off, we have $26 - 16 = 10$.

2. The largest power of 2 that's less than or equal to 10 is $2^3 = 8$. The largest multiple of 8 that's less than or equal to 10 is $1 \cdot 8 = 8$. Subtracting off, we have $10 - 8 = 2$.

3. The largest power of 2 that's less than or equal to 2 is $2^1 = 2$. The largest multiple of 2 that's less than or equal to 2 is $1 \cdot 2 = 2$. Subtracting off, we have $2 - 2 = 0$.

4. We subtracted off $1 \cdot 2^4$, $1 \cdot 2^3$, and $1 \cdot 2^1$. Therefore, We can write 26 as $1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, which has coefficients $1, 1, 0, 1, 0$ and therefore corresponds to the binary number 11010.

Likewise, to convert the decimal number 241791 to hexadecimal (base-16), we do the following:

1. The largest power of 16 that's less than or equal to 241791 is $16^4 = 65536$. The largest multiple of 65536 that's less than or equal to 241791 is $3 \cdot 65536 = 196608$. Subtracting off, we have $241791 - 196608 = 45183$.

2. The largest power of 16 that's less than or equal to 45183 is $16^3 = 4096$. The largest multiple of 4096 that's less than or equal to 45183 is $11 \cdot 4096 = 45056$. Subtracting off, we have $45183 - 45056 = 127$.

3. The largest power of 16 that's less than or equal to 127 is $16^1 = 16$. The largest multiple of 16 that's less than or equal to 127 is $7 \cdot 16 = 112$. Subtracting off, we have $127 - 112 = 15$.

4. We subtracted off $3 \cdot 16^4$, $11 \cdot 16^3$, $7 \cdot 16^1$, and $15 \cdot 16^0$. Therefore, We can write 241791 as $3 \cdot 16^4 + 11 \cdot 16^3 + 0 \cdot 16^2 + 7 \cdot 16^1$

$+15 \cdot 16^0$, which has coefficients 3, 11, 0, 7, 15 and therefore corresponds to the hexadecimal number 3B07F.

Exercises

Write the following functions. You can use string representations for all your inputs and outputs. As always, write a handful of tests for each function.

1. `binary_to_decimal(string)`
   Take a binary representation of a number and return the decimal representation. For example, `binary_to_decimal('11010')` should return `'26'`.
2. `hexadecimal_to_decimal(string)`
   Take a hexadecimal representation of a number and return the decimal representation.
3. `decimal_to_binary(string)`
   Take a decimal representation of a number and return the binary representation.
4. `decimal_to_hexadecimal(string)`
   Take a decimal representation of a number and return the hexadecimal representation.
5. `binary_to_hexadecimal(string)`
   Take a binary representation of a number and return the hexadecimal representation. This is easy once you've written the

functions above: just convert from binary to decimal, and then from decimal to hexadecimal.

6. `hexadecimal_to_binary(string)`
   Take a hexadecimal representation of a number and return the binary representation. This is easy once you've written the functions above: just convert from hexadecimal to decimal, and then from decimal to binary.

3. Recursive Sequences

A **recursive sequence** is a sequence where each term is a function of the previous terms.

For example, consider the following rule for generating a recursive sequence: starting with 3, generate each term by doubling the previous term and adding 1. The terms of this sequence are as follows:

$$3, 7, 15, 31, 63, 127, 255, 511, \ldots$$

Implementing Recursive Sequences

One way to implement recursive sequences in code is to store all terms in an array. For example, a function that returns the first n terms of the recursive sequence "starting with 3, generate each term by doubling the previous term and adding 1" could be implemented as follows:

```
def calc_first_n_terms(n):
    terms = [3]

    while len(terms) < n:
        prev_term = terms[-1]
        next_term = 2 * prev_term + 1
        terms.append(next_term)

    return terms
```

If all we want is the nth term, then it can sometimes be more convenient to make the implementation itself recursive. This way, we don't have to bother storing any intermediate values.

```
def calc_nth_term(n):
    if n == 1:
        return 3
    else:
        prev_term = calc_nth_term(n-1)
        return 2 * prev_term + 1
```

To understand how the function above works, first notice that `calc_nth_term(1)` will return 3. This is called the **base case**. If a different input is passed, then the function will keep calling itself in a lower input until it reaches the base case.

```
calc_nth_term(4)
|
|--> prev_term
     = calc_nth_term(3)
       |
       |--> prev_term
            = calc_nth_term(2)
              |
              |--> prev_term
                   = calc_nth_term(1)
                     |
                     |--> return 3
                          |
                     |<--
                     |
              |<-- return 2 * 3 + 1 = 7
              |
       |<-- return 2 * 7 + 1 = 15
       |
|<-- return 2 * 15 + 1 = 31

Output: 31
```

Exercises

Several different recursive sequences are described below. For each sequence, write a function to generate an array containing first n terms, and then write a separate *recursive* function to generate the nth term. Be sure to work these sequences out by hand and write tests.

1. Starting with 5, generate each term by multiplying the previous term by 3 and subtracting 4.

2. Starting with 25, generate each term by taking half of the previous term if it's even, or multiplying by 3 and adding 1 if it's odd. (This is an instance of a *Collatz sequence*.)

3. Starting with $0, 1$, generate each term by adding the previous two terms. (This is the famous *Fibonacci sequence*.)

4. Starting with $2, -3$, generate each term by adding the product of the previous two terms.

4. Simulating Coin Flips

It's often possible to estimate probabilities of events by simulating a large number of random experiments and measuring the proportion of trials in which the desired outcome occurred. This technique is known as **Monte Carlo simulation**, named after a famous casino town.

To simulate random experiments in code, it's common to use a random number generator and then map the output of the random number generator to an outcome of the experiment. For example, to simulate a coin flip, one could generate a random decimal r between 0 and 1 and apply the following function:

$$\text{outcome}(r) = \begin{cases} \text{heads} & \text{if } r < 0.5 \\ \text{tails} & \text{otherwise} \end{cases}$$

Exercises

Write a function `sim_probability(num_heads, num_flips)` that uses Monte Carlo simulation to compute the probability of getting a given number of heads in a given number of flips of a fair coin.

1. First, simulate a large number of trials (say, 1000). In each trial, flip a coin `num_flips` times and count how many heads appear. You may import a random number generator for this.
2. Then, count the number of trials in which exactly `num_heads` heads appeared and divide it by the total number of trials (1000).

To test your implementation, work out the result by hand for several combinations of `num_heads` and `num_flips`, and verify that your function consistently returns results that are close to these true values.

High-Level Sanity Checks

In this exercise, we tested the function by working out the probability by hand. However, Monte Carlo simulation is often used to estimate probabilities that are too complicated to work out by hand. In such cases, it's common to test the implementation by

- working out simple cases by hand when possible, and
- performing high-level sanity checks.

High-level sanity checks still involve identifying and verifying true statements, but these true statements do not need to refer to exact values.

To illustrate, let's brainstorm some characteristics about how the probability of getting a particular number of heads is related to the number of coin flips:

- The probabilities should form a distribution, i.e. they should be non-negative and add up to 1.
- This distribution should look somewhat like a bell curve, with the most likely outcome being that half of the coins land on heads and the other half on tails.
- Since landing on heads is just as likely as landing on tails, the distribution should be symmetric.

To verify these characteristics, you can choose some value of `num_flips`, compute the values

- `sim_probability(1, num_flips)`,
- `sim_probability(2, num_flips)`,
- ...,
- `sim_probability(num_heads, num_flips)`,

and then check the following:

- The values are all non-negative and their sum is approximately 1.
- The graph of probability vs `num_heads` looks like a symmetric bell curve.

Try doing this for several values of `num_flips`.

5. Roulette Wheel Selection

As we saw previously, it's easy to simulate a random coin flip by generating a random decimal r between 0 and 1 and applying the following function:

$$\text{outcome}(r) = \begin{cases} \text{heads} & \text{if } r < 0.5 \\ \text{tails} & \text{otherwise} \end{cases}$$

This is a special case of a more general idea: sampling from a discrete probability distribution. Flipping a fair coin is tantamount to sampling from the distribution `[0.5, 0.5]`, i.e. 0.5 probability heads and 0.5 probability tails.

More complicated contexts may require sampling from longer distributions that may or may not be uniform. For example, if we wish to simulate the outcome of rolling a die with two red faces, one blue face, one green face, and one yellow face, then we need to sample from the distribution `[0.4, 0.2, 0.2, 0.2]`.

Note that when we sample from the distribution, we need only sample an *index* from the distribution, and then use the index to look

up the desired value. For example, when we sample an index from the distribution `[0.4, 0.2, 0.2, 0.2]`, we have probabilities 0.4, 0.2, 0.2, and 0.2 of getting indices 0, 1, 2, and 3 respectively. Then, all we need to do is look up that index in the following array: `[red, blue, green, yellow]`.

Roulette Wheel Selection

Roulette wheel selection is an elegant technique for sampling an index from an arbitrary discrete probability distribution. It works as follows:

1. turn the distribution into a cumulative distribution,
2. sample a random number r between 0 and 1, and then
3. find the index of the first value in the cumulative distribution that is greater than or equal to r.

To illustrate, let's sample an index from the distribution `[0.4, 0.2, 0.2, 0.2]` that was mentioned above in the context of a die roll.

1. First, we construct the cumulative distribution: `[0.4, 0.4+0.2, 0.4+0.2+0.2, 0.4+0.2+0.2+0.2]`, or more simply, `[0.4, 0.6, 0.8, 1.0]`.
2. Then, we sample a random number r between 0 and 1. Suppose we get $r = 0.63$.

3. Finally, we find the index of the first value in the cumulative distribution that is greater than or equal to $r = 0.63$. In our case, this is the value 0.8 at index 2, because the next value (1.0 at index 3) is greater than $r = 0.63$.

So, we have sampled the index 2.

Exercise

Write a function `random_draw(distribution)` that samples a random number such that `distribution[i]` is the probability of sampling index `i`.

To test your function on a particular distribution, sample many indices from the distribution and ensure that the proportion of times each index gets sampled matches the corresponding probability in the distribution. Do this for a handful of different distributions.

6. Cartesian Product

Implementing the Cartesian product provides good practice working with arrays. Recall that the Cartesian product constructs all the points whose elements occupy the given ranges. To illustrate:

```
>>> calc_cartesian_product([
    ['a'],
    [1, 2, 3],
    ['Y', 'Z']
  ])

[
  ['a', 1, 'Y'],
  ['a', 1, 'Z'],
  ['a', 2, 'Y'],
  ['a', 2, 'Z'],
  ['a', 3, 'Y'],
  ['a', 3, 'Z']
]
```

Implementation

The Cartesian product can be implemented elegantly via the following procedure:

1. Create an array that will contain all the points in the cartesian product. Initialize it with a single empty point, i.e. `points = [[]]`.
2. Create a copy of `points`, and for each copied point, loop through the first range and create an extended point by appending a range item onto the copied point, create a handful of new points. Collect all these extended points and save them to `points`.
3. Repeat step 2 for each range.

Below is a worked example.

```
ranges = [
  ['a'],
  [1, 2, 3],
  ['Y', 'Z']
]

points: [
  []
]

looping through range ['a']:
 - item 'a' --> extended point ['a']

points: [
  ['a']
]

looping through range [1, 2, 3]:
 - item 1 --> extended point ['a', 1]
 - item 2 --> extended point ['a', 2]
 - item 3 --> extended point ['a', 3]

points: [
  ['a', 1],
  ['a', 2],
  ['a', 3]
]

looping through range ['Y', 'Z']:
 - item 'Y' --> extended points ['a', 1, 'Y'], ['a', 2,
     'Y'], ['a', 3, 'Y']
 - item 'Z' --> extended points ['a', 1, 'Z'], ['a', 2,
     'Z'], ['a', 3, 'Z']

points: [
  ['a', 1, 'Y'], ['a', 2, 'Y'], ['a', 3, 'Y'],
  ['a', 1, 'Z'], ['a', 2, 'Z'], ['a', 3, 'Z']
]
```

Copying an Array

When copying an array, it's essential to create an entirely new array, not just reference the original array. Observe that if we just reference the old array, then any new changes to the original array get propagated to the new array.

```
>>> arr = [1, 2]
>>> arr_copy = arr   # just references the original
    array

>>> arr.append(3)
>>> print(arr_copy)
[1, 2, 3]   # this is not [1, 2]
```

To avoid this issue, you need to create an entirely new array:

```
>>> arr = [1, 2]
>>> arr_copy = []

# create an entirely new array
>>> for item in arr:
      arr_copy.append(item)

>>> arr.append(3)
>>> print(arr_copy)
[1, 2]
```

Note that if our array contains items that are themselves arrays, then it's necessary to copy those items as well.

Implement `calc_cartesian_product`, verify that it reproduces the example above, and write a handful of other tests.

Part II

Searching and Sorting

7. Brute Force Search with Linear-Encoding Cryptography

An **encoding function** maps a string to a sequence of numbers. For example, you have already implemented the *trivial encoding function*, defined as follows:

```
' ' --> 0
'a' --> 1
'b' --> 2
...
'z' --> 26
```

There are many different types of encoding functions, but here we will focus on *linear encoding functions*. For example, using a linear encoding function $f(x) = 2x + 3$, we can encode the message `'a cat'` as follows:

```
Original message: 'a cat'
Trivial encoding: [1, 0, 3, 1, 20]
Linear encoding:  [2*1+3, 2*0+3, 2*3+3, 2*1+3, 2*20+3]
                  = [5, 3, 9, 5, 43]
```

Recovering a Message

If we want to recover our message from the linear encoding, we first solve for the inverse of the encoding function, i.e. the *decoding* function:

$$\begin{aligned} f(x) &= 2x + 3 \\ x &= 2f^{-1}(x) + 3 \\ f^{-1}(x) &= \frac{x-3}{2} \end{aligned}$$

Then, we apply the decoding function to the encoded message:

```
Linear encoding:  [5, 3, 9, 5, 43]
Trivial encoding: [(5-3)/2, (3-3)/2, (9-3)/2, (5-3)/2,
    (43-3)/2]
                  = [1, 0, 3, 1, 20]
Original message: 'a cat'
```

Recovering a Message with Multiple Possible Encoding Functions

Now, suppose we have a message `[3, 1, 9, 31, 15]` and we know that this message was encoded using a linear encoding function $f(x) = ax + b$ with $a \in \{1, 2\}$ and $b \in \{1, 2\}$. Can we break the code and recover the initial message?

The simplest way to do this is through **brute-force search**, which involves trying every single possibility. Here, there are 4 possible encoding functions:

$$f_{ab}(x) = ax + b$$

$$f_{11}(x) = 1x + 1$$
$$f_{12}(x) = 1x + 2$$
$$f_{21}(x) = 2x + 1$$
$$f_{22}(x) = 2x + 2$$

By inverting these encoding functions, we obtain the following decoding functions:

$$f_{ab}^{-1}(x) = \frac{x-b}{a}$$

$$f_{11}^{-1}(x) = \frac{x-1}{1}$$

$$f_{12}^{-1}(x) = \frac{x-2}{1}$$

$$f_{21}^{-1}(x) = \frac{x-1}{2}$$

$$f_{22}^{-1}(x) = \frac{x-2}{2}$$

Let's apply each of these decoding functions to our encoded message `[3, 1, 9, 31, 15]` and see what they come up with. Remember that in order to represent a message, the results must all be integers between 0 and 26 inclusive.

```
[3, 1, 9, 31, 15]

a, b
    [(3-b)/a, (1-b)/a, (9-b)/a, (31-b)/a, (15-b)/a]

a=1, b=1
    [(3-1)/1, (1-1)/1, (9-1)/1, (31-1)/1, (15-1)/1]
    [2, 0, 8, 30, 14]
    30 is too big
    does not represent a message

a=1, b=2
    [(3-2)/1, (1-2)/1, (9-2)/1, (31-2)/1, (15-2)/1]
    [1, -1, 7, 29, 13]
    -1 is too small, 29 is too big
    does not represent a message

a=2, b=1
    [(3-1)/2, (1-1)/2, (9-1)/2, (31-1)/2, (15-1)/2]
    [1, 0, 4, 15, 7]
    message: 'a dog'

a=2, b=2
    [(3-2)/2, (1-2)/2, (9-2)/2, (31-2)/2, (15-2)/2]
    [0.5, -0.5, 3.5, 14.5, 6.5]
    contains non-integer entries
    does not represent a message
```

We conclude that the original message was `'a dog'` and that it was encoded using $a = 2$ and $b = 1$.

Exercises

As usual, be sure to include a variety of tests.

1. Write a function `encode_string(string, a, b)` that encodes the `string` using the linear encoding function $f(x) = ax + b$ and returns the resulting array of numbers.

2. Write a function `decode_numbers(numbers, a, b)` that attempts to decode the `numbers` array under the assumption that the encoding function was $f(x) = ax + b$. If the numbers represent a message, then return the string corresponding to that message. Otherwise, return `False`.

3. Write a script to decode the message `[377, 717, 71, 513, 105, 921, 581, 547, 547, 105, 377, 717, 241, 71, 105, 547, 71, 377, 547, 717, 751, 683, 785, 513, 241, 547, 751]`, given that it was encoded with a linear encoding function $f(x) = ax + b$ where a and b are both integers between 0 and 100 inclusive. Be sure to print out all valid messages along with the values of a and b that generated them. Note that although there may be more than one valid message, only one will contain real words.

8. Solving Magic Squares via Backtracking

Brute force search can often be slow or infeasible when there are lots of possibilities that must be checked.

However, a technique called **backtracking** can often be used to drastically cut down the number of possibilities that must be checked. The idea is that whenever we find ourselves constructing a set of possibilities that we know are hopeless, we skip over them and *backtrack* to the the point right before we started constructing the hopeless possibilities.

Exercise: Brute Force

Use brute-force search to find all arrangements of the digits $1, 2, \ldots, 9$ into a 3×3 *magic square* where all the rows, columns, and diagonals add up to 15 and no digits are repeated.

You may use 9 nested for loops if you'd like. It's ugly but conceptually simple and gets the job done. To illustrate, pseudocode is provided below.

```
digits = [1, 2, ..., 9]
square = [
    [None, None, None]
    [None, None, None]
    [None, None, None]
]

for num1 in digits:
  clear out the square and put num1 in

  for num2 in digits excluding num1:
    clear out the square and put num1, num2 in

    for num3 in digits excluding num1, num2:
      clear out the square and put num1, num2, num3 in

      ... and so on

      for num9 in digits excluding num1, ..., num8:
        clear out the square and put num1, ..., num9 in

        if is_valid(square):
          print(square)
```

Note that this solution will be rather slow because it will require checking $9! = 362\,880$ arrangements of digits.

Exercise: Backtracking

Repeat the above exercise, but this time, whenever you reach an arrangement of digits that can no longer become a valid magic square, do not explore that arrangement any further. You can accomplish this by doing the following:

1. Write a function `is_hopeless(square)` to check whether an incomplete square could ever become a valid square. A square is hopeless if any row, column, or diagonal is filled in and does not sum to 15.

2. Place `continue` statements in your 9 nested for loops so that you skip inner loops whenever you detect a hopeless square.

Below are some examples to illustrate how the `is_hopeless` function should work.

```
>>> is_hopeless([
        [1,    2,    None],
        [None, 3,    None],
        [5,    6,    4   ]
    ])

OUTPUT: True

REASONING
- A diagonal is filled in and it doesn't sum to 15.

- No matter how we fill the rest of the square,
  this diagonal still won't sum to 15.

- Therefore, this incomplete arrangement cannot lead
  to any valid square.
```

```
>>> is_hopeless([
        [1,    2,    None],
        [None, None, None],
        [5,    6,    4   ]
    ])

OUTPUT: False

REASONING: There are no filled rows, columns, or
diagonals that don't sum to 15.
```

```
>>> is_hopeless([
        [None, None, None]
        [None, None, None]
        [None, None, None]
    ])

OUTPUT: False

REASONING: There are no filled rows, columns, or
diagonals that don't sum to 15.
```

Below is an illustration of how to skip inner loops using `continue` statements.

```
for num1 in digits:
    clear out the square and put num1 in it

    if is_hopeless(square):
        continue

    for num2 in digits excluding num1:
        clear out the square and put num1, num2 in it

        if is_hopeless(square):
            continue

        ... and so on
```

To give a concrete demonstration of backtracking, the first several arrangements that get explored are shown below.

```
[[_,_,_],
 [_,_,_],
 [_,_,_]]

[[1,_,_],
 [_,_,_],
 [_,_,_]]

[[1,2,_],
 [_,_,_],
 [_,_,_]]

[[1,2,3],
 [_,_,_],
 [_,_,_]]
^ hopeless -- do not explore further

[[1,2,4],
 [_,_,_],
 [_,_,_]]
^ hopeless -- do not explore further

[[1,2,5],
 [_,_,_],
 [_,_,_]]
^ hopeless -- do not explore further

...

[[1,2,9],
 [_,_,_],
 [_,_,_]]
^ hopeless -- do not explore further

[[1,3,_],
 [_,_,_],
 [_,_,_]]

...
```

9. Estimating Roots via Bisection Search and Newton-Raphson Method

Two of the simplest methods for estimating roots of functions are **bisection search** and the **Newton-Raphson method**. We will cover both of these here.

Bisection Search

To estimate the root of a function using **bisection search**, we start with a lower bound and an upper bound and then repeatedly move one bound halfway to the other.

Let's use bisection search to approximate the value of $\sqrt[3]{2}$ by approximating the root of the function $f(x) = x^3 - 2$. We'll use $x = 1$ as the lower bound and $x = 3$ as the upper bound since $f(1) < 0 < f(3)$. Note that the function values are positive to the right of the root and negative to the left of the root.

1. Our bounds are $[1, 3]$ and the midpoint 2. Since $f(2) = 6 > 0$, we know that 2 is bigger than the root. So, we use 2 as our new upper bound.

2. Our bounds are $[1, 2]$ and the midpoint 1.5. Since $f(1.5) = 1.375 > 0$, we know that 1.5 is bigger than the root. So, we use 1.5 as our new upper bound.

3. Our bounds are $[1, 1.5]$ and the midpoint is 1.25. Since $f(1.25) = -0.046875 < 0$, we know that 1.25 is smaller than the root. So, we use 1.25 as our new lower bound.

4. Our bounds are $[1.25, 1.5]$ and the midpoint is 1.375. Since $f(1.375) \approx 0.599609 > 0$, we know that 1.375 is bigger than the root. So, we use 1.375 as our new upper bound.

Our next bounds are $[1.25, 1.375]$, which tells us that $\sqrt[3]{2}$ is between 1.25 and 1.375. Our best guess is the midpoint, 1.3125, and the *precision* of our guess (i.e. the maximum amount we can be off by) is the distance from the midpoint to the bounds. In our case, the precision is 0.0625. We can keep on repeating the bisection procedure until our guess is as precise as we want.

Newton-Raphson Method

To estimate the root of a function using the **Newton-Raphson method**, we start with an initial guess and then repeatedly update our guess to be the root of the *tangent line* to the function.

To illustrate, let's again approximate the value of $\sqrt[3]{2}$ by approximating the root of the function $f(x) = x^3 - 2$. We'll use $x = 2$ as our initial guess. Remember that the slope of the tangent line is given by the derivative, which in this case is $f'(x) = 3x^2$.

1. Our guess is $x = 2$. The function value is $f(2) = 6$ and the slope of the tangent line is $f'(2) = 12$, so the tangent line is given by $y - 6 = 12(x - 2)$. The root of this tangent line is obtained by solving the equation $0 - 6 = 12(x - 2)$, which gives us $x = 1.5$.

2. Our guess is $x = 1.5$. The function value is $f(1.5) = 1.375$ and the slope of the tangent line is $f'(1.5) = 6.75$, so the tangent line is given by $y - 1.375 = 6.75(x - 1.5)$. The root of this tangent line is obtained by solving the equation $0 - 1.375 = 6.75(x - 1.5)$, which gives us $x \approx 1.2963$.

Our next guess would be $x = 1.2963$. Note that this is rounded to 4 decimal places, but we wouldn't actually round this number in our computer program. We can keep on repeating the Newton-Raphson procedure until our guesses *converge*, i.e. stay the same when rounded to the desired number of decimal places.

Lastly, note that to implement the Newton-Raphson procedure, it's necessary to come up with a formula for the root of the tangent line. If the tangent line is $y - y_0 = m(x - x_0)$, then the root is the value of x that solves the equation $0 - y_0 = m(x - x_0)$. You can find this value of x in terms of the other variables x_0, y_0, and m.

Exercises

As usual, be sure to include a variety of tests.

1. Write a script that approximates $\sqrt[3]{2}$ to a precision of 6 decimal places using bisection search. (This should match up against the provided example and continue for more iterations.)

2. Write a script that approximates $\sqrt[3]{2}$ to a precision of 6 decimal places using the Newton-Raphson method. (This should match up against the provided example and continue for more iterations.)

3. Write a function `calc_root_bisection(a, n, precision)` that approximates $\sqrt[n]{a}$ to the desired level of precision using bisection search.

4. Write a function `calc_root_newton_raphson(a, n, precision)` that approximates $\sqrt[n]{a}$ to the desired level of precision using the Newton-Raphson method.

10. Single-Variable Gradient Descent

Gradient descent is a technique for minimizing differentiable functions. The idea is that we take an initial guess as to what the minimum is, and then repeatedly use the gradient to nudge that guess further and further "downhill" into an actual minimum.

Let's start by considering the simple case of minimizing a single-variable function – say, $f(x) = x^2$ with the initial guess that the minimum is at $x = 1$. Of course, this guess is not correct since the minimum is not actually at $x = 1$, but we will repeatedly update the guess to become more and more correct.

Intuition About the Sign of the Derivative

To update our guess, we start by computing the gradient at $x = 1$. For a single-variable function, the gradient is just the plain old derivative.

$$f(x) = x^2$$
$$f'(x) = 2x$$
$$f'(1) = 2 \cdot 1 = 2$$

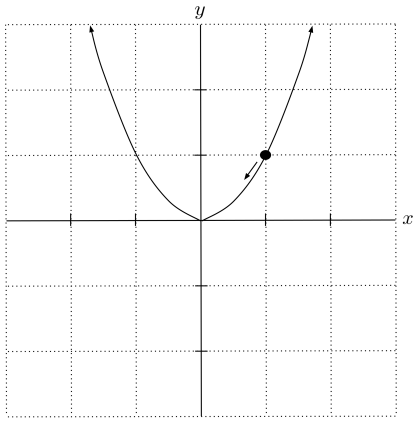
We found that the derivative is $f'(1) = 2$. Now that we know the value of the derivative at our guess, we can use it to update our guess. In general, we have the following intuition:

- If the derivative is *positive*, then the function is sloping *up*, and we can move downhill by nudging our guess to the *left* (i.e. *decreasing* the x-value of our guess).
- If the derivative is *negative*, then the function is sloping *down*, and we can move downhill by nudging our guess to the *right* (i.e. *increasing* our the x-value of our guess).

Put more simply:

- If the function is increasing, then we should decrease our guess.
- If the function is decreasing, then we should increase our guess.

Here, the derivative $f'(1) = 2$ is positive, which means the function is increasing and we should *decrease* the x-value of our guess. Indeed, if we sketch up a graph of the situation, we can see that decreasing our guess will take us closer to the actual minimum:



Gradient Descent Formula

Now, the question is: by how much should we update our guess? If we decrease the x-value of our guess by too much, then we'll pass right over the minimum and end up on the other side, maybe even further away from the minimum than we were to start with. But if we decrease the x-value of our guess by too little, then we will barely move at all and we will have to repeat this procedure an excessive number of times before actually getting close to the minimum.

For most differentiable functions that appear in machine learning, the amount by which we should update our guess depends on how steep the graph is. The steeper the graph is, the further we are from the minimum, and the more freedom we have to update our guess without worrying about moving it too far. So, we have the following formula for our next guess:

$$x_{n+1} = x_n - \alpha f'(x_n)$$

Before we perform any computations with this formula, let's elaborate on each part of it.

- x_n is our nth guess (i.e. the current guess), and x_{n+1} is the next guess.
- $f'(x_n)$ is the derivative at our current guess, and it tells us whether the function is increasing or decreasing (as well as how steep the function is).
- α is a positive constant called the *learning rate*, and the quantity $\alpha f'(x_n)$ is the amount by which we update our guess. (We generally try to run the gradient descent algorithm with a learning rate around $\alpha \approx 0.01$ to start with, but if this learning rate causes our guesses to diverge, then decrease the learning rate and try again. More on that later.)
- We are subtracting $\alpha f'(x_n)$ because we want to move our guess x_n in the *opposite* direction as the derivative $f'(x_n)$. Remember that if the function is increasing (i.e. positive derivative) then we want to move our guess to the left (i.e. decrease it), and if the function is decreasing (i.e. negative derivative) then we want to move our guess to the right (i.e. increase it). The steeper the graph is (i.e. the larger the magnitude of the derivative), the further we want to move our guess.

Worked Example

Now, let's finish working out our example. We are trying to minimize the function $f(x) = x^2$, our first guess is $x_0 = 1$, and the derivative at this guess is $f'(1) = 2$. If we use a learning rate of $\alpha = 0.01$, then our next guess is

$$\begin{aligned} x_1 &= x_0 - \alpha f'(x_0) \\ &= 1 - (0.01)\, f'(1) \\ &= 1 - (0.01)(2) \\ &= 0.98. \end{aligned}$$

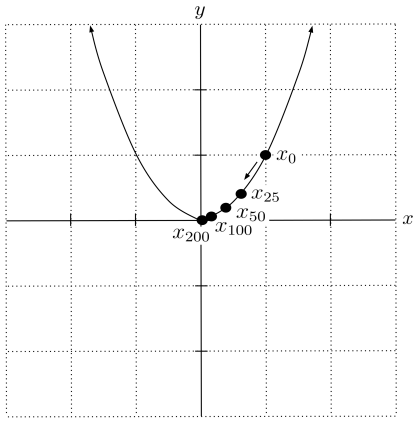
To get the guess after that, we simply repeat the procedure again:

$$\begin{aligned} x_2 &= x_1 - \alpha f'(x_1) \\ &= 0.98 - (0.01)\, f'(0.98) \\ &= 0.98 - (0.01)(1.96) \\ &= 0.9604 \end{aligned}$$

In practice, we usually implement gradient descent as a computer program because it is extremely tedious to do by hand. You should do this and verify that you get the same results as shown in the table below. In the table, we will round to 6 decimal places (but we do not actually round in our computer program).

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	1	2	0.02
1	0.98	1.96	0.0196
2	0.9604	1.9208	0.019208
3	0.941192	1.882384	0.018824
⋮	⋮	⋮	⋮
25	0.603465	1.206929	0.012069
50	0.364170	0.728339	0.007283
100	0.132620	0.265239	0.002652
200	0.017588	0.035176	0.000352
300	0.002333	0.004665	0.000047
400	0.000309	0.000619	0.000006

We can visualize these results on our graph. Indeed, we see that our guesses are converging to the minimum at $x = 0$.



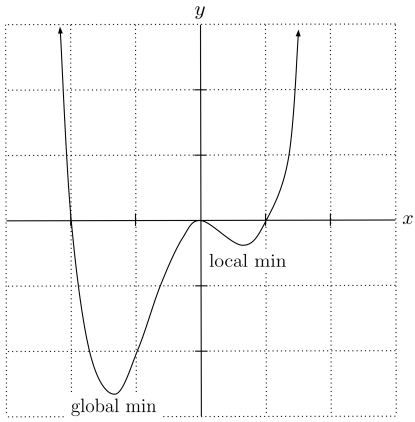
Local vs Global Minima

There is one big caveat to gradient descent that is essential to understand: it is not guaranteed to converge to a global minimum. It's possible for gradient descent to get "stuck" in a local minimum, just like a ball rolling down a hill might roll into a depression and get stuck there instead of falling all the way to the bottom of the hill.

To illustrate this caveat numerically, let's use gradient descent to find the minimum value of the function $f(x) = x^4 + x^3 - 2x^2$ starting with the initial guess $x = 1$. Note that the derivative of this function is $f'(x) = 4x^3 + 3x^2 - 4x$.

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	1	3	0.03
1	0.97	2.593392	0.025934
2	0.944066	2.263154	0.022632
3	0.921435	1.990732	0.019907
⋮	⋮	⋮	⋮
25	0.736701	0.280693	0.002807
50	0.701878	0.053456	0.000535
100	0.693413	0.002445	0.000024
200	0.6930014	0.000005	0.000000

Uh oh! We are converging to the local minimum at $x = 0.69$, but this is not the global minimum of the function.



Trying Different Initial Guesses

Because it's possible for gradient descent to get stuck in local minima, it's often a good idea to run the gradient descent procedure with several different initial guesses and use whichever final guess gives the best result (i.e. lowest function value).

To illustrate, let's run the gradient descent algorithm a couple more times with different initial guesses. For the next run, we'll start with the initial guess $x = 0$.

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0

Uh oh! This is even worse. We started at the top of a hill, and the function is flat there (i.e. the derivative is zero), so our guesses are not changing at all. In other words, the ball is not rolling down the hill because it was placed at the very top where it's flat.

Let's try again, this time with initial guess $x = -1$.

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	-1	3	0.03
1	-1.03	2.931792	0.029318
2	-1.059318	2.848862	0.028489
3	-1.087807	2.752289	0.027523
⋮	⋮	⋮	⋮
25	-1.410141	0.389810	0.003898
50	-1.441723	0.015732	0.000157
100	-1.442999	0.000022	0.000000
200	-1.443000	0.000000	0.000000

There we go! This time, we reached the desired global minimum.

Final Remarks

There are many different variations of gradient descent that attempt to better avoid getting stuck in local minima. For example, some variations incorporate the "momentum" of a ball rolling down a hill to help the guesses "roll through" shallow local minima, whereas other variations include random perturbations.

Additionally, there are ways to choose better initial guesses. In the example above, we did not use any particular rationale when choosing our initial guesses ($x = 1, 0, -1$). But we could have (for instance) randomly generated a large number of initial guesses, plugged each one into our function $f(x)$, and only run gradient descent on those initial guesses that were associated with the lowest values of $f(x)$.

Also note that it's often necessary to try out different learning rates. In the examples above, we used a learning rate of $\alpha = 0.01$ and this worked out fine. If we had chosen a learning rate that was too high, say $\alpha = 0.5$, then our guesses would update by too large an amount each time, causing them to jump too far and never converge. This is demonstrated below.

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	-1	3	1.5
1	-2.5	-33.75	-16.875
2	14.375	12444	6222
3	-6208	-956777563439	-478388781719

On the other hand, if we had chosen a learning rate that was too low, say $\alpha = 0.000001$, then our guesses would update by too miniscule an amount each time and we would require an excessive number of iterations to converge. This is demonstrated below.

n	x_n	$f'(x_n)$	$\alpha f'(x_n)$
0	-1	3	0.000003
1	-1.000003	2.999994	0.000003
10	-1.000030	2.999940	0.000003
100	-1.000300	2.999399	0.000003
1000	-1.002997	2.993925	0.000003
10000	-1.029675	2.932619	0.000003
100000	-1.250157	1.873863	0.000002
1000000	-1.442997	0.000046	0.000000

Lastly, note that we can maximize functions by performing gradient *ascent*, which is similar to gradient descent except that we replace the subtraction with addition in the update rule: $x_{n+1} = x_n + \alpha f'(x_n)$.

Exercises

Use gradient descent to minimize the following functions. Before looking at the graph, perform gradient descent using several different initial guesses, and then plug your final guesses back into the function to determine which final guess is the best (i.e. gives you the lowest function value). Then, look at the graph to check whether you found the global maximum.

1. $f(x) = x^2 + x + 1$
2. $f(x) = x^3 - x^4 - x^2$

3. $f(x) = \dfrac{\sin x}{1 + x^2}$

4. $f(x) = 3\cos x + x^2 e^{\sin x}$

11. Multivariable Gradient Descent

Multivariable gradient descent is similar to single-variable gradient descent, except that we replace the derivative $f'(x)$ with the gradient $\nabla f(\vec{x})$ as follows:

$$\vec{x}_{n+1} = \vec{x}_n - \alpha \nabla f(\vec{x}_n)$$

Here, $\vec{x}_n$ denotes the vector of nth guesses for all the variables. For example, if f is a function of 2 input variables x, y, then we denote

$$\vec{x}_n = \langle x_n, y_n \rangle$$

and the update rule can be expressed as follows:

$$\begin{aligned}\langle x_{n+1}, y_{n+1}\rangle &= \langle x_n, y_n\rangle - \alpha \nabla f(x_n, y_n) \\ &= \langle x_n, y_n\rangle - \alpha \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right\rangle \Bigg|_{(x_n, y_n)}\end{aligned}$$

Worked Example

To illustrate, let's use gradient descent to minimize the following function:

$$f(x, y) = x \sin y + x^2$$

First, we work out the gradient:

$$\begin{aligned}\nabla f(x, y) &= \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right\rangle \\ &= \left\langle \frac{\partial}{\partial x}\left(x \sin y + x^2\right), \frac{\partial}{\partial y}\left(x \sin y + x^2\right) \right\rangle \\ &= \langle \sin y + 2x, x \cos y \rangle\end{aligned}$$

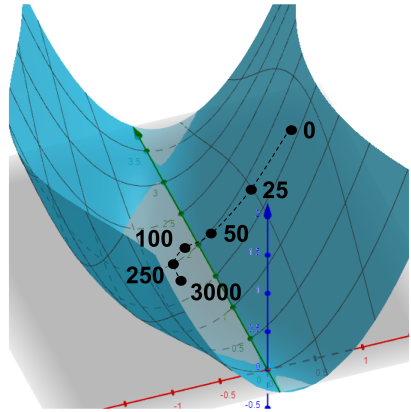
If we start with the initial guess $x = 1, y = 2$ (which we denote as $\langle x, y\rangle_0 = \langle 1, 2\rangle$) and use a learning rate of $\alpha = 0.01$, then our next guess is

$$\begin{aligned}\langle x_1, y_1\rangle &= \langle x_0, y_0\rangle - \alpha\nabla f(x_0, y_0)\\ &= \langle 1, 2\rangle - (0.01)\left\langle \sin y + 2x, x\cos y\right\rangle\big|_{(1,2)}\\ &= \langle 1, 2\rangle - (0.01)\langle \sin 2 + 2, \cos 2\rangle\\ &\approx \langle 0.970907, 2.004161\rangle\,.\end{aligned}$$

We will carry out the rest of the iterations using a computer program. You should do this too and verify that you get the same results as shown in the table below. In the table, we will round to 6 decimal places (but we do not actually round in our computer program).

n	$\langle x_n, y_n\rangle$	$\nabla f(x_n, y_n)$
0	$\langle 1, 2\rangle$	$\langle 2.909297, -0.416147\rangle$
1	$\langle 0.970907, 2.004161\rangle$	$\langle 2.849372, -0.40771\rangle$
2	$\langle 0.942413, 2.008239\rangle$	$\langle 2.790665, -0.399229\rangle$
3	$\langle 0.914507, 2.012231\rangle$	$\langle 2.733153, -0.390711\rangle$
$\vdots$	$\vdots$	$\vdots$
25	$\langle 0.427158, 2.078619\rangle$	$\langle 1.728121, -0.207717\rangle$
50	$\langle 0.086462, 2.109688\rangle$	$\langle 1.031202, -0.044371\rangle$
100	$\langle -0.242655, 2.084195\rangle$	$\langle 0.385770, 0.119178\rangle$
250	$\langle -0.457667, 1.862063\rangle$	$\langle 0.042547, 0.131426\rangle$
500	$\langle -0.496295, 1.657935\rangle$	$\langle 0.003616, 0.043192\rangle$
1000	$\langle -0.499975, 1.577936\rangle$	$\langle 0.000025, 0.003569\rangle$
2000	$\langle -0.500000, 1.570844\rangle$	$\langle 0.000000, 0.000024\rangle$
3000	$\langle -0.500000, 1.570797\rangle$	$\langle 0.000000, 0.000000\rangle$

If we plot graph of the surface and some of our intermediate guesses, we can see that our guesses do indeed take us down the valley into a minimum:



Sanity Check

When we run gradient descent on functions with more than two input variables, it becomes difficult to visualize the graph of the function. However, we can still verify that we're moving in the correct direction by evaluating the function at each guess and making sure the function values are decreasing.

To illustrate, let's add another column $f(x_n, y_n)$ on the right side of the table above. We can see that the function values in this column are decreasing, and this tells us that we are successfully minimizing our function f.

n	$\langle x_n, y_n \rangle$	$\nabla f(x_n, y_n)$	$f(x_n, y_n)$
0	$\langle 1, 2 \rangle$	$\langle 2.909297, -0.416147 \rangle$	1.909297
1	$\langle 0.970907, 2.004161 \rangle$	$\langle 2.849372, -0.40771 \rangle$	1.823815
2	$\langle 0.942413, 2.008239 \rangle$	$\langle 2.790665, -0.399229 \rangle$	1.741817
3	$\langle 0.914507, 2.012231 \rangle$	$\langle 2.733153, -0.390711 \rangle$	1.663164
$\vdots$	$\vdots$	$\vdots$	$\vdots$
25	$\langle 0.427158, 2.078619 \rangle$	$\langle 1.728121, -0.207717 \rangle$	0.555717
50	$\langle 0.086462, 2.109688 \rangle$	$\langle 1.031202, -0.044371 \rangle$	0.081684
100	$\langle -0.242655, 2.084195 \rangle$	$\langle 0.385770, 0.119178 \rangle$	-0.152491
250	$\langle -0.457667, 1.862063 \rangle$	$\langle 0.042547, 0.131426 \rangle$	-0.228931
500	$\langle -0.496295, 1.657935 \rangle$	$\langle 0.003616, 0.043192 \rangle$	-0.248103
1000	$\langle -0.499975, 1.577936 \rangle$	$\langle 0.000025, 0.003569 \rangle$	-0.249987
2000	$\langle -0.500000, 1.570844 \rangle$	$\langle 0.000000, 0.000024 \rangle$	-0.250000
3000	$\langle -0.500000, 1.570797 \rangle$	$\langle 0.000000, 0.000000 \rangle$	-0.250000

Exercises

Use gradient descent to minimize the following functions. Use several different initial guesses, and then plug your final guesses back into the function to determine which final guess is the best (i.e. gives you the lowest function value). Be sure to evaluate the function at each guess and verify that the function values are decreasing.

1. $f(x, y) = (x - 1)^2 + 3y^2$
2. $f(x, y) = y^2 + y \cos x$
3. $f(x, y, z) = (x - 1)^2 + 3(y - 2)^2 + 4(z + 1)^2$
4. $f(x, y, z) = x^2 + 3y^2 + 4z^2 + \cos(xyz)$

12. Selection, Bubble, Insertion, and Counting Sort

There are many different methods for sorting items in arrays. Let's go over some of the simplest methods.

Selection Sort

The absolute simplest sorting method, **selection sort**, involves building up a separate sorted array by repeatedly taking the minimum element from the original array. To illustrate, let's sort the array `[3, 5, 8, 2, 5]` using selection sort.

- Our original array is `[3, 5, 8, 2, 5]` and our sorted array is empty `[ ]` since we just started. The minimum element of our original array is `2`, and we move this element to the sorted array.

- Our original array is now `[3, 5, 8, 5]` and our sorted array is `[2]`. The minimum element of our original array is `3`, and we move this element to the sorted array.
- Our original array is now `[5, 8, 5]` and our sorted array is `[2, 3]`. The minimum element of our original array is `5`, and we move this element to the sorted array.
- Our original array is now `[8, 5]` and our sorted array is `[2, 3, 5]`. The minimum element of our original array is `5`, and we move this element to the sorted array.
- Our original array is now `[8]` and our sorted array is `[2, 3, 5, 5]`. The minimum element of our original array is `8`, and we move this element to the sorted array.
- Our original array is now empty `[ ]` and our sorted array is `[2, 3, 5, 5, 8]`. We're done!

Bubble Sort

Another simple sorting method, **bubble sort** involves repeatedly looping through all pairs of consecutive elements in the array and swapping them if they are out of order. Let's sort the array `[3, 5, 8, 2, 5]` using bubble sort.

1. First pass
    - `[(3, 5), 8, 2, 5]` - The first pair of consecutive elements is `(3,5)`. These elements are in the correct order, so we do nothing.

- `[3, (5, 8), 2, 5]` - The next pair of consecutive elements is `(5,8)`. These elements are in the correct order, so we do nothing.
- `[3, 5, (8, 2), 5]` - The next pair of consecutive elements is `(8,2)`. These elements are out of order, so we swap them. Our updated array is `[3, 5, 2, 8, 5]`.
- `[3, 5, 2, (8, 5)]` - The next pair of consecutive elements is `(8,5)`. These elements are out of order, so we swap them. Our updated array is `[3, 5, 2, 5, 8]`.

2. Since we made a swap in the first pass, we proceed to a second pass.
   - `[(3, 5), 2, 5, 8]`
   - `[3, (5, 2), 5, 8]` - Swap. The array is now `[3, 2, 5, 5, 8]`.
   - `[3, 2, (5, 5), 8]`
   - `[3, 2, 5, (5, 8)]`
3. Since we made a swap in the second pass, we proceed to a third pass.
   - `[(3, 2), 5, 5, 8]` - Swap. The array is now `[2, 3, 5, 5, 8]`.
   - `[2, (3, 5), 5, 8]`
   - `[2, 3, (5, 5), 8]`
   - `[2, 3, 5, (5, 8)]`
4. Since we made a swap in the third pass, we proceed to a fourth pass.

   - `[(2, 3), 5, 5, 8]`
   - `[2, (3, 5), 5, 8]`
   - `[2, 3, (5, 5), 8]`
   - `[2, 3, 5, (5, 8)]`

5. Since we made no swaps in the fourth pass, we are done!

Insertion Sort

Insertion sort is similar to bubble sort, with one key difference. Whenever we find something out of order, instead of carrying out just one swap, we repeatedly swap the out-of-order element backwards until it is in the correct position. Let's illustrate.

- `[(3, 5), 8, 2, 5]`
- `[3, (5, 8), 2, 5]`
- `[3, 5, (8, 2), 5]` - Swap and note where we left off. We will now start looking backwards, continuing to swap the `2` until it is in the correct position.
- `[3, (5, 2), 8*, 5]` - Swap and continue looking backwards.
- `[(3, 2), 5, 8*, 5]` - Swap. Normally we would continue looking backwards, but we can't go backwards any more since we're at the beginning of the array. So, we're done dealing with the `2` and we can pick up from where we left off.

- `[2, 3, 5, (8, 5)]` - Swap and note where we left off. We will now start looking backwards, continuing to swap the `5` until it is in the correct position.
- `[2, 3, (5, 5), 8*]` - No swap needed. The `5` is in the correct position. Normally we would pick up from where we left off, but we left off at the end of the array, so we're done!

Counting Sort

Lastly, **counting sort** is quite different from all the sorting methods we've covered so far. It's more involved, so we will outline the procedure before showing an example. The procedure is as follows:

1. Identify the minimum number in the array and then subtract it from all numbers in the array. This way, the updated array has a minimum of `0` and all other elements are positive.
2. Let `N` be the maximum number in the array. Create another array, `counts`, with `N+1` entries, all initialized to `0`.
3. Loop through the array. For each number `n` that you encounter, increment `counts[n]`. This way, the value of `counts[n]` represents the amount of times that you encountered the number `n`.
4. Read off the `counts` array into a sorted array that contains `counts[n]` instances of each number `n`.

5. In the first step, you subtracted the minimum number of the original array. Undo that by adding the minimum number to each element in your sorted array. Finally, we're done!

Below is an example of applying counting sort to sort the array `[3, 5, 8, 2, 5]`.

1. The minimum number is `2`. Subtracting `2` from all numbers in the array, the updated array is `[1,3,6,0,3]`.
2. The maximum number in the new array is `6`. So, we let `counts = [0,0,0,0,0,0,0]`.
3. We loop through the array `[1,3,6,0,3]` and increment `counts`.
   - The first element is `1`, so we increment `counts[1]`. Now we have `counts = [0,1,0,0,0,0,0]`.
   - The next element is `3`, so we increment `counts[3]`. Now we have `counts = [0,1,0,1,0,0,0]`.
   - The next element is `6`, so we increment `counts[6]`. Now we have `counts = [0,1,0,1,0,0,1]`.
   - The next element is `0`, so we increment `counts[0]`. Now we have `counts = [1,1,0,1,0,0,1]`.
   - The next element is `3`, so we increment `counts[3]`. Now we have `counts = [1,1,0,2,0,0,1]`.
4. We read off the array `counts = [1,1,0,2,0,0,1]` as follows: `1` zero, `1` one, `0` twos, `2` threes, `0` fours, `0` fives, `1` six. So, we have a sorted array `[0, 1, 3, 3, 6]`.

5. In the first step, we subtracted `2`. Now we add that back to each item of the sorted array and get `[2, 3, 5, 5, 8]`. We're done!

Time Complexity

It's common to refer to **time complexity** when talking about the speed of various algorithms. Selection, bubble, and insertion sort all have average-case time complexity $O(n^2)$, pronounced *order of n squared*. Loosely speaking, this means that if we were to create a mathematical expression for the average number of operations required to sort a list of n elements, the variable part of the dominating term would be n^2.

To understand why selection sort is $O(n^2)$, remember that selection sort involves repeatedly building up a separate sorted array by repeatedly taking the minimum element from the original array. The original array initially consists of n elements, and we need to check each of them when computing the minimum, so there are n operations. Once we've computed the minimum and moved it to the sorted array, we need to repeat the procedure on the remaining $n - 1$ elements. Continuing the pattern and adding up all the operations, we get the following expression:

$$\begin{aligned} & n + (n-1) + (n-2) + \ldots + 1 \\ &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \end{aligned}$$

Indeed, the dominating term is $\frac{1}{2}n^2$, and its variable part is n^2. The time complexities of bubble and insertion sort can be derived in a similar way.

Counting sort is generally faster than selection, bubble, and insertion sort. However, the drawback is that it can require lots of memory.

Exercises

First, write a function `calc_min(arr)` that calculates the minimum element of an array by looping through and keeping track of the smallest element that has been found. Then, for each sorting method described above, write a function that takes an input array and sorts it using the described procedure.

Do not use the built-in `min()` function; instead, use your `calc_min` function. Be sure to write plenty of tests that cover a variety of cases (negative numbers, repeated numbers, duplicates, decimals, etc).

1. Write `calc_min(arr)`.
2. Write `selection_sort(arr)`.

3. Write `bubble_sort(arr)`.
4. Write `insertion_sort(arr)`.
5. Write `counting_sort(arr)`.
6. Derive the average-case time complexity of bubble sort.
7. Derive the average-case time complexity of insertion sort.
8. Derive the average-case time complexity of counting sort.
9. Construct an example in which counting sort requires drastically more memory than selection, bubble, and insertion sort. (Hint: what would cause the `counts` array to become enormous?)

13. Merge Sort and Quicksort

Merge sort is a sorting algorithm that works by breaking an array in half, recursively calling merge sort on each half separately, and then merging the two sorted halves. The base case is that empty arrays and arrays of 1 item are already sorted (so if you call merge sort on such an array, it leaves the array as-is).

```
merge_sort(array):
    if the array is empty or has 1 item:
        return the array as-is

    otherwise:
        break the array in half
        recursively call merge_sort on each half
            separately
        merge the two halves and return the result
```

Note that when an odd-length array is broken in half, it is okay for one of the halves to have one more item than the other half.

Example of Merge Sort

Below is a concrete example illustrating how `merge_sort` sorts an array.

```
merge_sort([6,9,7,4,2,1,8,5])
|    halves: [6,9,7,4] [2,1,8,5]
|
|--> merge_sort([6,9,7,4])
     |    halves: [6,9] [7,4]
     |
     |--> merge_sort([6,9])
          |    halves: [6] [9]
          |
          |--> merge_sort([6])
               |
          |<-- [6]
          |
          |--> merge_sort([9])
               |
          |<-- [9]
          |
          merge([6],[9])
     |<-- [6,9]
     |
     |--> merge_sort([7,4])
          |    halves: [7] [4]
          |
          |--> merge_sort([7])
               |
          |<-- [7]
          |
          |--> merge_sort([4])
               |
          |<-- [4]
          |
          merge([7],[4])
     |<-- [4,7]
     |
     merge([6,9],[4,7])
|<-- [4,6,7,9]
```

```
|
|--> merge_sort([2,1,8,5])
     |    halves: [2,1] [8,5]
     |
     |--> merge_sort([2,1])
          |    halves: [2] [1]
          |
          |--> merge_sort([2])
               |
          |<-- [2]
          |
          |--> merge_sort([1])
               |
          |<-- [1]
          |
          merge([2],[1])
     |<-- [1,2]
     |
     |--> merge_sort([8,5])
          |    halves: [8] [5]
          |
          |--> merge_sort([8])
               |
          |<-- [8]
          |
          |--> merge_sort([5])
               |
          |<-- [5]
          |
          merge([8],[5])
     |<-- [5,8]
     |
     merge([1,2],[5,8])
|<-- [1,2,5,8]
|
merge([4,6,7,9],[1,2,5,8])
[1,2,4,5,6,7,8,9]
```

Quicksort

Another sorting algorithm, **quicksort**, is very similar to merge sort. The only difference is that instead of splitting the array in half, we randomly choose a *pivot element* and split the array into 3 pieces: elements that are less than the pivot element, elements that are equal to the pivot element, and elements that are greater than the pivot element. Then, we apply quicksort recursively on the "less than" and "greater than" pieces before combining the pieces.

```
quicksort([6,9,7,4,2,1,8,5])
|     pivot: 7
|     pieces: [6,4,2,1,5] [7] [9,8]
|
|--> quicksort([6,4,2,1,5])
     |     pivot: 5
     |     pieces: [4,2,1] [5] [6]
     |
     |--> quicksort([4,2,1])
          |     pivot: 2
          |     pieces: [1] [2] [4]
          |
          |--> quicksort([1])
               |
          |<-- [1]
          |
          |--> quicksort([4])
               |
          |<-- [4]
          |
          combine([1],[2],[4])
     |<-- [1,2,4]
     |
     |--> quicksort([6])
          |
     |<-- [6]
     |
     combine([1,2,4],[5],[6])
```

```
|<-- [1,2,4,5,6]
|
|--> quicksort([9,8])
     |    pivot: 9
     |    halves: [8] [9] []
     |
     |--> quicksort([8])
     |         |
     |<-- [8]
     |
     |--> quicksort([])
     |         |
     |<-- []
     |
     combine([8],[9],[])
|<-- [8,9]
|
combine([1,2,4,5,6],[7],[8,9])
[1,2,4,5,6,7,8,9]
```

Time Complexity

With an average-case time complexity of $O(n \log n)$, merge sort and quicksort are generally faster than selection, bubble, and insertion sort. And unlike counting sort, they are not susceptible to blowup in the amount of memory required.

Exercises

Implement merge sort and quicksort. As always, be sure to write plenty of tests that cover a variety of cases (negative numbers, repeated numbers, duplicates, decimals, etc).

1. Write a helper function `merge(arr1, arr2)` that merges two sorted arrays by repeatedly looking at the first element of each array and moving the smaller one into a new array.
2. Write a function `merge_sort(arr)` that sorts an array using the merge sort algorithm.
3. Write a function `quicksort(arr)` that sorts an array using the quicksort algorithm.

Part III

Objects

14. Basic Matrix Arithmetic

Let's use arrays to implement matrices and their associated mathematical operations. We will jump straight to exercises – it is assumed that you are already familiar with matrix arithmetic and reduced row echelon form.

Exercise: Addition, Subtraction, Transpose, and Scalar Multiplication

To start, create a class `Matrix` that implements basic matrix arithmetic.

```
>>> A = Matrix([
    [1, 2],
    [3, 4]
  ])

>>> A.show()
[ 1 , 2 ]
[ 3 , 4 ]

>>> At = A.transpose()
>>> At.show()
[ 1 , 3 ]
[ 2 , 4 ]

>>> A.add(At).show()
[ 2 , 5 ]
[ 5 , 8 ]
```

In addition to the methods shown above, you should also implement `subtract` and `scalar_multiply`. (The input to `scalar_multiply` is a number, and it should multiply all entries of the matrix.)

To keep your code clean, you'll need to implement some helper attributes like `num_cols` and `num_rows`.

```
>>> A = Matrix([
    [1, 2, 3],
    [4, 5, 6]
  ])

>>> A.num_cols
3

>>> A.num_rows
2
```

Your class (and tests) should be general to matrices of any dimensions. Do not assume the matrix is square. However, if you struggle and get stuck, then you can build some scaffolding for yourself by first hard-coding a class for, say, a 2×3 matrix. Handling the general case often becomes easier once you've worked through a specific case.

Remember that some operations impose constraints on the dimensions of the matrices involved. For example, you cannot add two matrices unless their dimensions are equal. In your code, be sure to check that all such constraints are satisfied before attempting to carry out the operation. If any constraint is not satsified, then throw an error with a descriptive message so that the user can understand why your matrix class is unable to perform the desired computation.

Exercise: Matrix Multiplication and Recursive Determinant

Next, implement the method `matrix_multiply`. This will be trickier than the methods above, but remember that the element at row i and column j in the product AB is just the dot product of row i in A and column j in B. (This means that you should write a helper function for computing dot products.)

Then, implement the method `recursive_determinant` which computes the determinant using recursive cofactor expansion. As a refresher, below is an example of using recursive cofactor expansion to compute the determinant of a 3×3 matrix.

$$\det \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$= 1 \det \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} - 2 \det \begin{bmatrix} 4 & 6 \\ 7 & 9 \end{bmatrix} + 3 \det \begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}$$

$$= 1 \left(5 \det [9] - 6 \det [8]\right) - 2 \left(4 \det [9] - 6 \det [7]\right) + 3 \left(4 \det [8] - 5 \det [7]\right)$$

$$= 1 \left(5 \cdot 9 - 6 \cdot 8\right) - 2 \left(4 \cdot 9 - 6 \cdot 7\right) + 3 \left(4 \cdot 8 - 5 \cdot 7\right)$$

$$= 1 \left(45 - 48\right) - 2 \left(36 - 42\right) + 3 \left(32 - 35\right)$$

$$= 1 \left(-3\right) - 2 \left(-6\right) + 3 \left(-3\right)$$

$$= -3 + 12 - 9$$

$$= 0$$

15. Reduced Row Echelon Form and Applications to Matrix Arithmetic

Recall from linear algebra the following procedure for converting a matrix to reduced row echelon form (RREF):

```
row_index = 0

for each column:
  if pivot row exists for column:

    if pivot row does not match current row_index:
      swap current row with pivot row
      (so that it matches)

    divide pivot row (so that first nonzero entry is 1)

    clear entries below and above pivot entry
    (by subtracting multiples of pivot row)

    row_index += 1
```

This algorithm is a bit tricky, so before you attempt to implement anything, be sure to work out the above algorithm by hand on several concrete examples until you feel comfortable with it.

Again, we will jump straight to exercises – it is assumed that you are already familiar with matrix arithmetic and reduced row echelon form.

Exercise: RREF with Helper Functions

Once you're comfortable working out the algorithm by hand, write some helper functions.

- For example, the first helper method you'll need to write will check whether a pivot row exists for a particular column in a matrix. If it does exist, then return the index of the row. Otherwise, return nothing.
- There are at least 3 other helper methods that you'll need to write. But don't stop thinking once you come up with 3 helper methods. Keep going until you think you've really narrowed down the problem to its atomic sub-procedures.

It's often a good idea to write tests for your helper functions and get them passing *before* you start trying to use them together, especially when you're fairly new to coding. In your tests, be sure to consider a variety of matrices that are substantially different from each other. When writing tests, your goal is to try (and fail) to break the function that you wrote.

Once you've written the helper functions, you can stitch them together into the full RREF algorithm.

Exercise: Inverse via RREF

Once you've written a working RREF algorithm, you can use it to implement a method that computes the inverse of a matrix. Remember that to compute the inverse of a square matrix A, you can just

1. augment the matrix as $[A|I]$ where I is the identity matrix,
2. run RREF on the augmented matrix to convert it to $[I|A^{-1}]$, and then
3. read A^{-1} on the right-hand side.

Exercise: Determinant via RREF

You can also use the RREF algorithm to compute determinants much faster than with the recursive cofactor expansion method.

- Whenever you divide a row by some amount during the RREF algorithm, the determinant gets divided by that same amount.
- Whenever you swap two rows during the RREF algorithm, the determinant switches sign (from positive to negative or vice versa).

- At the end of the RREF algorithm, the final matrix has a determinant of 1.
- So, you can "build up" the determinant during the execution of the RREF algorithm by keeping track of the product of the numbers you divide by and switching the sign every time you swap two rows.

To avoid cluttering up your RREF algorithm, it is advisable to just copy-paste it into a new `determinant` method and then make whatever adjustments you need to build up the determinant along the way.

16. K-Means Clustering

Clustering is the process of grouping similar records within data. Some examples of clusters in real-life data include users who buy similar items, songs in a similar genre, and patients with similar health conditions.

One of the simplest clustering methods is called **k-means clustering**. It works by guessing some initial clusters in the data and then repeatedly updating the guesses to make the clusters more cohesive.

1. Initialize the clusters.
    1.1 Randomly divide the data into k parts (where k is an input parameter). Each part is an initial cluster.
    1.2 Compute the mean of each part. Each mean is an initial cluster center.
2. Update the clusters.
    2.1 Re-assign each record to the cluster with the nearest center.
    2.2 Compute the new cluster centers by taking the mean of the records in each cluster.
3. Keep repeating step 2 until the clusters stop changing.

Worked Example

As a concrete example, consider the following data set. Each row represents a cookie with some ratio of ingredients. We can use k-means clustering to separate the data into different overarching "types" of cookies.

```
columns = ['Portion Eggs', 'Portion Butter', 'Portion
    Sugar', 'Portion Flour']

data = [
    [0.14, 0.14, 0.28, 0.44],
    [0.22, 0.1,  0.45, 0.33],
    [0.1,  0.19, 0.25, 0.4 ],
    [0.02, 0.08, 0.43, 0.45],
    [0.16, 0.08, 0.35, 0.3 ],
    [0.14, 0.17, 0.31, 0.38],
    [0.05, 0.14, 0.35, 0.5 ],
    [0.1,  0.21, 0.28, 0.44],
    [0.04, 0.08, 0.35, 0.47],
    [0.11, 0.13, 0.28, 0.45],
    [0.0,  0.07, 0.34, 0.65],
    [0.2,  0.05, 0.4,  0.37],
    [0.12, 0.15, 0.33, 0.45],
    [0.25, 0.1,  0.3,  0.35],
    [0.0,  0.1,  0.4,  0.5 ],
    [0.15, 0.2,  0.3,  0.37],
    [0.0,  0.13, 0.4,  0.49],
    [0.22, 0.07, 0.4,  0.38],
    [0.2,  0.18, 0.3,  0.4 ]
]
```

We will work out the first iteration of the k-means algorithm supposing that there are $k = 3$ clusters in the data.

The first step is to randomly divide the data into $k = 3$ clusters. To do this, we can simply add an extra column in our data that represents the cluster number, and count off cluster numbers $1, 2, 3, 1, 2, 3$, and so on. We will put the extra column at the beginning of the data set.

```
[
    [1, 0.14, 0.14, 0.28, 0.44],
    [2, 0.22, 0.1,  0.45, 0.33],
    [3, 0.1,  0.19, 0.25, 0.4 ],
    [1, 0.02, 0.08, 0.43, 0.45],
    [2, 0.16, 0.08, 0.35, 0.3 ],
    [3, 0.14, 0.17, 0.31, 0.38],
    [1, 0.05, 0.14, 0.35, 0.5 ],
    [2, 0.1,  0.21, 0.28, 0.44],
    [3, 0.04, 0.08, 0.35, 0.47],
    [1, 0.11, 0.13, 0.28, 0.45],
    [2, 0.0,  0.07, 0.34, 0.65],
    [3, 0.2,  0.05, 0.4,  0.37],
    [1, 0.12, 0.15, 0.33, 0.45],
    [2, 0.25, 0.1,  0.3,  0.35],
    [3, 0.0,  0.1,  0.4,  0.5 ],
    [1, 0.15, 0.2,  0.3,  0.37],
    [2, 0.0,  0.13, 0.4,  0.49],
    [3, 0.22, 0.07, 0.4,  0.38],
    [1, 0.2,  0.18, 0.3,  0.4 ]
]
```

So, our initial guesses for the clusters are as follows:

```
# Cluster 1
[
    [1, 0.14, 0.14, 0.28, 0.44],
    [1, 0.02, 0.08, 0.43, 0.45],
    [1, 0.05, 0.14, 0.35, 0.5 ],
    [1, 0.11, 0.13, 0.28, 0.45],
    [1, 0.12, 0.15, 0.33, 0.45],
    [1, 0.15, 0.2,  0.3,  0.37],
    [1, 0.2,  0.18, 0.3,  0.4 ]
]

# Cluster 2
[
    [2, 0.22, 0.1,  0.45, 0.33],
    [2, 0.16, 0.08, 0.35, 0.3 ],
    [2, 0.1,  0.21, 0.28, 0.44],
    [2, 0.0,  0.07, 0.34, 0.65],
    [2, 0.25, 0.1,  0.3,  0.35],
    [2, 0.0,  0.13, 0.4,  0.49]
]

# Cluster 3
[
    [3, 0.1,  0.19, 0.25, 0.4 ],
    [3, 0.14, 0.17, 0.31, 0.38],
    [3, 0.04, 0.08, 0.35, 0.47],
    [3, 0.2,  0.05, 0.4,  0.37],
    [3, 0.0,  0.1,  0.4,  0.5 ],
    [3, 0.22, 0.07, 0.4,  0.38]
]
```

To compute each cluster center, we take the mean of each component of the data (ignoring the first component, which is the cluster label and was not part of the original data set).

```
# Cluster 1 center
[
    (0.14 + 0.02 + 0.05 + 0.11 + 0.12 + 0.15 + 0.2 ) /
        7,
    (0.14 + 0.08 + 0.14 + 0.13 + 0.15 + 0.2  + 0.18) /
        7,
    (0.28 + 0.43 + 0.35 + 0.28 + 0.33 + 0.3  + 0.3 ) /
        7,
    (0.44 + 0.45 + 0.5  + 0.45 + 0.45 + 0.37 + 0.4 ) / 7
]

# Cluster 2 center
[
    (0.22 + 0.16 + 0.1  + 0.0  + 0.25 + 0.0 ) / 6,
    (0.1  + 0.08 + 0.21 + 0.07 + 0.1  + 0.13) / 6,
    (0.45 + 0.35 + 0.28 + 0.34 + 0.3  + 0.4 ) / 6,
    (0.33 + 0.3  + 0.44 + 0.65 + 0.35 + 0.49) / 6
]

# Cluster 3 center
[
    (0.1  + 0.14 + 0.04 + 0.2  + 0.0 + 0.22) / 6,
    (0.19 + 0.17 + 0.08 + 0.05 + 0.1 + 0.07) / 6,
    (0.25 + 0.31 + 0.35 + 0.4  + 0.4 + 0.4 ) / 6,
    (0.4  + 0.38 + 0.47 + 0.37 + 0.5 + 0.38) / 6
]
```

Carrying out the above computations, we get the following results (rounded to 3 decimal places for readability):

```
# Cluster 1 center
[0.113, 0.146, 0.324, 0.437]

# Cluster 2 center
[0.122, 0.115, 0.353, 0.427]

# Cluster 3 center
[0.117, 0.110, 0.352, 0.417]
```

Once we've computed each cluster center, we then loop through the data and re-assign each data point to the nearest cluster center. We will use the Euclidean distance when determining the nearest cluster center.

```
Data point: [0.14, 0.14, 0.28, 0.44]
Distance
    from cluster 1 center: 0.052 <-- nearest
    from cluster 2 center: 0.080
    from cluster 3 center: 0.085

Data point: [0.22, 0.1, 0.45, 0.33]
Distance
    from cluster 1 center: 0.202
    from cluster 2 center: 0.169
    from cluster 3 center: 0.167 <-- nearest

Data point: [0.1, 0.19, 0.25, 0.4]
Distance
    from cluster 1 center: 0.095 <-- nearest
    from cluster 2 center: 0.132
    from cluster 3 center: 0.132

...

Data with re-assigned clusters:
[
    [1, 0.14, 0.14, 0.28, 0.44],
    [3, 0.22, 0.1,  0.45, 0.33],
    [1, 0.1,  0.19, 0.25, 0.4 ],
    ...
]
```

Interpreting the Clusters

If you repeat this process over and over, the cluster labels will eventually stop changing, indicating that every data point is assigned to the nearest cluster. In this example, it should be straightforward to interpret the meaning of your final clusters:

1. One cluster represents cookies with a greater proportion of sugar. These might be sugar cookies.

2. Another cluster represents cookies with a greater proportion of butter. These might be shortbread cookies.

3. Another cluster represents cookies with a greater proportion of eggs. These might be fortune cookies.

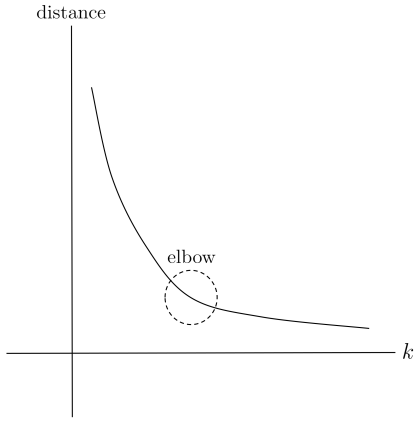
Elbow Method

Finally, remember that k (the number of clusters that we assume) is an input parameter to the algorithm. Because we usually don't know the number of clusters in the data beforehand, it's helpful to graph the "cohesiveness" of the clusters versus the value of k. We can measure cohesiveness by computing the total sum of distances between points and their cluster centers (the smaller the total distance, the more cohesive the clusters).

The graph of total distance versus k will be decreasing: the more clusters we assume, the closer the points will be to their clusters. In the extreme case where we set k equal to the number of points in our data set, it's possible that each point could be assigned to a separate cluster, resulting in a total distance of 0 but providing us with absolutely no information about groups of similar records in the data.

To choose k, it's common to use the **elbow method** and roughly estimate where the graph forms an "elbow," i.e. exhibits maximum curvature. This represents the point of diminishing returns, meaning

that assuming a greater number of clusters in the data will not make the clusters that much more cohesive.



Exercises

First, implement the example that was worked out above and interpret the resulting clusters. Then, generate a plot of total distance versus k and identify the elbow in the graph. Was $k = 3$ a good choice for the number of clusters in our data set? In other words, is the elbow of the graph near $k = 3$?

17. Tic-Tac-Toe and Connect Four

One of the best ways to get practice with object-oriented programming is implementing games. Tic-tac-toe and connect four are great for learning the ropes. Later on, we will also use these implementations for developing AI players.

Exercise: Tic-Tac-Toe with Random Players

Develop a tic-tac-toe game in which two random players play against each other. You can implement your game however you want, provided that you adhere to the constraints below.

- There should be a `Game` class and a `RandomPlayer` class. The game should be initialized via `game = Game(player1, player2)`, where `player1` and `player2` are both instances of `RandomPlayer` and `player1` moves first.

- The `Game` should have an attribute `game.board`, and there should be a method `player.choose_move(self, board)` in the `Player` class that takes a copy of the tic-tac-toe board as input and returns a random (but legal) move as output.
- Players should NOT actually update the board themselves – otherwise, they could cheat by changing the board in any way they please. Rather, the game should ask the player what move it chooses, and then the game should update its own board (provided that it's a legal move). If a player attempts to make an illegal move, then the game should skip that player's turn.
- You should be able to run an entire game via `game.run(log)`. If `log == true`, then the game should print out the sequence of board states and player moves (as well as whether or not the move was legal). *Be sure to implement logging as soon as you start coding up the game, because printing out logs will save you a lot of time debugging.*

Exercise: Tic-Tac-Toe with Manual Player

Then, create a `ManualPlayer` class that allows you to play manually via the command line. You can use Python's built-in `input()` function for this.

```
player1 = RandomPlayer()
player2 = ManualPlayer()

game = Game(player1, player2)
game.run()
```

Be sure to test your game by manually playing a handful of games against the random player. (Don't try to win every game – you'll need to tie and lose some games for testing purposes.)

Exercise: Strategy Functions

Currently, you have two types of tic-tac-toe players: `RandomPlayer` and `ManualPlayer`. The only difference between these players is in how they choose moves. The rest of the code is duplicated, which is not ideal. There should really be just one `Player` class, where the `choose_move` method is automatically adjusted as desired.

To make your code cleaner, implement a single `Player` class that is initialized via `player = Player(strategy_function)` where `strategy_function(board)` is a function that takes a copy of the tic-tac-toe board as input and returns a random move as output. Then `player.choose_move(self, board)` will simply call the `strategy_function` on the board and return the result.

```
player1 = Player(random_strategy_function)
player2 = Player(manual_strategy_function)

game = Game(player1, player2)
game.run()
```

Once you've implemented this, test your game again by manually playing a handful of games against the random player. Additionally, make sure that you are always able to beat the "cheater" strategy shown below. (If the cheater strategy wins, then it probably means that you're allowing the player to access the actual game board instead of giving it a *copy* of the board.)

```
def cheater_strategy_function(board):

    # put our own pieces everywhere on the board
    for i in range(3):
        for j in range(3):
            board[i][j] = our own piece

    # doesn't really matter what we return;
    # we'll arbitrarily move into top-left corner
    return (0,0)
```

Exercise: Custom Strategy

Create a custom strategy that beats the random player most of the time.

Exercise: Connect Four

Repeat the above exercises for the game of connect four. There are really only two differences:

1. A player chooses a column to place their piece into, rather than an actual board space. So, a move will be a single integer rather than a tuple.
2. Checking whether a player has won is more complicated.

18. Euler Estimation

Arrays can be used to implement more than just matrices. We can also implement other mathematical procedures like Euler estimation. We will jump straight to exercises – it is assumed that you're already familiar with Euler estimation from calculus.

Exercise: Single-Variable Euler Estimator

To start, build a single-variable Euler estimation class as follows:

```
>>> def derivative(t):
        return t+1

>>> euler = EulerEstimator(derivative)

>>> initial_point = (1,4)
>>> euler.eval_derivative(initial_point) # evaluates
    derivative at point (1,4)
2

>>> step_size = 0.5
>>> num_steps = 4
>>> euler.estimate_points(initial_point, step_size,
    num_steps)
[
    (1,   4   ),   # starting point
    (1.5, 5   ),   # after 1st step
    (2,   6.25),   # after 2nd step
    (2.5, 7.75),   # after 3rd step
    (3,   9.5 )    # after 4th step
]
```

Then, use your Euler estimator to plot several solution curves to the following differential equation on the interval $x \in [0, 5]$. (Your Euler estimator generates a list of points, and then you can use that list of points to generate a plot.)

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x - 2$$

For one curve, use the initial condition $y(0) = -2$. For another curve, use $y(0) = -1$. Then another curve with $y(0) = 0$, another with $y(0) = 1$, and another with $y(0) = 2$. All 5 of these curves can go on the same plot.

Based on your knowledge of calculus, you should be able to tell if your plots look right.

Exercise: Multivariable Euler Estimator

Once you've implemented a single-variable Euler estimator, you can generalize it to simulate systems of differential equations. For example, consider the following system:

$$\begin{aligned} a'(t) &= a(t) + 1 \\ b'(t) &= a(t) + b(t) \\ c'(t) &= 2b(t) + 3t \end{aligned}$$

To simulate this system starting with the initial state $a(-0.4) = -0.45$, $b(-0.4) = -0.05$, $c(-0.4) = 0$, construct a multivariable Euler estimator as follows:

```
>>> initial_state = {'a': -0.45, 'b': -0.05, 'c': 0}

>>> initial_point = (-0.4, initial_state) # points take
    form (t, state)

>>> def da_dt(t, state):
        return state['a'] + 1

>>> def db_dt(t, state):
        return state['a'] + state['b']

>>> def dc_dt(t, state):
        return 2 * state['b'] + 3 * t

>>> derivatives = {
    'a': da_dt,
    'b': db_dt,
    'c': dc_dt
}

>>> euler = EulerEstimator(derivatives)

>>> euler.eval_derivative_at_point(initial_point)
{'a': 0.55, 'b': -0.5, 'c': -1.3}

>>> step_size = 2
>>> num_steps = 3
>>> euler.estimate_points(initial_point, step_size,
    num_steps)
[
   (-0.4, {'a': -0.45, 'b': -0.05, 'c': 0   }),
   (1.6,  {'a': 0.65,  'b': -1.05, 'c': -2.6}),
   (3.6,  {'a': 3.95,  'b': -1.85, 'c': 2.8 }),
   (5.6,  {'a': 13.85, 'b': 2.35,  'c': 17  })
]
```

19. SIR Model For the Spread of Disease

One of the simplest ways to model the spread of disease using differential equations is the **SIR model**. Let's construct a SIR model and use our multivariable Euler estimator to plot the solution curves.

The SIR model assumes three sub-populations: susceptible, infected, and recovered.

- The number of susceptible people (S) decreases at a rate proportional to the rate of meeting between susceptible and infected people (because susceptible people have a chance of catching the disease when they come in contact with infected people).
- The number of infected people (I) increases at a rate proportional to the rate of meeting between susceptible and infected people (because susceptible people become infected after catching the disease), and decreases at a rate proportional to the number of infected people (as the diseased people recover).

- The number of recovered people (R) increases at a rate proportional to the number of infected people (as the diseased people recover).

Exercise: SIR Model Equations and Initial Conditions

First, write a system of differential equations using the following assumptions:

- There are initially 1000 susceptible people and 1 infected person.
- The number of meetings between susceptible and infected people each day is proportional to the product of the numbers of susceptible and infected people, by a factor of 0.01. The transmission rate of the disease is 3%. (In other words, 3% of meetings result in transmission.)
- Each day, 2% of infected people recover.

$$\begin{cases} \dfrac{\mathrm{d}S}{\mathrm{d}t} = \_\_\_, & S(0) = \_\_\_ \\ \dfrac{\mathrm{d}I}{\mathrm{d}t} = \_\_\_, & I(0) = \_\_\_ \\ \dfrac{\mathrm{d}R}{\mathrm{d}t} = \_\_\_, & R(0) = \_\_\_ \end{cases}$$

If you've written the system correctly, then at $t = 0$ you should have

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -0.3, \quad \frac{\mathrm{d}I}{\mathrm{d}t} = 0.28, \quad \frac{\mathrm{d}R}{\mathrm{d}t} = 0.02\,.$$

Exercise: SIR Model Simulation

Then use your multivariable Euler estimator to simulate the solution curve, and plot the result.

- Be sure to choose the step size small enough that the simulation doesn't blow up, but large enough that it doesn't take long to run.
- Likewise, choose an interval that displays all the main features of the differential equation, i.e. an interval that shows the behavior of the curves until they start to asymtpote off.

If your plot is correct, then you should be able to easily describe what is happening in the plot and why it is happening.

20. Hodgkin-Huxley Model of Action Potentials in Neurons

In 1952, Alan Hodgkin and Andrew Huxley published a differential equation model of "spikes" (i.e. "action potentials") in the voltage of neurons. For this work, they received the 1963 Nobel Prize in Physiology or Medicine (shared with Sir John Carew Eccles).

Below, we summarize the key points of the model so that you may implement and simulate it yourself. The primary source is the original paper, *A quantitative description of membrane current and its application to conduction and excitation in nerve.*

For background information, it is recommended to watch the following short videos:

1. *Neurons or nerve cells - Structure function and types of neurons* by Elearnin
   (`https://www.youtube.com/watch?v=cUGuWh2UeMk`)
2. *2-Minute Neuroscience: Action Potential* by Neuroscientifically Challenged
   (`https://www.youtube.com/watch?v=W2hHt_PXe5o`)

Idea 1: Start with Physics Fundamentals

From physics, we know that current is proportional to voltage by a constant C called the capacitance:

$$I = C\frac{\mathrm{d}V}{\mathrm{d}t}$$

So, the voltage of a neuron can be modeled as

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{I}{C}.$$

For neurons, we have $C \approx 1.0$.

Idea 2: Decompose Current into Four Subcurrents (Stimulus and Ion Channels)

The current I consists of

- a stimulus s to the neuron (from an electrode or other neurons),
- current flux across sodium and potassium ion channels (I_{Na} and I_{K}), and
- current leakage, treated as a channel I_{L}.

The stimulus increases the voltage, while the flux and leakage decrease it. So, we have

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{1}{C}\left[s - I_{\mathrm{Na}} - I_{\mathrm{K}} - I_{\mathrm{L}}\right].$$

Idea 3: Model the Ion Channel Currents

The current across an ion channel is proportional to the voltage difference, relative to the equilibrium voltage of that channel:

$$\begin{aligned} I_{\mathrm{Na}}(V, m, h) &= g_{\mathrm{Na}}(m, h)\,(V - V_{\mathrm{Na}}) & V_{\mathrm{Na}} &\approx 115 \\ I_{\mathrm{K}}(V, n) &= g_{\mathrm{K}}(n)\,(V - V_{\mathrm{K}}) & V_{\mathrm{K}} &\approx -12 \\ I_{\mathrm{L}}(V) &= g_{\mathrm{L}} \cdot (V - V_{\mathrm{L}}) & V_{\mathrm{L}} &\approx 10.6 \end{aligned}$$

The constants of proportionality are conductances, which were modeled experimentally:

$$\begin{aligned} g_{\mathrm{Na}}(m, h) &= \overline{g}_{\mathrm{Na}} m^3 h & \overline{g}_{\mathrm{Na}} &\approx 120 \\ g_{\mathrm{K}}(n) &= \overline{g}_{\mathrm{K}} n^4 & \overline{g}_{\mathrm{K}} &\approx 36 \\ g_{\mathrm{L}} &= \overline{g}_{\mathrm{L}} & \overline{g}_{\mathrm{L}} &\approx 0.3, \end{aligned}$$

where

$$\begin{aligned}\frac{\mathrm{d}n}{\mathrm{d}t} &= \alpha_n(V)(1-n) - \beta_n(V)n \\ \frac{\mathrm{d}m}{\mathrm{d}t} &= \alpha_m(V)(1-m) - \beta_m(V)m \\ \frac{\mathrm{d}h}{\mathrm{d}t} &= \alpha_h(V)(1-h) - \beta_h(V)h.\end{aligned}$$

and

$$\begin{aligned}\alpha_n(V) &= \frac{0.01(10-V)}{\exp\left[0.1(10-V)\right]-1} & \beta_n(V) &= 0.125\exp\left[-\frac{V}{80}\right] \\ \alpha_m(V) &= \frac{0.1(25-V)}{\exp\left[0.1(25-V)\right]-1} & \beta_m(V) &= 4\exp\left[-\frac{V}{18}\right] \\ \alpha_h(V) &= 0.07\exp\left[-\frac{V}{20}\right] & \beta_h\,(V) &= \frac{1}{\exp\left[0.1(30-V)\right]+1}.\end{aligned}$$

Exercise

Your task is to implement the Hodgkin-Huxley neuron model using Euler estimation. You can represent the state of the neuron at time t using

$$\Big(t, (V, n, m, h)\Big),$$

and you can approximate the initial values by setting $V_0 = 0$ and setting n, m, and h equal to their asymptotic values for $V_0 = 0$:

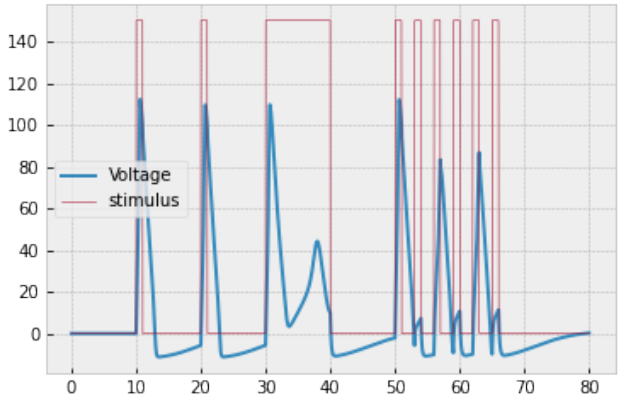
$$n_0 = \frac{\alpha_n(V_0)}{\alpha_n(V_0) + \beta_n(V_0)}$$
$$m_0 = \frac{\alpha_m(V_0)}{\alpha_m(V_0) + \beta_m(V_0)}$$
$$h_0 = \frac{\alpha_h(V_0)}{\alpha_h(V_0) + \beta_h(V_0)}$$

(When we take $V_0 = 0$, we are letting V represent the voltage *offset* from the usual resting potential.)

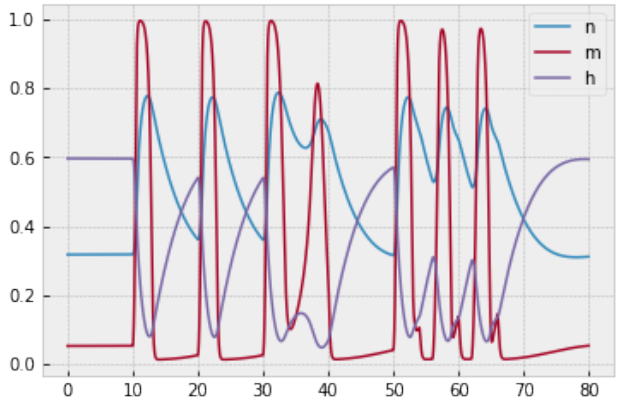
Simulate the system for $t \in [0, 80\,\text{ms}]$ with step size $\Delta t = 0.01$ and stimulus

$$s(t) = \begin{cases} 150, & t \in [10, 11] \cup [20, 21] \cup [30, 40] \cup [50, 51] \cup [53, 54] \\ & \quad \cup [56, 57] \cup [59, 60] \cup [62, 63] \cup [65, 66] \\ 0 & \text{otherwise.} \end{cases}$$

You should get the following result:



The corresponding plot of n, m, and h is provided to help you debug:



Lastly, here is an incomplete code template to get you started:

```
################################
### constants

V_0 = ...
n_0 = ...
m_0 = ...
h_0 = ...

C = 1.0
V_Na = 115
...

################################
### main variables: V, n, m, h

def dV_dt(t,x):
    ...

def dn_dt(t,x):
    n = x['n']
    return alpha_n(t,x) * (1-n) - beta_n(t,x) * n
```

```
def dm_dt(t,x):
    ...

def dh_dt(t,x):
    ...

################################
### intermediate variables: alphas, betas, stimulus (s),
     currents (I's), ...

def alpha_n(t,x):
    ...

def beta_n(t,x):
    ...

...

################################
### input into EulerEstimator

derivatives = {
    'V': dV_dt,
    'n': dn_dt,
    ...
}

initial_point = ...
```

21. Hash Tables

Under the hood, dictionaries are hash tables.

The most elementary (and inefficient) version of a hash table would be a list of tuples. For example, if we wanted to implement the dictionary

```
{
    'a': [0 ,1],
    'b': 'abcd',
    'c': 3.14
}
```

then we'd have the following list of tuples:

```
[
    ('a', [0, 1]),
    ('b', 'abcd'),
    ('c', 3.14  )
]
```

To add a new key-value pair to the dictionary, we'd just append the corresponding tuple, and to look up the value for some key, we'd just

loop through the tuples until we get to the tuple with the key we wanted (and then we'd return the corresponding value).

Exercise: Buckets and Hash Functions

Searching through a long array is slow. So, to be more efficient, we use several lists of tuples. We call those lists **buckets**, and we use a **hash function** to tell us which bucket to put the new key-value pair in.

Complete the code below to implement a special case of an elementary hash table. We'll expand on this example soon, but let's start with something simple.

```
array = [[], [], [], [], []] # has 5 empty buckets

def hash(string):
    # Return the sum of character indices in the string
    # (where 'a' has index 0, 'b' has index 1, ..., 'z'
        has index 25)
    # modulo 5.

    # We'll assume the string consists of lowercase
    # letters with no other characters or spaces.

def insert(array, key, value):
    # Apply the hash function to the key to get the
        bucket index.
    # then append the (key, value) pair to the bucket.

def find(array, key):
    # Apply the hash function to the key to get the
        bucket index.
    # Then, loop through the bucket until you get to the
         tuple with
    # the desired key, and return the corresponding
        value.
```

Here's an example of how the hash table will work:

```
>>> print(array)
array = [[], [], [], [], []]

>>> insert(array, 'a', [0,1])
>>> insert(array, 'b', 'abcd')
>>> insert(array, 'c', 3.14)
>>> print(array)
[
    [('a', [0, 1])],
    [('b', 'abcd')],
    [('c', 3.14  )],
    [],
    []
]

>>> insert(array, 'd', 0)
>>> insert(array, 'e', 0)
>>> insert(array, 'f', 0)
>>> print(array)
[
    [('a', [0, 1]), ('f', 0)],
    [('b', 'abcd')],
    [('c', 3.14  )],
    [('d', 0     )],
    [('e', 0     )]
]
Test your code as follows:

alphabet = 'abcdefghijklmnopqrstuvwxyz'
for i, char in enumerate(alphabet):
    key = 'someletters'+char
    value = [i, i**2, i**3]
    insert(array, key, value)

for i, char in enumerate(alphabet):
    key = 'someletters'+char
    output_value = find(array, key)
    desired_value = [i, i**2, i**3]
    assert output_value == desired_value
```

Exercise: Hash Table

Write a class `HashTable` that generalizes the hash table you previously wrote. This class should store an array of buckets, and the hash function should add up the alphabet indices of the input string and mod the result by the number of buckets.

```
>>> ht = HashTable(num_buckets = 3)
>>> ht.buckets
[[], [], []]
>>> ht.hash('cabbage')
2    (because 2+0+1+1+0+6+4 mod 3 = 14 mod 3 = 2)

>>> ht.insert('cabbage', 5)
>>> ht.buckets
[
    [],
    [],
    [('cabbage', 5)]
]

>>> ht.insert('cab', 20)
>>> ht.buckets
[
    [('cab',    20)],
    [],
    [('cabbage', 5)]
]

>>> ht.insert('c', 17)
>>> ht.buckets
[
    [('cab',    20)],
    [],
    [('cabbage', 5), ('c', 17)]
]

>>> ht.insert('ac', 21)
>>> ht.buckets
```

```
[
    [('cab',    20)],
    [],
    [('cabbage', 5), ('c', 17), ('ac', 21)]
]

>>> ht.find('cabbage')
5
>>> ht.find('cab')
20
>>> ht.find('c')
17
>>> ht.find('ac')
21
```

22. Simplex Method

The **simplex method** is a technique for maximizing linear expressions subject to linear constraints. Many applied problems can be framed like this – for example, a business may have a variety of raw materials and need to determine the most profitable inventory of products that they can produce from those materials. This would ultimately amount to maximizing a linear expression for profit, subject to a linear inequality for each type of raw material (the total amount of raw material used in the products produced must be less than or equal to the amount of raw material available).

At its core, the simplex method is just algebra that can be implemented elegantly using array manipulations. Let's work through the algebra on an example:

$$\begin{aligned} &\text{maximize } x_1 + 2x_2 + x_3 \text{ such that} \\ 2x_1 + x_2 + x_3 &\leq 14 \\ 4x_1 + 2x_2 + 3x_3 &\leq 28 \\ 2x_1 + 5x_2 + 5x_3 &\leq 30 \\ x_1,\, x_2,\, x_3 &\geq 0 \end{aligned}$$

Step 1: Introduce Slack Variables

Math is generally easier when we work with equations (rather than inequalities). We can convert most of the system above into equations if we move everything to the RHS and then assign those RHS expressions to new variables called *slack variables*.

$$\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&0 \leq 14 - 2x_1 - x_2 - x_3 \\
&0 \leq 28 - 4x_1 - 2x_2 - 3x_3 \\
&0 \leq 30 - 2x_1 - 5x_2 - 5x_3 \\
&0 \leq x_1,\ x_2,\ x_3
\end{aligned}$$

$$\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&x_4 = 14 - 2x_1 - x_2 - x_3 \\
&x_5 = 28 - 4x_1 - 2x_2 - 3x_3 \\
&x_6 = 30 - 2x_1 - 5x_2 - 5x_3 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}$$

The slack variables that we created are x_4, x_5, x_6. Note that the original system is entirely equivalent to the definition of the slack variables. The system involving slack variables is exactly the same as the original system, but written a little differently. (This is analogous to change of variables in integration.)

Step 2: Evaluate the Objective Quantity

Each system in the above form is associated with a "guess" that comes from setting all the *basic* variables equal to 0. The basic variables are the variables that appear in the expressions for other *non-basic* variables.

In our case, the basic variables are x_1, x_2, x_3 because they appear in the quantity we're trying to maximize and also in the expressions for the non-basic variables x_4, x_5, x_6. You can also remember that the basic variables are the RHS and the non-basic variables are the LHS.

When we set the basic variables equal to 0, we plug

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 0 \\ x_3 &= 0 \end{aligned}$$

into the quantity that we're trying to maximize (known as the *objective quantity*) and get

$$\begin{aligned} & x_1 + 2x_2 + x_3 \\ & \to 0 + 2(0) + 0 \\ & \to 0. \end{aligned}$$

So currently, we have a guess $x_1 = x_2 = x_3 = 0$ and this brings the objective quantity to 0. This is a pretty bad guess, but we're going to repeatedly improve it until it's optimal.

Step 3: Identify the Basic Variable with the Greatest Slope

We're going to improve upon our guess by repeatedly increasing the basic variable that will most quickly improve our guess.

To improve our guess, we need to increase the objective quantity $x_1 + 2x_2 + x_3$, which we write as a function below:

$$M(x_1, x_2, x_3) = x_1 + 2x_2 + x_3$$

The basic variable that will most quickly improve our guess is the one that will make the objective quantity increase the fastest (when we increase that variable). In other words, it is the one with the greatest slope.

$$\text{slope}(x_1) = 1, \quad \text{slope}(x_2) = 2, \quad \text{slope}(x_3) = 1$$

Here, the basic variable with the largest slope is x_2. So this what we will increase.

Step 4: Identify which Non-Basic Variable to Trade for that Basic Variable

The basic variable with the largest slope is x_2. We will now trade this for one of the non-basic variables (x_4, x_5, x_6).

For our next guess, we want to increase x_2 as much as possible (since doing so will increase our objective quantity the fastest). But we can't increase it too much – remember that the system has constraints, and we can't invalidate those constraints.

Each of the constraints of the original system was converted into a non-basic variable. So, we need to identify the non-basic variable that places the strictest constraint on x_2, and then trade it with x_2 (i.e. make that variable basic in exchange for making x_2 non-basic).

Here is where we are so far:

$$
\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&x_4 = 14 - 2x_1 - x_2 - x_3 \\
&x_5 = 28 - 4x_1 - 2x_2 - 3x_3 \\
&x_6 = 30 - 2x_1 - 5x_2 - 5x_3 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}
$$

Since $0 \leq x_4, x_5, x_6$, we have the following constraints:

$$
\begin{aligned}
0 &\leq x_4 = 14 - 2x_1 - x_2 - x_3 \\
0 &\leq x_5 = 28 - 4x_1 - 2x_2 - 3x_3 \\
0 &\leq x_6 = 30 - 2x_1 - 5x_2 - 5x_3
\end{aligned}
$$

Let's simplify a bit:

$$
\begin{aligned}
0 &\leq 14 - 2x_1 - x_2 - x_3 \\
0 &\leq 28 - 4x_1 - 2x_2 - 3x_3 \\
0 &\leq 30 - 2x_1 - 5x_2 - 5x_3
\end{aligned}
$$

Now, let's set move x_2 to the LHS of each constraint and set the other basic variables (x_1, x_3) equal to zero so that we can see which constraint is strictest:

$$
\begin{aligned}
x_2 &\leq 14 \\
x_2 &\leq 14 \\
x_2 &\leq 6
\end{aligned}
$$

The strictest constraint is the constraint that places the lowest non-negative upper bound on x_2. Here, the third constraint is the strictest, and if you look at the work backwards, you'll see that it came from the non-basic variable x_6:

$$
\begin{aligned}
x_6 &= 30 - 2x_1 - 5x_2 - 5x_3 \\
&\updownarrow \\
0 &\leq 30 - 2x_1 - 5x_2 - 5x_3 \\
&\updownarrow \\
x_2 &\leq 6
\end{aligned}
$$

So, we need to trade the non-basic variable x_6 for the basic variable x_2.

Step 5: Execute the Trade

This step will feel more familiar than the earlier steps. We have an equation relating x_6 and x_2:

$$x_6 = 30 - 2x_1 - 5x_2 - 5x_3$$

All we need to do is solve this equation for x_2 and then substitute that into our system.

Solving for x_2, we get

$$x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6.$$

Then, we substitute into our system. The only catch is that we need to fully replace the $x_6 =$ equation with the $x_2 =$ equation. (This equation is supposed to capture the relationship between x_2 and x_6, and if we just substituted x_2 in the RHS, then it would simplify to a useless equation $x_6 = x_6$).

$$\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&x_4 = 14 - 2x_1 - x_2 - x_3 \\
&x_5 = 28 - 4x_1 - 2x_2 - 3x_3 \\
&x_6 = 30 - 2x_1 - 5x_2 - 5x_3 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}$$

$$\begin{aligned}
&\text{maximize } x_1 + 2\left(6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6\right) + x_3 \\
&x_4 = 14 - 2x_1 - \left(6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6\right) - x_3 \\
&x_5 = 28 - 4x_1 - 2\left(6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6\right) - 3x_3 \\
&x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}$$

$$
\begin{aligned}
&\text{maximize } 12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6 \\
&x_4 = 8 - \frac{8}{5}x_1 + \frac{1}{5}x_6 \\
&x_5 = 16 - \frac{16}{5}x_1 - x_3 + \frac{2}{5}x_6 \\
&x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6 \\
&0 \leq x_1,\, x_2,\, x_3,\, x_4,\, x_5,\, x_6
\end{aligned}
$$

Just to keep things tidy, we'll re-order the equations so that the LHS variables are sorted:

$$
\begin{aligned}
&\text{maximize } 12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6 \\
&x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6 \\
&x_4 = 8 - \frac{8}{5}x_1 + \frac{1}{5}x_6 \\
&x_5 = 16 - \frac{16}{5}x_1 - x_3 + \frac{2}{5}x_6 \\
&0 \leq x_1,\, x_2,\, x_3,\, x_4,\, x_5,\, x_6
\end{aligned}
$$

Step 6: Repeat Steps 2-5 Until No Slopes Are Positive

Evaluate the objective quantity. We will evaluate the objective quantity for our new guess. Remember that our guess is obtained by setting the

basic variables equal to 0. The basic variables are now x_1, x_3, x_6. Setting these variables equal to 0, our objective quantity becomes

$$\begin{aligned}
&12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6 \\
&\rightarrow 12 + \frac{1}{5}(0) - 2(0) - \frac{2}{5}(0) \\
&\rightarrow 12.
\end{aligned}$$

So our guess has improved! Our previous guess gave us an objective quantity of 0, and our new guess gave us an objective quantity of 12.

Sanity Checks

In this example, if you ever get a new guess that does not appear to increase the objective quantity, then something went wrong. The new guess should *always* increase the objective quantity. (In rare cases, it's possible for the simplex algorithm to "cycle" around suboptimal minima that yield the same value of the objective function, but this example is not one of those cases.)

Additionally, you should always verify that your guess satisfies the constraints of the *original* problem statement. Below is an explanation of how to do that. As a reminder, the original problem statement was to maximize $x_1 + 2x_2 + x_3$ subject to the following constraints:

$$\begin{aligned} 2x_1 + x_2 + x_3 &\leq 14 \\ 4x_1 + 2x_2 + 3x_3 &\leq 28 \\ 2x_1 + 5x_2 + 5x_3 &\leq 30 \\ x_1,\, x_2,\, x_3 &\geq 0 \end{aligned}$$

The original problem statement is framed in terms of variables x_1, x_2, x_3. The slack variables x_4, x_5, x_6 are artificial things that we made up. Here, our guess is $x_1 = x_3 = x_6 = 0$. Although x_2 is not explicit in our guess, we can find it by substituting our guess into our system. Doing so, we find

$$x_2 = 6, \quad x_4 = 8, \quad x_5 = 16.$$

So in terms of the original variables of our system, our guess is

$$x_1 = 0, \quad x_2 = 6, \quad x_3 = 0.$$

Indeed, if we evaluate the objective quantity shown in the problem statement, we get

$$\begin{aligned} & x_1 + 2x_2 + x_3 \\ & \to 0 + 2(6) + 0 \\ & \to 12. \end{aligned}$$

Additionally, the guess $x_1 = 0,\ x_2 = 6,\ x_3 = 0$ satisfies all constraints in the original problem statement.

Another Iteration

Identify the basic variable with the largest slope. Our objective function is now

$$M(x_1, x_3, x_6) = 12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6,$$

and the basic variable with the largest slope is x_1.

Identify which non-basic variable to trade for that basic variable. With a bit of algebra work (which you should do on your own), we have the following constraints:

$$\begin{aligned} x_1 &\leq 15 \\ x_1 &\leq 5 \\ x_1 &\leq 5 \end{aligned}$$

The second and third constraints are the strictest. They are equally strict, so we can choose to proceed with either one. Referring back to our system (shown below), these two constraints come from x_4 and x_5.

$$
\begin{aligned}
&\text{maximize } 12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6 \\
&x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6 \\
&x_4 = 8 - \frac{8}{5}x_1 + \frac{1}{5}x_6 \\
&x_5 = 16 - \frac{16}{5}x_1 - x_3 + \frac{2}{5}x_6 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}
$$

Let's proceed with the x_4 constraint since it's simpler (it contains fewer variables).

Execute the trade. Solving for x_1 in terms of x_4 (which you should do on your own), we get

$$x_1 = 5 - \frac{5}{8}x_4 + \frac{1}{8}x_6$$

I'll leave it to you to finish the trade execution by substituting into your system.

In the next round, you should find that your guess has improved, but that no slopes are positive, which means you're done and have found the optimal solution.

Array Manipulations

The algebra above can be implemented more elegantly as array manipulations. Let's start by taking the system that we obtained after introducing slack variables, and converting it to an array:

$$\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&x_4 = 14 - 2x_1 - x_2 - x_3 \\
&x_5 = 28 - 4x_1 - 2x_2 - 3x_3 \\
&x_6 = 30 - 2x_1 - 5x_2 - 5x_3 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}$$

$$\begin{aligned}
&\text{maximize } x_1 + 2x_2 + x_3 \\
&2x_1 + x_2 + x_3 + x_4 = 14 \\
&4x_1 + 2x_2 + 3x_3 + x_5 = 28 \\
&2x_1 + 5x_2 + 5x_3 + x_6 = 30
\end{aligned}$$

	x_1	x_2	x_3	x_4	x_5	x_6	constant
maximize	1	2	1	0	0	0	0
constraint	2	1	1	1	0	0	14
constraint	4	2	3	0	1	0	28
constraint	2	5	5	0	0	1	30

From the array, we can tell that x_2 has the largest slope since it has the largest entry in the top row. To find the row that places the strictest constraint on x_2, we just divide the constant column by the x_2 column:

	x_1	x_2	x_3	x_4	x_5	x_6	constant	constraint
maximize	1	2	1	0	0	0	0	–
constraint	2	1	1	1	0	0	14	$14/1 = 14$
constraint	4	2	3	0	1	0	28	$28/2 = 14$
constraint	2	5	5	0	0	1	30	$30/5 = 6$

The bottom row places the strictest constraint. To execute the trade, we perform elementary row operations using the bottom row to kill off the rest of the x_2 column. (That is, we divide the bottom row so that the entry in the x_2 column becomes 1, and then we subtract multiples of the bottom row from the other rows.)

	x_1	x_2	x_3	x_4	x_5	x_6	constant	
maximize	0.2	0	-1	0	0	-0.4	-12	$\leftarrow$ max is 12
constraint	1.6	0	0	1	0	-0.2	8	
constraint	3.2	0	1	0	1	-0.4	16	
constraint	0.4	1	1	0	0	0.2	6	

This matches up with the system that we got after the first iteration when we worked out the algebra (shown below). The only catch is that the -12 in the "constant" column really means that we've reached a maximum of *positive* 12. (This happens because the constant column usually represents the constant once it's been moved to the other side

of the equality sign, but there is no equality sign in the expression that we're trying to maximize.)

$$\begin{aligned}
&\text{maximize } 12 + \frac{1}{5}x_1 - x_3 - \frac{2}{5}x_6 \\
&x_4 = 8 - \frac{8}{5}x_1 + \frac{1}{5}x_6 \\
&x_5 = 16 - \frac{16}{5}x_1 - x_3 + \frac{2}{5}x_6 \\
&x_2 = 6 - \frac{2}{5}x_1 - x_3 - \frac{1}{5}x_6 \\
&0 \leq x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6
\end{aligned}$$

Remember that the array manipulations are just implementing algebra that we've already worked through, so if you get confused or stuck when implementing the simplex method, it will help to go back to the algebra and make sure that every array manipulation you do matches up against the algebra.

Exercises

1. Work out the example that was covered, using algebra, on paper.
2. Write all the relevant array manipulations next to the algebra.
3. Implement the simplex method in code. The input should be the first array that you write down, and the output should be the last array that you wrote down.

4. Work out the array manipulations manually for the new system shown below, and then ensure that your implementation gives the same result. Note that when you are identifying the strictest constraint, if you ever get a constraint of that involves a comparison to a negative number, then you can discard it (since we already know the variables are non-negative). This should only require you to work out 3 iterations (i.e. your 4th guess will be the final one).

$$\begin{aligned}\text{maximize } 20x_1 + 10x_2 + 15x_3 \text{ such that}\\ 3x_1 + 2x_2 + 5x_3 &\leq 55\\ 2x_1 + x_2 + x_3 &\leq 26\\ x_1 + x_2 + 3x_3 &\leq 30\\ 5x_1 + 2x_2 + 4x_3 &\leq 57\\ x_1,\, x_2,\, x_3 &\geq 0.\end{aligned}$$

Part IV

Regression and Classification

23. Linear, Polynomial, and Multiple Linear Regression via Pseudoinverse

Supervised learning consists of fitting functions to data sets. The idea is that we have a sample of inputs and outputs from some process, and we want to come up with a mathematical model that predicts what the outputs would be for other inputs. Supervised *machine* learning involves programming computers to compute such models.

Linear Regression and the Pseudoinverse

One of the simplest ways to construct a predictive algorithm is to assume a linear relationship between the input and output. Even if this assumption isn't correct, it's always useful to start with a linear model because it provides a baseline against which we can compare the accuracy of more complicated models. (We can only justify using a complicated model if it is significantly more accurate than a linear model.)

For example, let's fit a linear model $y = mx+b$ to the following data set. That is, we want to find the values of m and b so that the line $y = mx+b$ most accurately represents the following data set.

$$[(0,1),(2,5),(4,3)]$$

While there is no single line that goes through all three points in the data set, we can choose the slope m and intercept b so that the line represents the data as accurately as possible. There are a handful of ways to accomplish this, the simplest of which involves leveraging the *pseudoinverse* from linear algebra.

To start, let's set up the system of equations that would need to be satisfied in order for our model to have perfect accuracy on the data set. We do this by simply taking each point (x,y) in our data set and plugging it into the model $y = mx + b$.

$$(x,y) \quad \rightarrow \quad y = m \cdot x + b$$

$$\begin{aligned}(0,1) \quad &\rightarrow \quad 1 = m \cdot 0 + b \\ (2,5) \quad &\rightarrow \quad 5 = m \cdot 2 + b \\ (4,3) \quad &\rightarrow \quad 3 = m \cdot 4 + b\end{aligned}$$

Now, we rewrite the above system using matrix notation:

$$\begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} m \cdot 0 + b \\ m \cdot 2 + b \\ m \cdot 4 + b \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

You may be tempted to solve the matrix equation by inverting the coefficient matrix that's multiplying the unknown:

$$\begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 4 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix}$$

Then you might remember that the inverse of a non-square matrix does not exist, and think that this was all fruitless.

But really, this is the main idea of the pseudoinverse method. The only difference is that before inverting the coefficient matrix, we multiply both sides of the equation by the transpose of the coefficient matrix so that it becomes square.

$$\begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 & 4 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

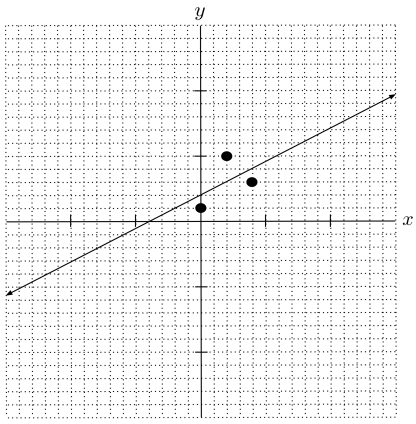
$$\begin{bmatrix} 22 \\ 9 \end{bmatrix} = \begin{bmatrix} 20 & 6 \\ 6 & 3 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

Geometrically, multiplying by the transpose takes the matrices involved in the equation and *projects* them onto the nearest subspace where a solution exists. This means we can now take the inverse:

$$\begin{aligned} \begin{bmatrix} m \\ b \end{bmatrix} &= \begin{bmatrix} 20 & 6 \\ 6 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 22 \\ 9 \end{bmatrix} \\ &= \frac{1}{24} \begin{bmatrix} 3 & -6 \\ -6 & 20 \end{bmatrix} \begin{bmatrix} 22 \\ 9 \end{bmatrix} \\ &= \frac{1}{24} \begin{bmatrix} 12 \\ 48 \end{bmatrix} \\ &= \begin{bmatrix} 1/2 \\ 2 \end{bmatrix} \end{aligned}$$

Reading off the parameters $m = \frac{1}{2}$ and $b = 2$, we have the following linear model:

$$y = \frac{1}{2}x + 2$$



Polynomial Regression

We can use the pseudoinverse method to fit polynomial models as well. To illustrate, let's fit a quadratic model $y = ax^2 + bx + c$ to the following data set:

$$[(0, 1), (2, 5), (4, 3), (5, 0)]$$

First, we set up our system of equations:

$$(x, y) \quad \rightarrow \quad y = a \cdot x^2 + b \cdot x + c$$

$$\begin{aligned}
(0, 1) \quad &\rightarrow \quad 1 = a \cdot 0^2 + b \cdot 0 + c \\
(2, 5) \quad &\rightarrow \quad 5 = a \cdot 2^2 + b \cdot 2 + c \\
(4, 3) \quad &\rightarrow \quad 3 = a \cdot 4^2 + b \cdot 4 + c \\
(5, 0) \quad &\rightarrow \quad 0 = a \cdot 5^2 + b \cdot 5 + c
\end{aligned}$$

Then we convert to a matrix equation, multiply both sides by the transpose of the coefficient matrix, and invert the resulting square matrix.

$$\begin{bmatrix} 1 \\ 5 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0^2 & 0 & 1 \\ 2^2 & 2 & 1 \\ 4^2 & 4 & 1 \\ 5^2 & 5 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\begin{bmatrix} 0^2 & 2^2 & 4^2 & 5^2 \\ 0 & 2 & 4 & 5 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0^2 & 2^2 & 4^2 & 5^2 \\ 0 & 2 & 4 & 5 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0^2 & 0 & 1 \\ 2^2 & 2 & 1 \\ 4^2 & 4 & 1 \\ 5^2 & 5 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\begin{bmatrix} 68 \\ 22 \\ 9 \end{bmatrix} = \begin{bmatrix} 897 & 197 & 45 \\ 197 & 45 & 11 \\ 45 & 11 & 4 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

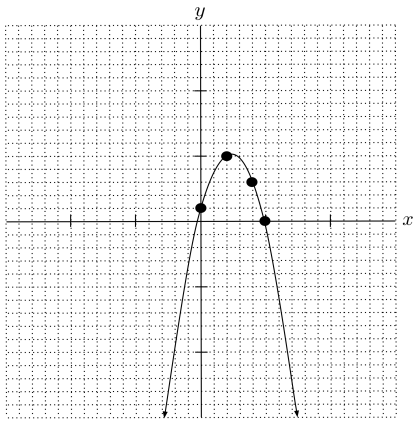
$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 897 & 197 & 45 \\ 197 & 45 & 11 \\ 45 & 11 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 68 \\ 22 \\ 9 \end{bmatrix}$$

$$\approx \begin{bmatrix} -0.73 \\ 3.42 \\ 1.02 \end{bmatrix}$$

Note that we used a computer to simplify the final step. You can start to see why computers are essential in machine learning (and this is just the tip of the iceberg).

We have the following quadratic model:

$$y \approx -0.73x^2 + 3.42x + 1.02$$



Multiple Linear Regression

Lastly, the pseudoinverse method can also be used to fit models consisting of multiple input variables. For example, let's fit a linear model $z = ax + by + c$ to the following data set:

$$[(0, 1, 50), (2, 5, 30), (4, 3, 20), (5, 1, 10)]$$

Here, we have two input variables, x and y. Our output variable is z.

As usual, we begin by setting up our system of equations:

$$(x, y, z) \quad \rightarrow \quad z = a \cdot x + b \cdot y + c$$

$$\begin{aligned}
(0, 1, 50) \quad &\rightarrow \quad 50 = a \cdot 0 + b \cdot 1 + c \\
(2, 5, 30) \quad &\rightarrow \quad 30 = a \cdot 2 + b \cdot 5 + c \\
(4, 3, 20) \quad &\rightarrow \quad 20 = a \cdot 4 + b \cdot 3 + c \\
(5, 0, 10) \quad &\rightarrow \quad 10 = a \cdot 5 + b \cdot 0 + c
\end{aligned}$$

Then we convert to a matrix equation, multiply both sides by the transpose of the coefficient matrix, and invert the square matrix that results.

$$\begin{bmatrix} 50 \\ 30 \\ 20 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 5 & 1 \\ 4 & 3 & 1 \\ 5 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 & 4 & 5 \\ 1 & 5 & 3 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 30 \\ 20 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 & 2 & 4 & 5 \\ 1 & 5 & 3 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 5 & 1 \\ 4 & 3 & 1 \\ 5 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

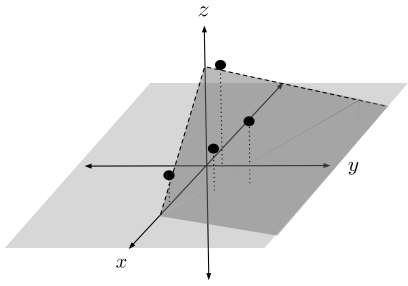
$$\begin{bmatrix} 190 \\ 260 \\ 110 \end{bmatrix} = \begin{bmatrix} 45 & 22 & 11 \\ 22 & 35 & 9 \\ 11 & 9 & 4 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 45 & 22 & 11 \\ 22 & 35 & 9 \\ 11 & 9 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 190 \\ 260 \\ 110 \end{bmatrix}$$

$$\approx \begin{bmatrix} -7.74 \\ -0.60 \\ 50.12 \end{bmatrix}$$

So, we have the following model which represents a plane in 3-dimensional space:

$$z \approx -7.74x - 0.60y + 50.12$$



When the Pseudoinverse Fails

Note that the pseudoinverse method requires that the columns of the coefficient matrix be independent. Otherwise, when we multiply by the transpose, the result is not guaranteed to be invertible. In particular, this means that the number of parameters of the model that we want to fit (i.e. the width of the coefficient matrix) should not exceed the number of distinct data points in our data set (i.e. the height of the coefficent matrix).

To illustrate what happens when the number of parameters of the model exceeds the number of distinct data points, let's trying to fit a line $y = mx + b$ to a data set $[(2, 3)]$ consisting of only a single point. (The line has 2 parameters, m and b.)

$$3 = m \cdot 2 + b$$

$$\begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 1 \end{bmatrix} \begin{bmatrix} 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$\begin{bmatrix} 6 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$\begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix}^{-1}$ does not exist

Although multiplying by the transpose gives us a square matrix, the square matrix is not invertible. This happens because there are infinitely many lines that have pass through the point $(2, 3)$ and therefore have perfect accuracy on the data set.

General Formula

Looking back at our work, we can write down the general procedure as follows, where y is the vector of outputs, X is the coefficients matrix of our system of equations, and p is the vector of model parameters.

$$\begin{aligned} y &= Xp \\ X^T y &= X^T X p \\ p &= (X^T X)^{-1} X^T y \end{aligned}$$

The *pseudoinverse* of the matrix X is defined by $(X^T X)^{-1} X^T$, as shown in the last row of the general equation. To understand why this quantity is called the pseudoinverse, first recall that if we attempt to solve the equation $y = Xp$ using the regular matrix, then we get a solution of the form

$$p = X^{-1} y.$$

The issue that we run into is that the regular inverse X^{-1} usually does not exist (because X is usually a tall rectangular matrix, not a square matrix). The best approximation of the solution is

$$p = (X^T X)^{-1} X^T y,$$

which takes a similar form to the previous equation, except that the inverse X^{-1} is replaced by the pseudoinverse $(X^T X)^{-1} X^T$.

Final Remarks

Finally, note that quantitative models are usually referred to as *regression* models when the goal is to predict some numeric value. This is contrasted with *classification* models, where the goal is to predict the category or *class* that best represents an input. So far, we have only encountered regression models, but we will learn about classification models soon.

Exercises

Use the pseudoinverse method to fit the given model to the given data set. Check your answer by sketching the resulting model on a graph containing the data points and verifying that it visually appears to capture the trend of the data. Remember that the model does not need to actually pass through all the data points (this is usually impossible).

1. Fit $y = mx + b$ to $[(1,0),(3,-1),(4,5)]$.
2. Fit $y = mx + b$ to $[(-2,3),(1,0),(3,-1),(4,5)]$.
3. Fit $y = ax^2 + bx + c$ to $[(-2,3),(1,0),(3,-1),(4,5)]$.
4. Fit $y = ax^2 + bx + c$ to $[(-3,-4),(-2,3),(1,0),(3,-1),(4,5)]$.
5. Fit $y = ax^3 + bx^2 + cx + d$ to $[(-3,-4),(-2,3),(3,-1),(1,0),(4,5)]$.

6. Fit $z = ax + by + c$ to $[(-2, 3, -3), (1, 0, -4), (3, -1, 2), (4, 5, 3)]$.

24. Regressing a Linear Combination of Nonlinear Functions via Pseudoinverse

Previously, we learned how to use the pseudoinverse to fit linear and polynomial models to data sets consisting of one or more input variables. However, the pseudoinverse is even more general than that.

Worked Example

For example, let's fit the model

$$y = a \sin x + b \cdot 2^x$$

to the following data set:

$$[(0,1),(2,5),(4,3)]$$

Although the model is more complicated, the procedure for fitting it is exactly the same. First, we set up our system:

$$\begin{aligned}(x,y) &\rightarrow y = a\cdot\sin x + b\cdot 2^x \\ \\ (0,1) &\rightarrow 1 = a\cdot\sin 0 + b\cdot 2^0 \\ (2,5) &\rightarrow 5 = a\cdot\sin 2 + b\cdot 2^2 \\ (4,3) &\rightarrow 3 = a\cdot\sin 4 + b\cdot 2^4\end{aligned}$$

Then we convert to a matrix equation and multiply both sides by the transpose of the coefficient matrix.

$$\begin{bmatrix}1\\5\\3\end{bmatrix} = \begin{bmatrix}\sin 0 & 2^0\\ \sin 2 & 2^2\\ \sin 4 & 2^4\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}$$

$$\begin{bmatrix}\sin 0 & \sin 2 & \sin 4\\ 2^0 & 2^2 & 2^4\end{bmatrix}\begin{bmatrix}1\\5\\3\end{bmatrix} = \begin{bmatrix}\sin 0 & \sin 2 & \sin 4\\ 2^0 & 2^2 & 2^4\end{bmatrix}\begin{bmatrix}\sin 0 & 2^0\\ \sin 2 & 2^2\\ \sin 4 & 2^4\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}$$

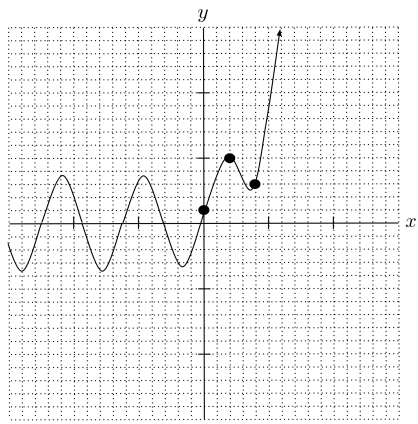
Finally, we evaluate the expression involving the inverse using a computer:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \left(\begin{bmatrix} \sin 0 & \sin 2 & \sin 4 \\ 2^0 & 2^2 & 2^4 \end{bmatrix} \begin{bmatrix} \sin 0 & 2^0 \\ \sin 2 & 2^2 \\ \sin 4 & 2^4 \end{bmatrix} \right)^{-1} \begin{bmatrix} \sin 0 & \sin 2 & \sin 4 \\ 2^0 & 2^2 & 2^4 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix}$$

$$\approx \begin{bmatrix} 3.89 \\ 0.37 \end{bmatrix}$$

We have the following model:

$$y \approx 3.89 \sin x + 0.37(2)^x$$



Pulling Functions Apart

We can apply the pseudoinverse method to fit linear combinations of general functions:

$$y = af(x) + bg(x) + ch(x) + \dots$$

Note that we can sometimes simplify models into the general form above even if they appear not to be of that form. For example, consider the following crazy-looking model:

$$y = \frac{\pm 2^{a+x} \pm \sqrt{bx}}{1+x}$$

By applying the rules of algebra, we can "pull apart" this model as follows:

$$\begin{aligned} y &= \frac{\pm 2^{a+x}}{1+x} + \frac{\pm\sqrt{bx}}{1+x} \\ &= \frac{\pm 2^a \cdot 2^x}{1+x} + \frac{\pm\sqrt{b}\cdot\sqrt{x}}{1+x} \\ &= \pm 2^a \cdot \frac{2^x}{1+x} \pm \sqrt{b} \cdot \frac{\sqrt{x}}{1+x} \end{aligned}$$

Note that 2^a and $\sqrt{b}$ are themselves constants, so this is in the desired general form. Let's fit this model to the data set that we've been working with:

$$[(0,1),(2,5),(4,3)]$$

First, we set up our system:

$$(x, y) \quad \rightarrow \quad y = \pm 2^a \cdot \frac{2^x}{1+x} \pm \sqrt{b} \cdot \frac{\sqrt{x}}{1+x}$$

$$\begin{aligned}
(0,1) \quad &\rightarrow \quad 1 = \pm 2^a \cdot \frac{2^0}{1+0} \pm \sqrt{b} \cdot \frac{\sqrt{0}}{1+0} \quad \rightarrow \quad 1 = \pm 2^a \cdot 1 \pm \sqrt{b} \cdot 0 \\
(2,5) \quad &\rightarrow \quad 5 = \pm 2^a \cdot \frac{2^2}{1+2} \pm \sqrt{b} \cdot \frac{\sqrt{2}}{1+2} \quad \rightarrow \quad 1 = \pm 2^a \cdot \frac{4}{3} \pm \sqrt{b} \cdot \frac{\sqrt{2}}{3} \\
(4,3) \quad &\rightarrow \quad 3 = \pm 2^a \cdot \frac{2^4}{1+4} \pm \sqrt{b} \cdot \frac{\sqrt{4}}{1+4} \quad \rightarrow \quad 1 = \pm 2^a \cdot \frac{16}{5} \pm \sqrt{b} \cdot \frac{2}{5}
\end{aligned}$$

Then we convert to a matrix equation and multiply both sides by the transpose of the coefficient matrix.

$$\begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 4/3 & \sqrt{2}/3 \\ 16/5 & 2/5 \end{bmatrix} \begin{bmatrix} \pm 2^a \\ \pm \sqrt{b} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4/3 & 16/5 \\ 0 & \sqrt{2}/3 & 2/5 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 & 4/3 & 16/5 \\ 0 & \sqrt{2}/3 & 2/5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 4/3 & \sqrt{2}/3 \\ 16/5 & 2/5 \end{bmatrix} \begin{bmatrix} \pm 2^a \\ \pm \sqrt{b} \end{bmatrix}$$

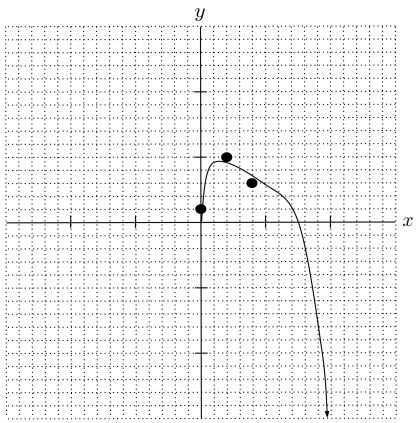
Finally, we evaluate the expression involving the inverse using a computer:

$$\begin{bmatrix} \pm 2^a \\ \pm\sqrt{b} \end{bmatrix} = \left(\begin{bmatrix} 1 & 4/3 & 16/5 \\ 0 & \sqrt{2}/3 & 2/5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 4/3 & \sqrt{2}/3 \\ 16/5 & 2/5 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 4/3 & 16/5 \\ 0 & \sqrt{2}/3 & 2/5 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix}$$

$$\approx \begin{bmatrix} -0.14 \\ 10.01 \end{bmatrix}$$

We have the following model:

$$y \approx -0.14 \cdot \frac{2^x}{1+x} + 10.01 \cdot \frac{\sqrt{x}}{1+x}$$



Functions that Cannot be Pulled Apart

Despite the example above, not every model can be fit using the pseudoinverse method. For example, the model below cannot be fit using the pseudoinverse because there are no rules of algebra that would allow us to pull it apart:

$$y = \sin(ax)\cos(bx)$$

Functions of Multiple Inputs

Finally, note that all the discussion above also generalizes to functions of multiple inputs. For example, we can apply the pseudoinverse method to fit any model of the following form:

$$z = af(x, y) + bg(x, y) + ch(x, y) + \dots$$

To demonstrate how this works, let's fit the model

$$z = ax \sin y + by \ln(1 + x)$$

to the following data set:

$$[(0,1,50),(2,5,30),(4,3,20),(5,1,10)]$$

First, we set up our system:

$$(x,y,z) \quad \to \quad z = a \cdot x \sin y + b \cdot y \ln(1+x)$$

$$\begin{aligned}
(0,1,50) &\quad \to \quad 50 = a \cdot 0 \sin 1 + b \cdot 1 \ln(1+0) \quad \to \quad 50 = a \cdot 0 + b \cdot 0 \\
(2,5,30) &\quad \to \quad 30 = a \cdot 2 \sin 5 + b \cdot 5 \ln(1+2) \quad \to \quad 30 = a \cdot 2 \sin 5 + b \cdot 5 \ln 3 \\
(4,3,20) &\quad \to \quad 20 = a \cdot 4 \sin 3 + b \cdot 3 \ln(1+4) \quad \to \quad 20 = a \cdot 4 \sin 3 + b \cdot 3 \ln 5 \\
(5,1,10) &\quad \to \quad 10 = a \cdot 5 \sin 1 + b \cdot 1 \ln(1+5) \quad \to \quad 10 = a \cdot 5 \sin 1 + b \cdot \ln 6
\end{aligned}$$

Then we convert to a matrix equation and multiply both sides by the transpose of the coefficient matrix.

$$\begin{bmatrix} 50 \\ 30 \\ 20 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 2\sin 5 & 5\ln 3 \\ 4\sin 3 & 3\ln 5 \\ 5\sin 1 & \ln 6 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2\sin 5 & 4\sin 3 & 5\sin 1 \\ 0 & 5\ln 3 & 3\ln 5 & \ln 6 \end{bmatrix} \begin{bmatrix} 50 \\ 30 \\ 20 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 & 2\sin 5 & 4\sin 3 & 5\sin 1 \\ 0 & 5\ln 3 & 3\ln 5 & \ln 6 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 2\sin 5 & 5\ln 3 \\ 4\sin 3 & 3\ln 5 \\ 5\sin 1 & \ln 6 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

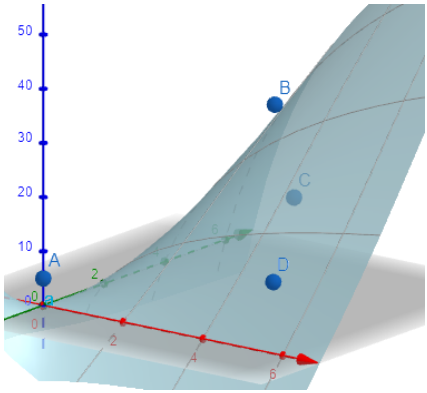
Finally, we evaluate the expression involving the inverse using a computer:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \left(\begin{bmatrix} 0 & 2\sin 5 & 4\sin 3 & 5\sin 1 \\ 0 & 5\ln 3 & 3\ln 5 & \ln 6 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 2\sin 5 & 5\ln 3 \\ 4\sin 3 & 3\ln 5 \\ 5\sin 1 & \ln 6 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 & 2\sin 5 & 4\sin 3 & 5\sin 1 \\ 0 & 5\ln 3 & 3\ln 5 & \ln 6 \end{bmatrix} \begin{bmatrix} 50 \\ 30 \\ 20 \\ 10 \end{bmatrix}$$

$$\approx \begin{bmatrix} -0.13 \\ 4.93 \end{bmatrix}$$

We have the following model:

$$z \approx -0.13\, x \sin y + 4.93\, y \ln(1 + x)$$



Exercises

Use the pseudoinverse method to fit each model to the given data set. Check your answer each time by sketching the resulting model on a graph containing the data points and verifying that it visually appears to capture the trend of the data.

1. Fit $y = a\ln(1+x) + \dfrac{b}{x}$ to $[(1,0),(3,-1),(4,5)]$.
2. Fit $y = \dfrac{ax+b}{2^x}$ to $[(1,0),(3,-1),(4,5)]$.
3. Fit $y = \pm 3^{a+x} \pm \sqrt[3]{bx}$ to $[(1,0),(3,-1),(4,5)]$.
4. Fit $z = axy^2 + b\cdot 2^{x+y}$ to $[(-2,3,-3),(1,0,-4),(3,-1,2),(4,5,3)]$.

25. Power, Exponential, and Logistic Regression via Pseudoinverse

Previously, we learned that we can use the pseudoinverse to fit any regression model that can be expressed as a linear combination of functions. Unfortunately, there are a handful of useful models that *cannot* be expressed as a linear combination of functions. Here, we will explore 3 of these models in particular.

$$\text{Power regression: } y = a \cdot x^b$$

$$\text{Exponential regression: } y = a \cdot b^x$$

$$\text{Logistic regression: } y = \frac{1}{1 + e^{-(ax+b)}}$$

The techniques that we will learn for fitting these models will apply more generally to any model that can be *transformed* into a linear combination of functions (where the parameters are the coefficients in the linear combination).

Power and Exponential Regressions

Power and exponential regressions are familiar algebraic functions, so we will address them first. To transform a power or exponential regression into a linear combination of functions, we can transform the equation by taking the natural logarithm of both sides and applying the laws of logarithms to pull apart the expression on the right-hand side:

Power regression

$$\ln y = \ln\left(a \cdot x^b\right) \quad \rightarrow \quad \ln y = \ln a + b \ln x$$

Exponential regression

$$\ln y = \ln\left(a \cdot b^x\right) \quad \rightarrow \quad \ln y = \ln a + (\ln b) \cdot x$$

Let's fit the power regression to the following data set.

$$[(1,1),(2,5),(4,3)]$$

As usual, we set up the system of equations, convert it into a matrix equation, and apply the pseudoinverse.

$$(x, y) \quad \rightarrow \quad \ln y = (\ln a) \cdot 1 + b \cdot \ln x$$

$$\begin{aligned}
(1, 1) \quad &\rightarrow \quad \ln 1 = (\ln a) \cdot 1 + b \cdot \ln 1 \quad \rightarrow \quad 0 = (\ln a) \cdot 1 + b \cdot 0 \\
(2, 5) \quad &\rightarrow \quad \ln 5 = (\ln a) \cdot 1 + b \cdot \ln 2 \\
(4, 3) \quad &\rightarrow \quad \ln 3 = (\ln a) \cdot 1 + b \cdot \ln 4
\end{aligned}$$

$$\begin{bmatrix} 0 \\ \ln 5 \\ \ln 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & \ln 2 \\ 1 & \ln 4 \end{bmatrix} \begin{bmatrix} \ln a \\ b \end{bmatrix}$$

$$\begin{aligned}
\begin{bmatrix} \ln a \\ b \end{bmatrix} &= \left(\begin{bmatrix} 1 & 1 & 1 \\ 0 & \ln 2 & \ln 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & \ln 2 \\ 1 & \ln 4 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 0 & \ln 2 & \ln 4 \end{bmatrix} \begin{bmatrix} 0 \\ \ln 5 \\ \ln 3 \end{bmatrix} \\
&\approx \begin{bmatrix} 0.35 \\ 0.79 \end{bmatrix}
\end{aligned}$$

So, we have the following model:

$$\ln y \approx 0.35 \cdot 1 + 0.79 \cdot \ln x$$

By combining the logarithms and reversing the transformation that we originally applied to the power regression $y = a \cdot x^b$, we can write the above model in power regression form:

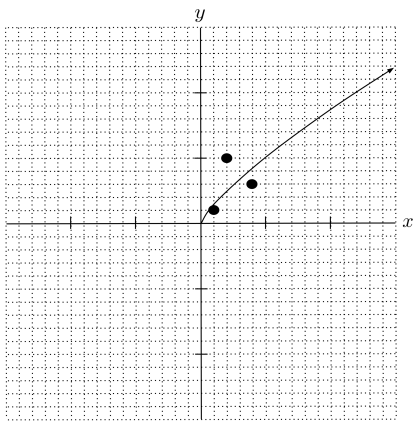
$$\ln y \approx 0.35 + 0.79 \cdot \ln x$$

$$\ln y \approx \ln(e^{0.35}) + \ln x^{0.79}$$

$$\ln y \approx \ln 1.42 + \ln x^{0.79}$$

$$\ln y \approx \ln 1.42\, x^{0.79}$$

$$y \approx 1.42\, x^{0.79}$$



Danger: Unintentional Domain Constraints

Notice that the data set we used above was slightly different from the data set that we've used earlier in this chapter. We changed the x-coordinate 0 to a 1 in the first data point.

Earlier: $[(\mathbf{0}, 1), (2, 5), (4, 3)]$

Now: $[(\mathbf{1}, 1), (2, 5), (4, 3)]$

The reason why we modified the data set is that the earlier data set exposes a limitation of the pseudoinverse method. We *can't* fit our power regression model to the earlier data set, because the x-coordinate of 0 causes a catastrophic issue in our transformed power regression equation.

To see what the issue is, let's substitute the point $(0, 1)$ into our transformed power regression equation and see what happens:

$$\begin{aligned} (x, y) \quad &\to \quad \ln y = (\ln a) \cdot 1 + b \cdot \ln x \\ (\mathbf{0}, 1) \quad &\to \quad \ln 1 = (\ln a) \cdot 1 + b \cdot \boxed{\ln \mathbf{0}} \end{aligned}$$

The quantity $\ln 0$ is not defined, so 0 is not a valid input for x. By transforming the equation, we unintentionally imposed a constraint on the valid inputs x.

Danger: Suboptimal Fit

Unfortunately, this is not the only bad news. Even with our new data set, which does not contain any points with an x-value of 0, the pseudoinverse did *not* give us the most accurate fit of the power regression $y = ax^b$. It gave us the most accurate fit of the transformed model $\ln y = (\ln a) \cdot 1 + b \cdot \ln x$, but this is *not* the most accurate fit of the original power regression even though the two equations are mathematically equivalent.

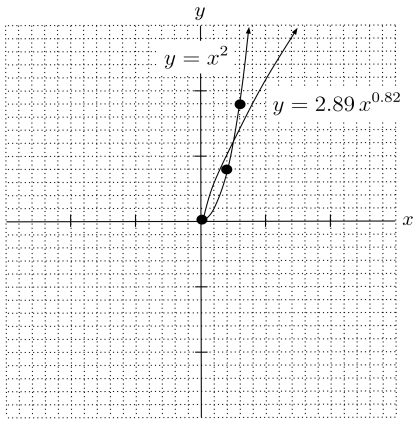
To understand this more clearly, let's consider an extreme example. If we were to fit a power regression to the data set

$$[(0.001, 0.01), (2, 4), (3, 9)]$$

using the same method that we demonstrated above, then we would get the following model:

$$y \approx 2.89\, x^{0.82}$$

However, if we plot this curve along with the data, then it's easy to see that this is not the most accurate model. It doesn't even curve the right way! A hand-picked model as simple as $y = x^2$ would be vastly more accurate.

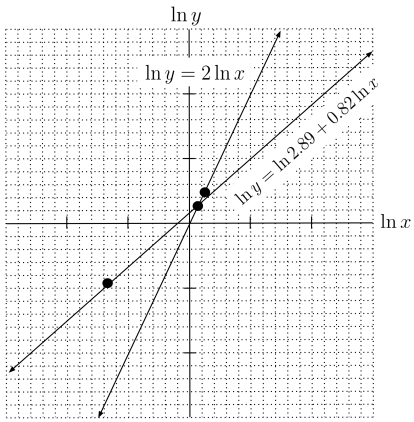


The reason the pseudoinverse method gave us an inaccurate model is that we transformed the model and data into a different space before we applied the pseudoinverse:

$$y = a \cdot x^b \quad \rightarrow \quad \ln y = (\ln a) \cdot 1 + b \cdot \ln x$$

In the space that the data was transformed to, it turns out that the model $y \approx 2.89\, x^{0.82}$ is more accurate than the model $y = x^2$. To visualize this, we can plot these two models along with the data in the space of points $(\ln x, \ln y)$.

$$\begin{aligned}
y = 2.89\, x^{0.82} \quad &\rightarrow \quad \ln y = \ln 2.89 + 0.82 \ln x \\
y = x^2 \quad &\rightarrow \quad \ln y = 2 \ln x \\
[(0.001, 0.01), (2, 4), (3, 9)] \quad &\rightarrow \quad [(\ln 0.001, \ln 0.01), (\ln 2, \ln 4), (\ln 3, \ln 9)] \\
&\qquad \approx [(-6.91, -4.61), (0.69, 1.39), (1.10, 2.20)]
\end{aligned}$$



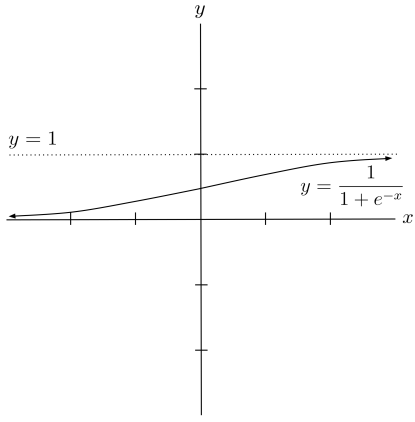
Indeed, $\ln y = \ln 2.89 + 0.82 \ln x$ is the most accurate model in the space of points $(\ln x, \ln y)$, but it is *not* the most accurate model in the space of points (x, y).

This is something to be cautious of whenever you transform a model into a space where the pseudoinverse can be applied – you should always check the function afterwards and verify that it is looks accurate "enough" for your purposes.

Logistic Regression

Soon, we will learn about other methods of fitting models that are robust to this sort of issue. But for now, let's end by discussing the logistic function:

$$y = \frac{1}{1 + e^{-(ax+b)}}$$



The logistic function has a range of $0 < y < 1$, and it is a common choice of model when the goal is to predict a bounded quantity. For example, probabilities are bounded between 0 and 1, and rating scales (such as movie ratings) are often bounded between $1 - 5$ and $1 - 10$.

Let's construct a real-life scenario in which to fit a logistic regression. Suppose that a food critic has rated sandwiches on a scale of $1 - 5$ as follows:

- A roast beef sandwich with 1 slice of roast beef was given a rating of 1 out of 5.
- A roast beef sandwich with 2 slices of roast beef was given a rating of 1 out of 5.
- A roast beef sandwich with 3 slices of roast beef was given a rating of 2 out of 5.

We will model the food critic's rating as a function of the number of slices of roast beef. Our data set will consist of points (x, y), where x represents the number of slices of roast beef and y represents the food critic's rating.

$$[(1, 1), (2, 1), (3, 2)]$$

To model the rating, we can fit a logistic function and then round the output to the nearest whole number. Since the rating scale is between $1 - 5$ and we are going to round the output of the logistic function, we

need to construct a logistic function with the range $0.5 < y < 5.5$. We can do this as follows:

$$\begin{aligned}
0 < \frac{1}{1+e^{-(ax+b)}} < 1 \\
0 < \frac{5}{1+e^{-(ax+b)}} < 5 \\
0.5 < \frac{5}{1+e^{-(ax+b)}} + 0.5 < 5.5
\end{aligned}$$

So, our goal is to fit the following model to the data set $[(1,1),(2,1),(3,2)]$.

$$y = \frac{5}{1+e^{-(ax+b)}} + 0.5$$

In order to fit the model using the pseudoinverse, we need to transform it into a linear combination of functions. We do so by manipulating the equation to isolate the $ax+b$:

$$
\begin{aligned}
y &= \frac{5}{1 + e^{-(ax+b)}} + 0.5 \\
y - 0.5 &= \frac{5}{1 + e^{-(ax+b)}} \\
1 + e^{-(ax+b)} &= \frac{5}{y - 0.5} \\
e^{-(ax+b)} &= \frac{5}{y - 0.5} - 1 \\
-(ax + b) &= \ln\left(\frac{5}{y - 0.5} - 1\right) \\
-\ln\left(\frac{5}{y - 0.5} - 1\right) &= ax + b
\end{aligned}
$$

Now, we can proceed with the usual process of fitting our model using the pseudoinverse.

$$(x, y) \quad \rightarrow \quad -\ln\left(\frac{5}{y - 0.5} - 1\right) = a \cdot x + b \cdot 1$$

$$
\begin{aligned}
&(1, 1) \quad \rightarrow \quad -\ln\left(\frac{5}{1 - 0.5} - 1\right) = a \cdot 1 + b \cdot 1 \quad \rightarrow \quad -\ln 9 = a \cdot 1 + b \cdot 1 \\
&(2, 1) \quad \rightarrow \quad -\ln\left(\frac{5}{1 - 0.5} - 1\right) = a \cdot 2 + b \cdot 1 \quad \rightarrow \quad -\ln 9 = a \cdot 2 + b \cdot 1 \\
&(3, 2) \quad \rightarrow \quad -\ln\left(\frac{5}{2 - 0.5} - 1\right) = a \cdot 3 + b \cdot 1 \quad \rightarrow \quad -\ln(7/3) = a \cdot 3 + b \cdot 1
\end{aligned}
$$

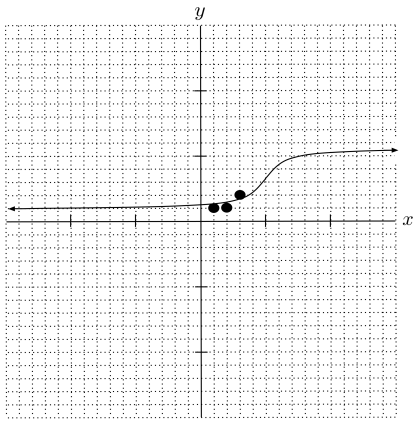
$$\begin{bmatrix} -\ln 9 \\ -\ln 9 \\ -\ln(7/3) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \left(\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -\ln 9 \\ -\ln 9 \\ -\ln(7/3) \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.67 \\ -3.10 \end{bmatrix}$$

So, we have the following model:

$$y \approx \frac{5}{1 + e^{-(0.67x - 3.10)}} + 0.5$$



Since we had to transform the model into a space where the pseudoinverse can be applied, we need to check the function and verify

that it looks accurate enough for our purposes. Here, our regression curve looks fairly accurate, so we will proceed to analyze it.

Now that we have a regression curve, we can use it to make predictions about the data. For example, what rating would we expect the critic to give a sandwich with 6 slices of roast beef? To answer this question, we just plug $x = 6$ into our model:

$$\begin{aligned} y(6) &\approx \frac{5}{1 + e^{-((0.67)(6) - 3.10)}} + 0.5 \\ &\approx 4.08 \end{aligned}$$

Our result of 4.08 rounds to 4. So, according to our model, we predict that the critic would give a rating of 4 to a sandwich that had 6 slices of roast beef.

Exercises

Use the pseudoinverse method to fit each model to the given data set. Check your answer each time by sketching the resulting model on a graph containing the data points and verifying that it visually appears to capture the trend of the data.

1. Fit a power regression $y = a \cdot x^b$ to $[(1, 0.2), (2, 0.3), (3, 0.5)]$.
2. Fit an exponential regression $y = a \cdot b^x$ to $[(1, 0.2), (2, 0.3), (3, 0.5)]$.

3. Fit a logistic regression $y = \dfrac{1}{1+e^{-(ax+b)}}$ to $[(1, 0.2), (2, 0.3), (3, 0.5)]$.

4. Construct a logistic regression whose range is $0.5 < y < 10.5$ and fit it to $[(1, 2), (2, 3), (3, 5)]$.

26. Overfitting, Underfitting, Cross-Validation, and the Bias-Variance Tradeoff

We have previously described a model as "accurate" when it appears to match closely with points in the data set. However, there are issues with this definition that we will need to remedy. In this section, we will expose these issues and develop a more nuanced understanding of model accuracy by way of a concrete example.

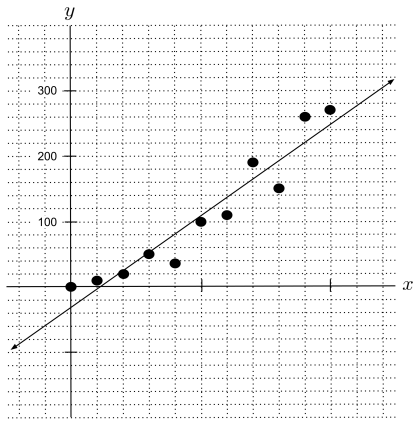
Overfitting and Underfitting

Consider the following data set of points (x, y), where x is the number of seconds that have elapsed since a rocket has launched and y is the height (in meters) of the rocket above the ground. The data is *noisy*, meaning that there is some random error in the measurements.

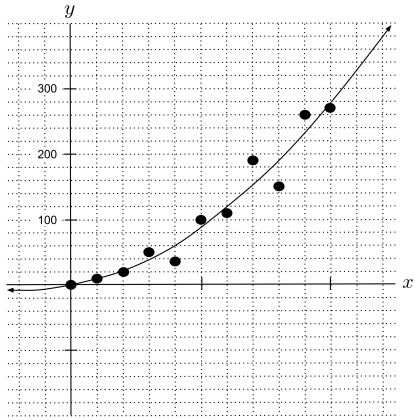
$$[(0,0),(1,10),(2,20),(3,50),(4,35),(5,100),$$
$$(6,110),(7,190),(8,150),(9,260),(10,270)]$$

If we use the pseudoinverse to fit linear, quadratic, and 8th degree polynomial regressions, we get the following results. (You should do this yourself and verify that you get the same results.)

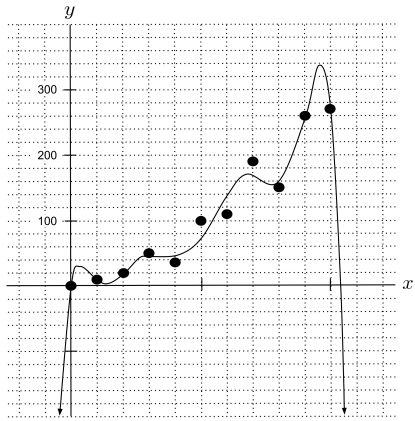
Linear: $y = 28.14x - 32.05$



Quadratic: $y = 2.05x^2 + 7.68x - 1.36$



8th degree: $y = -0.00778x^8 + 0.296666x^7 - 4.575859x^6$
$+ 36.618048x^5 - 162.04318x^4 + 390.1833x^3$
$- 462.14466x^2 + 212.239681x - 0.087308$



Note that we used more decimal places in the 8th degree polynomial regressor because it is very sensitive to rounding. (If you round the coefficients to 2 decimal places and plot the result, it will look wildly different.)

As we can see from the graph, the 8th degree polynomial regressor matches closest with the points in the data set. So, if we define "accuracy" as simply matching with points in the data set, then we'd say the 8th degree polynomial is the most accurate.

However, this doesn't pass the sniff test. In order to match up with the points in the data set so well, the 8th degree polynomial has to curve and contort itself in unrealistic ways between data points.

- For example, between the last two points in the data set, the 8th degree polynomial rises up and falls down sharply. Did the rocket really rise up 100 meters and fall down 100 meters between the 9th and 10th seconds? Probably not.
- Likewise, the 8th degree polynomial continues dropping sharply after the 10th second and hits the x-axis shortly after. Is the rocket really plummeting to the ground? Based on the actual data points, it seems unlikely.

So, we can't place much trust in the 8th degree polynomial's predictions. The 8th degree polynomial thinks the relationship between x and y is more complicated than it actually is. When this happens, we say that the model **overfits** the data.

On the other hand, the linear regression does not capture the fact that the data is curving upwards. It thinks the relationship between x and

y is *less* complicated than it actually is. When this happens, we say that the model **underfits** the data.

Based on the graph, the quadratic regressor looks the most accurate. It captures the fact that the data is curving upwards, but it does not contort itself in unrealistic ways between data points, and it does not predict that the rocket is going to fall straight down to the ground immediately after the 10th second. The quadratic regressor neither overfits nor underfits. So, we can place more trust in its predictions.

The Need for Cross Validation

The discussion above suggests that our definition of accuracy should be based on how well a model predicts things, not how well it matches up with the data set. So, we have a new definition of what it means for a model to be accurate:

- *Old (bad) definition:* The closer a model matches up with points in the data set, the more accurate it is.
- *New (good) definition:* The better a model predicts data points that it hasn't seen (i.e. were not used during the model fitting procedure), the more accurate it is.

But how can we measure how well a model predicts data points that it hasn't seen, if we are fitting it on the entire data set? The answer is surprisingly simple: when measuring the accuracy of a model, don't

fit the model on the entire data set. Instead, carry out the following procedure, known as **leave-one-out cross validation**:

1. Remove a point from the data set.
2. Fit the model to all points in the data set *except* the point that we removed.
3. Check how accurately the model would have predicted the point that we left out.
4. Place the point back in the data set.
5. Repeat steps $1 - 4$ for every point in the data set and add up how much each prediction was "off" by. This is called the *cross-validation error*.
6. The model with the lowest cross-validation error is the most accurate, i.e. it does the best job of predicting real data points that it has not seen before.

The phrase *leave-one-out* refers to when we remove a point from the data set. The phrase *cross validation* refers to when we have the model predict the point we removed: we're *validating* that the model still does a good job of predicting points that it hasn't already been fitted on. The word *cross* indicates that during our validation, we're asking the model to "cross over" from points it has seen, to points it has not seen.

Example: Cross Validation

Let's demonstrate the leave-one-out cross validation procedure by computing the cross-validation error for linear, quadratic, and 8th degree polynomial regression models on our rocket launch data set. We should find that the quadratic regression has the lowest error, the linear regression has slightly higher error, and the 8th degree polynomial has significantly higher error.

The table below shows some of the intermediate steps for computing the cross-validation error for linear regression. In each row of the table, we remove a point from the data set, fit a linear regression model to the remaining data, and then get the predicted y-value by plugging the x-coordinate of the removed point into the model.

Removed Point	Remaining Data	Model	Predicted Y-Value
$(0,0)$	$\begin{bmatrix} & (1,10), & (2,20), \\ (3,50), & (4,35), & (5,100), \\ (6,110), & (7,190), & (8,150), \\ (9,260), & (10,270) & \end{bmatrix}$	$y = 30.27x - 47.0$	-47.0
$(1,10)$	$\begin{bmatrix} (0,0) & & (2,20), \\ (3,50), & (4,35), & (5,100), \\ (6,110), & (7,190), & (8,150), \\ (9,260), & (10,270) & \end{bmatrix}$	$y = 28.8x - 37.01$	-8.21
$(2,20)$	$\begin{bmatrix} (0,0) & (1,10), & \\ (3,50), & (4,35), & (5,100), \\ (6,110), & (7,190), & (8,150), \\ (9,260), & (10,270) & \end{bmatrix}$	$y = 28.0x - 30.88$	25.12
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(10,270)$	$\begin{bmatrix} (0,0) & (1,10), & (2,20), \\ (3,50), & (4,35), & (5,100), \\ (6,110), & (7,190), & (8,150), \\ (9,260), & & \end{bmatrix}$	$y = 26.76x - 27.91$	239.69

Once we have all the predicted points, we can compare them to the removed points and compute the cross-validation error. For each row in the table, we compute the *square* of the difference between the predicted y-value and the actual y-value in the removed point.

$(x_{\text{actual}}, y_{\text{actual}})$	$y_{\text{predicted}}$	$(y_{\text{predicted}} - y_{\text{actual}})^2$
$(0, 0)$	-47.0	$(-47.0 - 0)^2$
$(1, 10)$	-8.21	$(-8.21 - 10)^2$
$(2, 20)$	25.12	$(25.12 - 20)^2$
$\vdots$	$\vdots$	$\vdots$
$(10, 270)$	239.69	$(239.69 - 270)^2$

Note that although it might feel more natural to take the absolute value instead of the square when computing the total error, it's conventional to work with squared distances in machine learning and statistics because squaring has more desirable mathematical properties. For example, squaring is differentiable everywhere, whereas absolute value is not (the graph of the absolute value function is not differentiable at 0).

Finally, we sum up all the individual squared errors to get the total cross-validation error. This sum of squared errors is also known as the *cross-validated RSS (residual sum of squares)*.

$$(-47.0 - 0)^2 + (-8.21 - 10)^2 + (25.12 - 20)^2 + \ldots + (239.69 - 270)^2$$

If we compute the cross-validation error for the linear, quadratic, and 8th degree polynomial regressors separately, then we get the following results (rounded to the nearest integer). Note that these results were generated via code, and there was no intermediate rounding like was done while working out the example above. You should try to calculate

these cross-validation errors on your own and verify that your numbers match up.

Model	Cross-Validation Error
Linear	13 143
Quadratic	8 033
8th Degree	7 615 290

If you need to debug anything by hand and round intermediate results, remember to keep many decimal places in the coefficients of the 8th degree polynomial regressor. (If you use too few decimal places, then the regressor will give wildly different results.)

The results are just as we expected:

- The quadratic regressor has the lowest cross-validation error, which means it is the most accurate model to use for this data set.
- The linear regressor has sightly higher cross-validation error because it slightly underfits the data (it doesn't capture the fact that the data is curving upwards).
- The 8th degree polynomial has massively higher cross-validation error because it massively overfits the data (it contorts itself and overcomplicates things so much that it thinks the rocket is plummeting to the ground).

Bias-Variance Tradeoff

Finally, let's build more intuition about why these results came out as they did. It turns out that the total cross-validation error in our model is the sum of two different types of error:

$$\begin{pmatrix}\text{total cross-}\\\text{validation error}\end{pmatrix} = \begin{pmatrix}\text{error due}\\\text{to bias}\end{pmatrix} + \begin{pmatrix}\text{error due}\\\text{to variance}\end{pmatrix}$$

Loosely speaking, error due to **bias** occurs if a model assumes too much about the relationship being modeled. In our example, the linear regressor had high error due to bias because it assumed that the relationship was a straight line (whereas the data was actually curving upwards a bit). The error due to bias comes from a model's inability to pass through all the points that are being used to fit it. A model with high bias is too *rigid* to capture some trends in the data, and therefore *underfits* the data.

On the other hand, error due to **variance** occurs if a model changes drastically when fit to a different sample of data points from the same data set. In our example, the 8th degree polynomial regressor had high error due to variance. To see this, you should plot all the different 8th degree regressors that you came up with during your leave-one-out cross validation and observe that the graphs look quite different from one another. This is bad because a model is supposed to pick up on the true relationship underlying some data, and if the model changes its mind significantly when it's shown different samples of data from the same data set, then we can't trust it! A model with high variance is so *flexible*

that it reads too much into the relationship between points in the data set and contorts itself in weird ways.

To build further intuition for bias and variance, it can help to anthropomorphize a bit:

- A high-bias model is dumb: it can't comprehend the complexity of the relationship in the data.
- A high-variance model is paranoid: it thinks it's seeing all sorts of complicated relationships that just aren't there.

Ideally, we would like to minimize error due to bias and error due to variance. However, bias and variance are two sides of the same coin: when one decreases, the other increases. By minimizing one, we maximize the other. In particular:

- By making the model less rigid (i.e. decreasing bias), we make the model more flexible (i.e. increase variance).
- By making the model less flexible (i.e. decreasing variance), we make the model more rigid (i.e. increase bias).

This is known as the the **bias-variance tradeoff**, and it means that we cannot simply minimize bias and variance independently. This is why cross-validation is so useful: it allows us to compute and thereby minimize the sum of error due to bias and error due to variance, so that we may find the ideal tradeoff between bias and variance.

K-Fold Cross Validation

In closing, note that although leave-one-out cross validation can take a long time to run on large data sets, a similar procedure called k-**fold cross validation** can be used instead. Instead of removing one point at a time, we shuffle the data set, break it up into k parts (with each part containing roughly the same number of points), and then remove one of the parts each time. Each time, we predict the points in the part that we left out, and then we add up all the squared errors. We usually choose k to be a small number (such as $k = 5$) so that the cross validation procedure runs fairly quickly.

Exercise

Work out the computations that were outlined in the examples above. Fit and compute leave-one-out cross validation error for linear, quadratic, and 8th degree polynomials and verify that your results match up with the examples.

27. Regression via Gradient Descent

Gradient descent can be applied to fit regression models. In particular, it can help us avoid the pitfalls that we've experienced when we attempt to fit nonlinear models using the pseudoinverse.

Previously, we fit a power regression $y = ax^b$ to the following data set and got a result that was quite obviously not the most accurate fit.

$$[(0.001, 0.01), (2, 4), (3, 9)]$$

This time, we will use gradient descent to fit the power regression and observe that our resulting model fits the data much better.

Model-Fitting as a Minimization Problem

To fit a model using gradient descent, we just have to construct an expression that represents the error between the model and the data

that it's supposed to fit. Then, we can use gradient descent to minimize that expression.

To represent the error between the model and the data that it's supposed to fit, we can use the *residual sum of squares (RSS)*. To compute the RSS, we just add up the squares of the differences between the model's predictions and the actual data.

data point	$\rightarrow$	predicted y-value	vs	data y-value
(x,y)	$\rightarrow$	ax^b	vs	y
$(0.001, 0.01)$	$\rightarrow$	$a \cdot (0.001)^b$	vs	0.01
$(2,4)$	$\rightarrow$	$a \cdot 2^b$	vs	4
$(3,9)$	$\rightarrow$	$a \cdot 3^b$	vs	9

Summing up the squared differences between the predicted y-values and the y-values in the data, we get the following expression for the RSS:

$$\text{RSS} = \left(a \cdot (0.001)^b - 0.01\right)^2 + \left(a \cdot 2^b - 4\right)^2 + \left(a \cdot 3^b - 9\right)^2$$

Now, this is a normal gradient descent problem. We choose an initial guess for a and b and then use the partial derivatives $\frac{\partial \text{RSS}}{\partial a}$ and $\frac{\partial \text{RSS}}{\partial b}$ to repeatedly update our guess so that it results in a lower RSS.

Computing partial derivatives, we have the following:

$$\begin{aligned}\frac{\partial \text{RSS}}{\partial a} &= 2\left(a \cdot (0.001)^b - 0.01\right)(0.001)^b \\ &\quad + 2\left(a \cdot 2^b - 4\right) \cdot 2^b \\ &\quad + 2\left(a \cdot 3^b - 9\right) \cdot 3^b \\ \frac{\partial \text{RSS}}{\partial b} &= 2\left(a \cdot (0.001)^b - 0.01\right) \cdot a \cdot (0.001)^b \ln(0.001) \\ &\quad + 2\left(a \cdot 2^b - 4\right) \cdot a \cdot 2^b \ln 2 \\ &\quad + 2\left(a \cdot 3^b - 9\right) \cdot a \cdot 3^b \ln 3\end{aligned}$$

Worked Example

Let's start with the initial guess $\langle a_0, b_0 \rangle = \langle 1, 1 \rangle$, which corresponds to the straight line $y = 1x^1$. Our gradient is

$$\begin{aligned}\nabla \text{RSS}(a_0, b_0) &= \left\langle \frac{\partial \text{RSS}}{\partial a}, \frac{\partial \text{RSS}}{\partial b} \right\rangle \Bigg|_{(a_0, b_0)} \\ &= \langle -44.000018, -45.095095 \rangle ,\end{aligned}$$

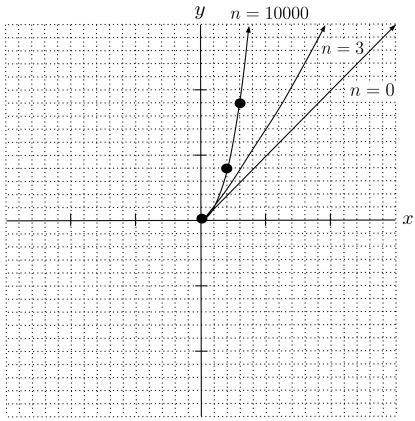
and using learning rate $\alpha = 0.001$ our updated guess is

$$\begin{aligned}\langle a_1, b_1 \rangle &= \langle a_0, b_0 \rangle - \alpha \nabla \text{RSS}(a_0, b_0) \\ &= \langle 1, 1 \rangle - (0.001) \langle -44.000018, -45.095095 \rangle \\ &= \langle 1.044000, 1.045095 \rangle .\end{aligned}$$

If we continue the process, we get the results shown in the table below.

n	$\langle a_n, b_n \rangle$	$\nabla \text{RSS}(a_n, b_n)$	$\text{RSS}(a_n, b_n)$
0	$\langle 1, 1 \rangle$	$\langle -44.000018, -45.095095 \rangle$	40.000081
1	$\langle 1.044000, 1.045095 \rangle$	$\langle -43.610529, -46.794623 \rangle$	35.998548
2	$\langle 1.087611, 1.091890 \rangle$	$\langle -42.948407, -48.155772 \rangle$	31.88666
3	$\langle 1.130559, 1.140045 \rangle$	$\langle -41.947662, -49.053128 \rangle$	27.719376
50	$\langle 1.450958, 1.640770 \rangle$	$\langle 0.849948, -0.569792 \rangle$	0.315108
100	$\langle 1.410093, 1.668529 \rangle$	$\langle 0.783786, -0.539881 \rangle$	0.266312
500	$\langle 1.185035, 1.836774 \rangle$	$\langle 0.378140, -0.307757 \rangle$	0.061737
1000	$\langle 1.065472, 1.939139 \rangle$	$\langle 0.137242, -0.123755 \rangle$	0.008426
5000	$\langle 1.000014, 1.999987 \rangle$	$\langle -0.000029, -0.000028 \rangle$	0.000100
10000	$\langle 1.000000, 2.000000 \rangle$	$\langle 0.000000, 0.000000 \rangle$	0.000100

Our gradient descent converged to $a = 1$ and $b = 2$, which corresponds to the function $y = 1x^2$. As we can see from the graph below, this is a very good fit.



Sigma Notation and Implementation

Note that when implementing gradient descent on a data set consisting of more than a few points, it becomes infeasible to hard-code the entire expression for the RSS gradient. Instead, it becomes necessary to write a function that loops through the points in the data set and incrementally adds up each point's individual contribution to the total RSS gradient. It also becomes convenient to re-use intermediate values when possible.

To think through this, it's helpful to express the RSS and its gradient using sigma notation. In the example above, the RSS is given by

$$\text{RSS} = \sum_{(x,y) \in \text{data}} \left(ax^b - y\right)^2,$$

and its gradient is computed as

$$\begin{aligned}
\nabla \text{RSS} &= \sum_{(x,y) \in \text{data}} \nabla \left(ax^b - y\right)^2 \\
&= \sum_{(x,y) \in \text{data}} 2\left(ax^b - y\right) \nabla \left(ax^b - y\right) \\
&= \sum_{(x,y) \in \text{data}} 2\left(ax^b - y\right) \left\langle \frac{\partial}{\partial a}\left(ax^b - y\right), \frac{\partial}{\partial b}\left(ax^b - y\right) \right\rangle \\
&= \sum_{(x,y) \in \text{data}} 2\left(ax^b - y\right) \left\langle x^b, abx^{b-1} \right\rangle \\
&= \sum_{(x,y) \in \text{data}} 2\left(ax^b - y\right) x^b \left\langle 1, abx^{-1} \right\rangle .
\end{aligned}$$

Now that we've worked out the sigma notation, we can write a function that mirrors it:

```
gradRSS(a, b, data):
    da = 0
    db = 0
    for (x,y) in data:
        common = 2 * (ax^b - y) * x^b
        da += common
        db += common * a * b * x^-1
    return da, db
```

Debugging with Central Difference Quotients

Lastly, note that when debugging broken gradient descent code, it can be helpful to check your partial derivatives against difference quotient approximations to ensure that you're computing the partial derivatives correctly:

$$\frac{\partial \text{RSS}}{\partial a} \approx \frac{\text{RSS}(a+h, b) - \text{RSS}(a-h, b)}{2h}$$
$$\frac{\partial \text{RSS}}{\partial b} \approx \frac{\text{RSS}(a, b+h) - \text{RSS}(a, b-h)}{2h}$$

where $0 < h \ll 1$ is a small positive number.

Do not abuse difference quotients and attempt to use them to fully bypass gradient computations. Use difference quotients *only* for debugging. Difference quotients will be too slow to effectively train more advanced models (such as neural networks), and it's useful to practice gradient computations on simpler models before moving on to more advanced models.

Exercises

Use gradient descent to fit the following models. Be sure to plot your model on the same graph as the data to ensure that the fit is looks reasonable.

1. Implement the example that was worked out above.
2. Fit $y = ax^2 + bx + c$ to $[(0.001, 0.01), (2, 4), (3, 9)]$. Verify that gradient descent gives the *same* fit as compared to using the pseudoinverse.
3. Fit $y = \dfrac{5}{1 + e^{-(ax+b)}} + 0.5$ to $[(1, 1), (2, 1), (3, 2)]$. Verify that gradient descent gives a *better* fit as compared to using the pseudoinverse.
4. Fit $y = a \sin bx + c \sin dx$ to

$$\begin{bmatrix} (0,0)\,, & (1,-1)\,, & (2,2)\,, & (3,0)\,, & (4,0) \\ (5,2)\,, & (6,-4)\,, & (7,4)\,, & (8,1)\,, & (9,-3) \end{bmatrix}.$$

28. Multiple Regression and Interaction Terms

In many real-life situations, there is more than one factor that controls the quantity we're trying to predict. That is to say, there is more than one input variable that controls the output variable.

Example: Multiple Input Variables

For example, suppose that a food manufacturing company is testing out different ingredients on sandwiches, including peanut butter and roast beef. They fed sandwiches to subjects and counted the proportion of subjects who liked each sandwich.

scoops peanut butter	scoops jelly	slices beef	proportion subjects liked
0	0	0	0.0
1	0	0	0.2
2	0	0	0.5
0	1	0	0.4
0	2	0	0.6
0	0	1	0.5
0	0	2	0.8
1	1	0	1.0
1	0	1	0.0
0	1	1	0.1

We want to build a model that has 3 input variables:

$$\begin{aligned} x_1 &= \text{scoops peanut butter} \\ x_2 &= \text{scoops jelly} \\ x_3 &= \text{slices beef} \end{aligned}$$

The model will predict 1 output variable:

$$y = \text{proportion subjects liked}$$

Since this output variable must be between 0 and 1, we will use logistic regression.

$$y = \frac{1}{1 + e^{-(ax+b)}}$$

The logistic model above is written with only a single input variable. Here, we have 3 different input variables, so we will introduce a new term for each input variable:

$$y = \frac{1}{1 + e^{-(a_1x_1+a_2x_2+a_3x_3+b)}}$$

We should also introduce terms that represent interactions between the variables, but to keep things simple and illustrate why such terms are needed, let's continue without them.

If we fit the above model to our data set by running gradient descent a handful of times with different initial guesses and choosing the best result, we get the following fitted model:

$$y = \frac{1}{1 + e^{-(0.79x_1+1.13x_2+0.75x_3-1.72)}}$$

The Need for Interaction Terms

This model makes the following predictions. Some of them seem accurate, but others do not.

scoops peanut butter	scoops jelly	slices beef	proportion subjects liked	prediction	
0	0	0	0.0	0.15	✓
1	0	0	0.2	0.28	✓
2	0	0	0.5	0.47	✓
0	1	0	0.4	0.36	✓
0	2	0	0.6	0.63	✓
0	0	1	0.5	0.27	×
0	0	2	0.8	0.44	×
1	1	0	1.0	0.55	×
1	**0**	**1**	**0.0**	**0.46**	×
0	1	1	0.1	0.54	×

The weirdest inaccurate prediction (bolded above) is that the model overrates peanut butter & roast beef sandwiches. It thinks that half of the subjects will like them, when in reality, none of the subjects did. And if you try to imagine that combination of ingredients, it probably doesn't seem appetizing.

The problem is that our model is not sophisticated enough to capture the idea that two ingredients can taste good alone but bad together (or vice versa). It's easy to see why this is:

- The logistic function $\frac{1}{1+e^{-(ax+b)}}$ is increasing if $a > 0$ and decreasing if $a < 0$.
- The coefficient on x_1 (peanut butter) is $a_1 = 1.02$ and the coefficient on x_3 (roast beef) is $a_3 = 1.91$.

- Both of these coefficients are positive. Consequently, the higher x_1 (the more scoops of peanut butter), the higher the prediction will be. Likewise, the higher x_3 (the more slices of roast beef), the higher the prediction will be.

Interaction Terms

To fix this, we can add **interaction terms** that multiply two variables together. These terms will vanish unless both variables are nonzero.

$$y = \frac{1}{1 + e^{-(a_1x_1 + a_2x_2 + a_3x_3 + a_{12}x_1x_2 + a_{13}x_1x_3 + a_{23}x_2x_3 + b)}}$$

The interaction terms above are $a_{12}x_1x_2$, $a_{13}x_1x_3$, and $a_{23}x_2x_3$. The subscripts indicate which variables are being multiplied together.

Notice that, for example, the interaction term $a_{13}x_1x_3$ will *not* have an effect on the predictions for x_1 (peanut butter) or x_3 (roast beef) in isolation, but it *will* have an effect when these ingredients are combined.

If we fit this model again using gradient descent, we get the following result:

$$y = \frac{1}{1 + e^{-(1.02x_1 + 1.34x_2 + 1.91x_3 + 3.82x_1x_2 - 4.82x_1x_3 - 3.34x_2x_3 - 2.11)}}$$

Now, the model makes much more accurate predictions.

scoops peanut butter	scoops jelly	slices beef	proportion subjects liked	prediction	
0	0	0	0.0	0.11	✓
1	0	0	0.2	0.25	✓
2	0	0	0.5	0.48	✓
0	1	0	0.4	0.32	✓
0	2	0	0.6	0.64	✓
0	0	1	0.5	0.45	✓
0	0	2	0.8	0.85	✓
1	1	0	1.0	0.98	✓
1	0	1	0.0	0.02	✓
0	1	1	0.1	0.10	✓

As a sanity check, we can also interpret the coefficients of the interaction terms:

- The interaction term between x_1 (peanut butter) and x_2 (jelly) is $3.82\,x_1x_2$. The *positive* coefficient indicates that combining peanut butter and jelly should *increase* the prediction.
- The interaction term between x_1 (peanut butter) and x_3 (roast beef) is $-4.82\,x_1x_3$. The *negative* coefficient indicates that combining peanut butter and roast beef should *decrease* the prediction.
- The interaction term between x_2 (jelly) and x_3 (roast beef) is $-3.34\,x_2x_3$. The *negative* coefficient indicates that combining jelly and roast beef should *decrease* the prediction.

Intuitively, this all makes sense. Peanut butter & jelly go together, but peanut butter & roast beef do not go together, and nor do jelly & roast beef.

Exercise

Implement the example that was worked out above.

29. K-Nearest Neighbors

Until now, we have been focused on **regression** problems, in which we predict an output quantity (that is often continuous). However, in the real world, it's even more common to encounter **classification** problems, in which we predict an output class (i.e. category). For example, predicting how much money a person will spend at a store is a regression problem, whereas predicting which items the person will buy is a classification problem.

K-Nearest Neighbors

One of the simplest classification algorithms is called **k-nearest neighbors**. Given a data set of points labeled with classes, the k-nearest neighbors algorithm predicts the class of a new data point by

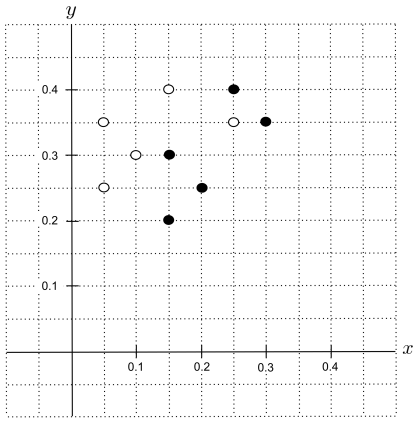
1. finding the k points in the data set that are nearest to the new point (i.e. its k nearest neighbors),
2. finding the class that occurs most often in those k points (also known as the *majority class*), and

3. and predicting that the new data point belongs to the majority class of its k nearest neighbors.

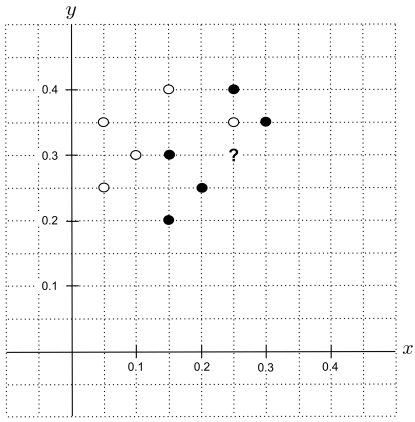
As a concrete example, consider the following data set. Each row represents a cookie with some ratio of ingredients. If we know the portion of ingredients in a new cookie, we can use the k-nearest neighbors algorithm to predict which type of cookie it is.

Cookie Type	Portion Butter	Portion Sugar
Shortbread	0.15	0.2
Shortbread	0.15	0.3
Shortbread	0.2	0.25
Shortbread	0.25	0.4
Shortbread	0.3	0.35
Sugar	0.05	0.25
Sugar	0.05	0.35
Sugar	0.1	0.3
Sugar	0.15	0.4
Sugar	0.25	0.35

Let's start by plotting the data. We'll represent shortbread cookies using filled circles and sugar cookies using open circles. The x-axis will measure the portion butter, while the y-axis will measure the portion sugar.

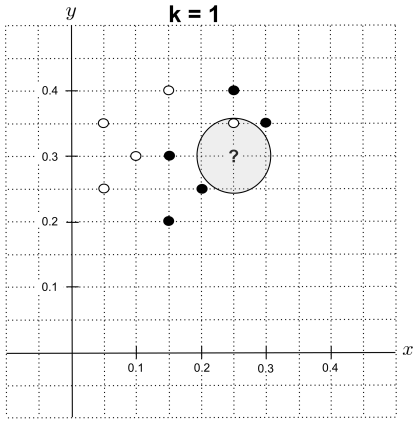


Suppose we have a cookie recipe that consists of 0.25 portion butter and 0.3 portion sugar, and we want to predict whether this is a shortbread cookie or a sugar cookie. First, we'll identify the corresponding point $(0.25, 0.3)$ on our graph and label it with a question mark.



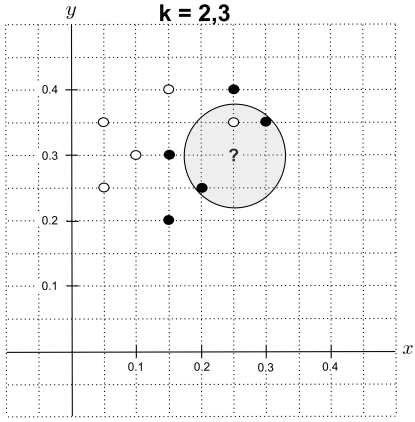
To identify the k nearest neighbors of the unknown point, we can draw the smallest circle around the unknown point such that the circle contains k other points from our data set.

The circle for $k = 1$ is shown below. Since this nearest neighbor is a sugar cookie, we predict that the unknown cookie is also a sugar cookie.

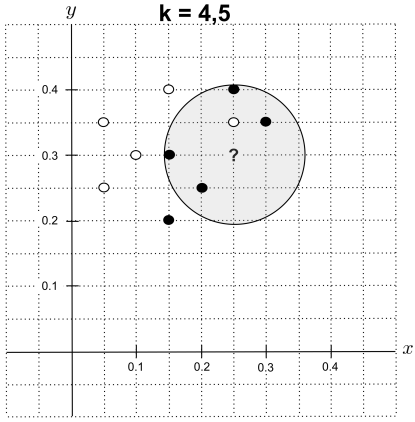


Using $k = 2$ instead, we get the circle shown below. Notice that after the first nearest neighbor, the next two nearest neighbors are the same distance away from our unknown point, so we have to include both of them in our circle. Consequently, using $k = 3$ gives us the exact same circle.

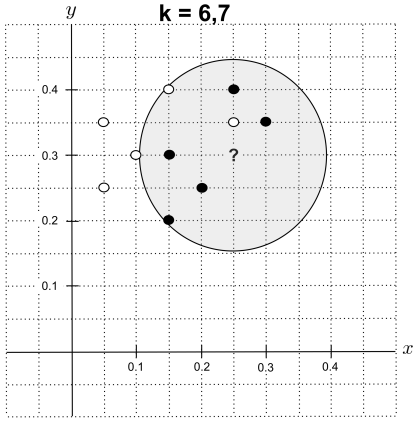
Now we have 3 nearest neighbors: 2 shortbread cookies and 1 sugar cookie. As a result, the majority class is shorbtread, and we predict that the unknown cookie is a shortbread cookie.



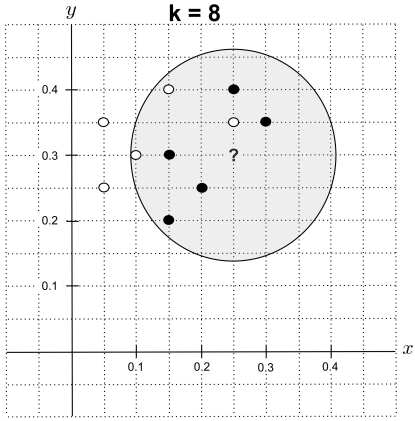
Using $k = 4$ or $k = 5$, we get the circle below. The nearest neighbors are 4 shortbread cookies and 1 sugar cookie, so we predict that the unknown cookie is a shortbread cookie.



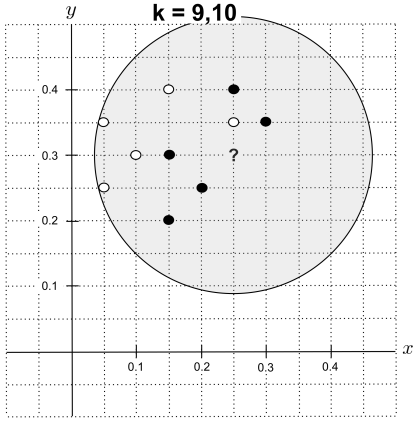
Using $k = 6$ or $k = 7$, the nearest neighbors are 5 shortbread cookies and 2 sugar cookies, so we predict that the unknown cookie is a shortbread cookie.



Using $k = 8$, the nearest neighbors are 5 shortbread cookies and 3 sugar cookies, so we predict that the unknown cookie is a shortbread cookie.



Using $k = 9$ or $k = 10$, the nearest neighbors are 5 shortbread cookies and 5 sugar cookies. There is a tie for the majority class, so we will break the tie by choosing the class of nearest neighbors that has the lowest total distance from the unknown point.



We compute the distances of the nearest neighbors as follows:

Cookie Type	Point	Distance From $(0.25, 0.3)$	
Shortbread	$(0.15, 0.2)$	$\sqrt{(0.15-0.25)^2+(0.2-0.3)^2}$	≈ 0.141
Shortbread	$(0.15, 0.3)$	$\sqrt{(0.15-0.25)^2+(0.3-0.3)^2}$	$= 0.1$
Shortbread	$(0.2, 0.25)$	$\sqrt{(0.2-0.25)^2+(0.25-0.3)^2}$	≈ 0.071
Shortbread	$(0.25, 0.4)$	$\sqrt{(0.25-0.25)^2+(0.4-0.3)^2}$	$= 0.1$
Shortbread	$(0.3, 0.35)$	$\sqrt{(0.3-0.25)^2+(0.35-0.3)^2}$	≈ 0.071
Sugar	$(0.05, 0.25)$	$\sqrt{(0.05-0.25)^2+(0.25-0.3)^2}$	≈ 0.206
Sugar	$(0.05, 0.35)$	$\sqrt{(0.05-0.25)^2+(0.35-0.3)^2}$	≈ 0.206
Sugar	$(0.1, 0.3)$	$\sqrt{(0.1-0.25)^2+(0.3-0.3)^2}$	$= 0.15$
Sugar	$(0.15, 0.4)$	$\sqrt{(0.15-0.25)^2+(0.4-0.3)^2}$	≈ 0.141
Sugar	$(0.25, 0.35)$	$\sqrt{(0.25-0.25)^2+(0.35-0.3)^2}$	$= 0.05$

Then, we compute the total distance for each class of nearest neighbors:

$$\begin{aligned} \text{Shortbread:}\quad & 0.141 + 0.1 \quad + 0.071 + 0.1 \quad + 0.071 = 0.483 \\ \text{Sugar:}\quad & 0.206 + 0.206 + 0.15 \quad + 0.141 + 0.05 \quad = 0.753 \end{aligned}$$

Since shortbread neighbors have a lower total distance from the unknown point, we predict that the unknown cookie is a shortbread cookie.

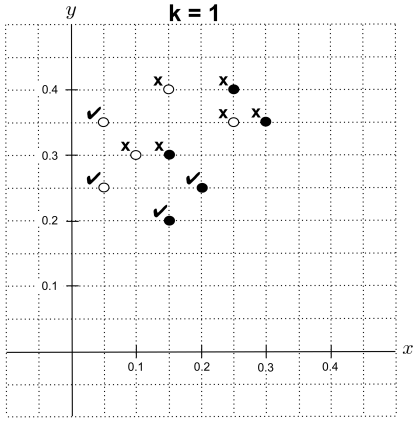
Choosing the Value of K

Now that we've learned how to make a prediction for any particular value of k, the big question is: which value of k should we use to make the prediction?

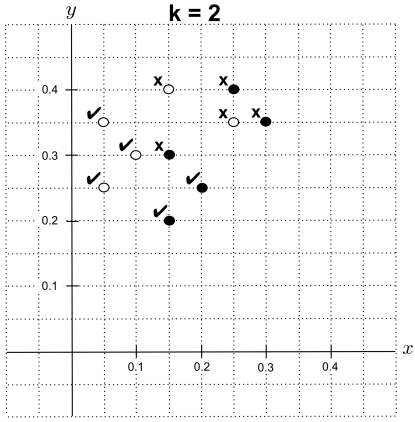
The value of k is a *parameter* in k-nearest neighbors, just like the degree is a *parameter* in polynomial regression. To choose an appropriate degree for a polynomial regression, we used leave-one-out cross validation.

We can take the same approach here. The only difference is that instead of computing the residual sum of squares (RSS), we can directly compute the accuracy by dividing the number of correct classifications by the number of total classifications.

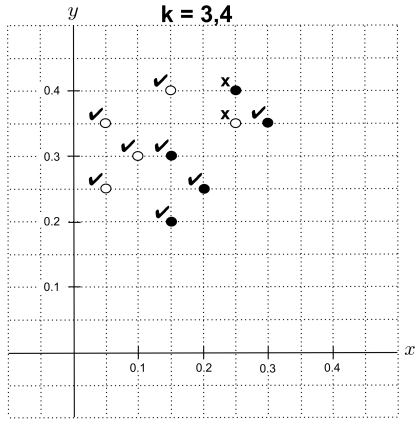
Using leave-one-out cross validation with $k = 1$, we get 4 correct classifications out of 10 total classifications, giving us an accuracy of $4/10 = 0.4$.



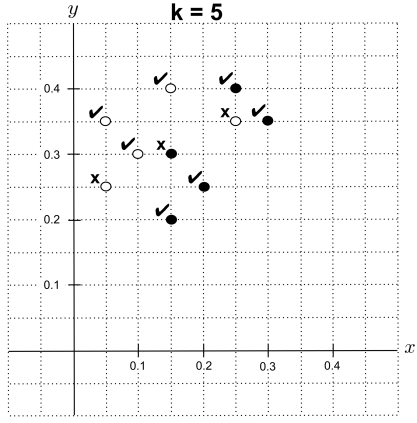
Using leave-one-out cross validation with $k = 2$, we get 5 correct classifications, giving us an accuracy of 0.5.



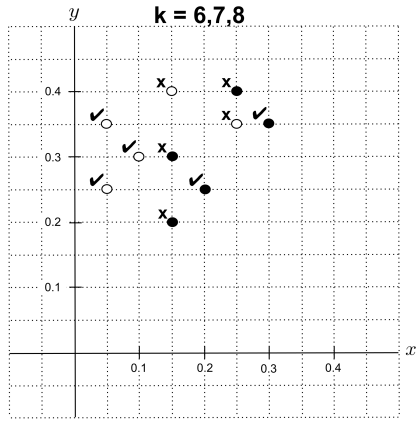
Using leave-one-out cross validation with $k = 3$ or $k = 4$, we get an accuracy of 0.8.



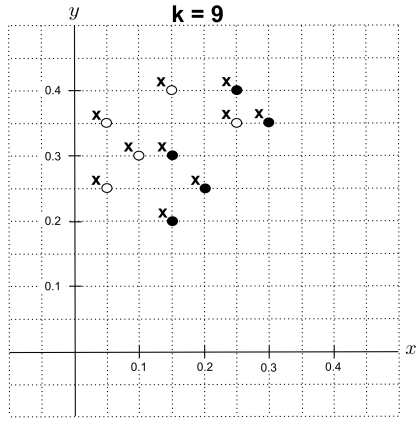
With $k = 5$, we get an accuracy of 0.7.



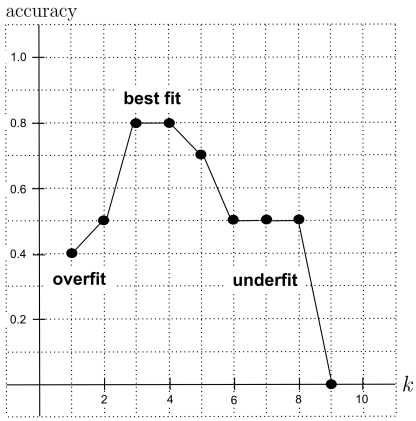
With $k = 6$, $k = 7$, or $k = 8$, we get an accuracy of 0.5.



With $k = 9$, we get an accuracy of 0. (Every point has 4 nearest neighbors of the correct class and 5 nearest neighbors of the incorrect class, leading us to predict the incorrect class.)



We organize our results in the graph below. The best fit (highest accuracy) occurred at $k = 3$ and $k = 4$, so those would be good values of k to use in our model.



When k is too *low*, the model *overfits* the data because it is too flexible (i.e. too high variance). In the extreme case of $k = 1$, the model only looks to a single nearest neighbor, which leads it to place too much trust in fine details (that could just be noise) instead of trying to understand the overall trend.

On the other hand, when k is too *high*, the model *underfits* the data because it is too rigid (i.e. too high bias). In the extreme case where k is equal to the number of points in the data set, the model totally ignores any sort of detail and will instead predict whichever class occurs most often in the data set.

In other words:

- when k is too low, the model relies too much on anecdotal evidence, and

- when k is too high, the model is unable to look beyond stereotypes.

Exercises

1. Implement the example that was worked out above. Start by classifying the unknown point $(0.25, 0.3)$ for various values of k, and then generate the leave-one-out cross validation curve.

2. Construct a cross-validation curve for the following data set, where we measure butter in cups and sugar in grams. The highest accuracy on this data set should be *lower* than the highest accuracy on the original data set. Why does using different scales for the variables cause worse performance? Run through the algorithm by hand until you notice and can describe what's happening.

Cookie Type	Cups Butter	Grams Sugar
Shortbread	0.6	200
Shortbread	0.6	300
Shortbread	0.8	250
Shortbread	1.0	400
Shortbread	1.2	350
Sugar	0.2	250
Sugar	0.2	350
Sugar	0.4	300
Sugar	0.6	400
Sugar	1.0	350

3. As demonstrated by the previous problem, k-nearest neighbors models tend to perform worse when variables have different scales. To ensure that variables are measured on the same scale, it's common to *normalize* data before fitting models.

   In particular, *min-max normalization* involves computing the minimum value of a variable, subtracting the minimum from all values, computing the new maximum value, and then dividing all values by that maximum. This ensures that the variable is measured on a scale from 0 to 1.

   Normalize the "Cups Butter" variable in the data set above using min-max normalization. Then, do the same with the "Grams Sugar" variable. Finally, construct a cross-validation curve for the normalized data set and verify that the performance has improved.

30. Naive Bayes

Naive Bayes is a classification algorithm that is grounded in *Bayesian probability*. It involves choosing the class with the highest conditional probability given the corresponding data point, assuming that all the variables in the data set are independent from each other.

Deriving the Formula

The derivation of the main formula is shown below: we apply Bayes' formula, discard the denominator $P(\text{point})$ since it doesn't depend on the class, and then express $P(\text{point} \mid \text{class})$ as a product since we're assuming that the variables in the point are independent.

$$\begin{aligned}
\text{class} &= \arg\max_{\text{class}} \left[P(\text{class} \mid \text{point})\right] \\
&= \arg\max_{\text{class}} \left[\frac{P(\text{point} \mid \text{class})P(\text{class})}{P(\text{point})}\right] \\
&= \arg\max_{\text{class}} \left[P(\text{point} \mid \text{class})P(\text{class})\right] \\
&= \arg\max_{\text{class}} \left[\prod_{\text{variables}} P(\text{variable} \mid \text{class})P(\text{class})\right] \\
&= \arg\max_{\text{class}} \left[P(\text{class}) \prod_{\text{variables}} P(\text{variable} \mid \text{class})\right]
\end{aligned}$$

The quantities in the final expression can be computed directly from our data set:

- $P(\text{class})$ is the number of records in the class, divided by the total number of records.
- $P(\text{variable} \mid \text{class})$ is the number of records in the class that have a matching variable value, divided by the total number of records in the class.

Example: Spam Detection

Let's walk through a simple concrete example in which we apply the naive Bayes algorithm to the task of spam detection. Spam detection is the canonical example for naive Bayes because it was one of the first commercial successes of naive Bayes.

Suppose that you go through 10 emails in your inbox and keep track of whether each email was a scam, along with whether it contained grammatical errors or links to other websites.

Scam	Errors	Links
No	No	No
Yes	Yes	Yes
Yes	Yes	Yes
No	No	No
No	No	Yes
Yes	Yes	Yes
No	Yes	No
No	No	Yes
Yes	Yes	No
No	No	Yes

Now, you look at 4 new emails. We can use the naive Bayes algorithm to predict whether each of these emails is a scam, based on whether it contains errors or links.

Scam	Errors	Links
?	No	No
?	Yes	Yes
?	Yes	No
?	No	Yes

First, let's consider the email with no errors and no links. We'll start by writing down the naive Bayes classification formula for this specific situation:

$$\begin{aligned}\text{class} &= \arg\max_{\text{class}} \left[P(\text{class}) \prod_{\text{variables}} P(\text{variable} \mid \text{class}) \right] \\ &= \arg\max_{\text{class}} \left[P(\text{class}) P(\text{no errors} \mid \text{class}) P(\text{no links} \mid \text{class}) \right]\end{aligned}$$

So, we have two quantities to compare:

$$P(\text{scam}) P(\text{no errors} \mid \text{scam}) P(\text{no links} \mid \text{scam})$$

vs

$$P(\text{no scam}) P(\text{no errors} \mid \text{no scam}) P(\text{no links} \mid \text{no scam})$$

Let's compute each of the probabilities in the above quantities:

- Of the 10 emails in the original data set, 4 are scams and 6 are not scams. Therefore, we have

$$P(\text{scam}) = \frac{4}{10}, \quad P(\text{no scam}) = \frac{6}{10}.$$

- Of the 4 emails in the original data set that are scams, 0 have no errors and 1 has no links. Therefore, we have

$$P(\text{no errors} \mid \text{scam}) = \frac{0}{4}, \quad P(\text{no links} \mid \text{scam}) = \frac{1}{4}.$$

- Of the 6 emails in the original data set that are not scams, 5 have no errors and 3 have no links. Therefore, we have

$$P(\text{no errors} \mid \text{no scam}) = \frac{5}{6}, \quad P(\text{no links} \mid \text{no scam}) = \frac{3}{6}.$$

Substituting these probabilities into the 2 quantities we wish to evaluate, we get

$$\begin{aligned}
&P(\text{scam})P(\text{no errors} \mid \text{scam})P(\text{no links} \mid \text{scam}) \\
&= \frac{4}{10} \cdot \frac{0}{4} \cdot \frac{1}{4} \\
&= 0
\end{aligned}$$

vs

$$\begin{aligned}
&P(\text{no scam})P(\text{no errors} \mid \text{no scam})P(\text{no links} \mid \text{no scam}) \\
&= \frac{6}{10} \cdot \frac{5}{6} \cdot \frac{3}{6} \\
&= \frac{1}{4}.
\end{aligned}$$

The "no scam" quantity gave us a greater value, so we predict that the email with no errors and no links is not a scam.

Quantity Scam vs No Scam	Scam	Errors	Links
0 vs $\frac{1}{4}$	No	No	No
	?	Yes	Yes
	?	Yes	No
	?	No	Yes

We can predict the next row in the same way:

$$\begin{aligned}
&P(\text{scam})P(\text{errors} \mid \text{scam})P(\text{links} \mid \text{scam}) \\
&= \frac{4}{10} \cdot \frac{4}{4} \cdot \frac{3}{4} \\
&= \frac{3}{10}
\end{aligned}$$

vs

$$\begin{aligned}
&P(\text{no scam})P(\text{errors} \mid \text{no scam})P(\text{links} \mid \text{no scam}) \\
&= \frac{6}{10} \cdot \frac{1}{6} \cdot \frac{3}{6} \\
&= \frac{1}{20}.
\end{aligned}$$

This time, the "scam" quantity gave us a greater value, so we predict that the email with errors and links is a scam.

Quantity Scam vs No Scam	Scam	Errors	Links
0 vs $\frac{1}{4}$	No	No	No
$\frac{3}{10}$ vs $\frac{1}{20}$	Yes	Yes	Yes
	?	Yes	No
	?	No	Yes

If we apply the naive Bayes algortithm to the remaining rows, we get the following results:

Quantity Scam vs No Scam	Scam	Errors	Links
0 vs $\frac{1}{4}$	No	No	No
$\frac{3}{10}$ vs $\frac{1}{20}$	Yes	Yes	Yes
$\frac{1}{10}$ vs $\frac{1}{20}$	Yes	Yes	No
0 vs $\frac{1}{4}$	No	No	Yes

Finally, note that in the event of a tie (i.e. both quantities give the same value), it is common to choose the class that occurred most frequently in the data set.

Exercise

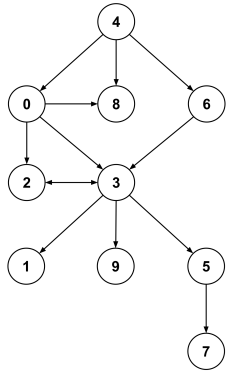
Implement the example that was worked out above.

Part V

Graphs

31. Breadth-First and Depth-First Traversals

Graphs show up all the time in computer science, so it's important to know how to work with them. For example, consider the following graph:



At its core, this graph is just a list of edges. Each edge (a, b) represents a connection from node a to node b.

```
edges = [
    (0,2), (0,3), (0,8),
    (2,3),
    (3,1), (3,2), (3,5), (3,9),
    (4,0), (4,6), (4,8),
    (5,7),
    (6,3)
]
```

Graph Class

When working with graphs, it's usually convenient and efficient to parse edges into a `Graph` class that handles operations behind the scenes.

```
>>> graph = Graph(edges)

>>> graph.get_child_ids(3)
[1, 2, 9, 5]

>>> graph.get_parent_ids(3)
[0, 2, 6]

>>> graph.get_child_ids(4)
[0, 8, 6]

>>> graph.get_parent_ids(4)
[]

>>> graph.get_child_ids(7)
[]

>>> graph.get_parent_ids(7)
[5]
```

Breadth-First and Depth-First Traversals

In addition to getting the children or parents of a particular node in the graph, it's common to need to traverse though the graph in various ways. The two most common types of traversals are **breadth first** and **depth first**.

A **breadth-first** traversal starts at a node and then visits its children, its grandchildren, its great-grandchildren, and so on. Intuitively, it proceeds outward from the root node in broad layers.

```
>>> graph.get_ids_breadth_first(4)
[4, 0, 8, 6, 2, 3, 1, 9, 5, 7]

# Note: there are other valid breadth-first traversals
# that would also be fine. For example:
[4, 8, 6, 0, 3, 2, 9, 5, 1, 7]
```

On the other hand, a **depth-first** traversal goes down the entire family tree of a single child before going down the family tree of another child. Intuitively, it proceeds outward from the root node in deep spikes.

```
>>> graph.depth_first(4)
[4, 0, 2, 3, 1, 9, 5, 7, 8, 5]

# Note: there are other valid depth-first traversals
# that would also be fine. For example:
[4, 8, 6, 3, 1, 5, 7, 9, 2, 0]
```

Implementation via Queues and Stacks

Take a moment to make sure you understand the examples above before reading on.

Breadth-first and depth-first traversals are simple to implement using *queues* and *stacks* (respectively), which are list-like data structures that follow specific conventions regarding the order in which items can be loaded and unloaded.

- In a *queue*, the first item loaded becomes the first item unloaded (i.e. first-in-first-out, just like a line at the grocery store).
- In a *stack*, the first item loaded becomes the LAST item unloaded (i.e. first-in-last-out, just like a stack of paper).

To generate a breadth-first traversal, the following algorithm can be used:

```
queue = [rootNode]
visited = {rootNode: True}
traversal = [rootNode]

while queue not empty:
    unload node from queue
    for each child of node:
        if child has not been visited:
            load child to queue and traversal
            record visit
```

Below is a concrete walkthrough showing how the algorithm above generates a breadth-first traversal from root node 4.

```
queue = [4], visited = {4:T}, traversal = [4]

unload 4 --> queue = []
- child 0 has not been visited, so visit it
    queue = [0]
    visited = {0:T,4:T}
    traversal = [4,0]
- child 8 has not been visited, so visit it
    queue = [0,8]
    visited = {0:T,4:T,8:T}
    traversal = [4,0,8]
- child 6 has not been visited, so visit it
    queue = [0,8,6]
    visited = {0:T,4:T,6:T,8:T}
    traversal = [4,0,8,6]

unload 0 --> queue = [8,6]
- child 2 has not been visited, so visit it
    queue = [8,6,2]
    visited = {0:T,2:T,4:T,6:T,8:T}
    traversal = [4,0,8,6,2]
- child 3 has not been visited, so visit it
    queue = [8,6,2,3]
    visited = {0:T,2:T,3:T,4:T,6:T,8:T}
    traversal = [4,0,8,6,2,3]

unload 8 --> queue = [6,2,3]
- (no children)

unload 6 --> queue = [2,3]
- child 3 has already been visited, so skip it

unload 2 --> queue = [3]
- (no children)

unload 3 --> queue = []
- child 1 has not been visited, so visit it
    queue = [1]
    visited = {0:T,1:T,2:T,3:T,4:T,6:T,8:T}
    traversal = [4,0,8,6,2,3,1]
- child 9 has not been visited, so visit it
    queue = [1,9]
    visited = {0:T,1:T,2:T,3:T,4:T,6:T,8:T,9:T}
    traversal = [4,0,8,6,2,3,1,9]
```

```
- child 5 has not been visited, so visit it
    queue = [1,9,5]
    visited = {0:T,1:T,2:T,3:T,4:T,5:T,6:T,8:T,9:T}
    traversal = [4,0,8,6,2,3,1,9,5]

unload 1 --> queue = [9,5]
- (no children)

unload 9 --> queue = [5]
- (no children)

unload 5 --> queue = []
- child 7 has not been visited, so visit it
    queue = [7]
    visited = {0:T,1:T,2:T,3:T,4:T,5:T,6:T,7:T,8:T,9:T}
    traversal = [4,0,8,6,2,3,1,9,5,7]

unload 7 --> queue = []
- (no children)

queue is empty --> nothing left to unload --> DONE
Final traversal: [4,0,8,6,2,3,1,9,5,7]
```

Generating a depth-first traversal is almost exactly the same. The only difference is that we use a stack instead of a queue. A concrete example illustrating the difference between a stack and a queue is given below.

1. With a queue, we load on the right and unload on the left. For example, given a queue `[1,2]`, loading `3` gives `[1,2,3]`. If we unload, the unloaded element is `1` and the remaining queue is `[2,3]`.

2. With a stack, we load on the right and unload on the right. For example, given a stack `[1,2]`, loading `3` gives `[1,2,3]`. If we unload, the unloaded element is `3` and the remaining stack is `[1,2]`.

Time Complexity

Because breadth-first and depth-first traversals both visit each node once and only once, they both have time complexity $O(n)$ where n is the number of nodes in the graph.

Exercises

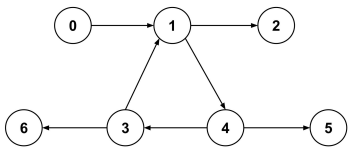
Implement a `Graph` class with all the methods described above, and make sure to test it on several different types of graphs. You can use the graph shown here as one of your test cases, but you should also test several significantly different cases (e.g. cycles, an instance of two arrows pointing the opposite way, a disconnected graph, etc).

32. Distance and Shortest Paths in Unweighted Graphs

The **distance** between two nodes in a graph is the fewest number of edges that must be crossed to travel from one node to the other. A **path** between two nodes is a sequence of nodes that can be traversed to get from one node to the other, traveling along edges. The **shortest path** is the path with the shortest distance.

Demonstration

Below is a demonstration of distances and shortest paths in a particular graph.



```
>>> graph.calc_distance(0,3)
3
>>> graph.calc_distance(3,5)
3
>>> graph.calc_distance(0,5)
3
>>> graph.calc_distance(4,1)
2
>>> graph.calc_distance(2,4)
False

>>> graph.calc_shortest_path(0,3)
[0, 1, 4, 3]
>>> graph.calc_shortest_path(3,5)
[3, 1, 4, 5]
>>> graph.calc_shortest_path(0,5)
[0, 1, 4, 5]
>>> graph.calc_shortest_path(4,1)
[4, 3, 1]
>>> graph.calc_shortest_path(2,4)
False
```

Implementation

The key to implementing these methods is to first create a method `graph.set_distance_and_previous(idx)` that does a breadth-first traversal from the node at the given index and sets the attributes `node.distance` and `node.previous` for each node encountered during the traversal.

1. Start at the node whose index is `idx`.
2. In the breadth-first traversal, you'll end up visiting all the children of that node, those children's children, and so on.
3. Whenever you add a child to the queue, set the child's `previous` attribute to be the parent node that the child is coming from, and set the child's `distance` attribute to be one more than that parent's distance.
4. When this is all done, each node's `distance` attribute will represent its distance from the initial starting node, and each node's `previous` attribute will represent the node that comes before it if you're traveling to it on the shortest path.

Note that this will require you to write a `Node` class and create an instance for every node in your graph. It's best to do this at the very beginning when you first initialize the graph.

When you run `graph.calc_distance(from_idx, to_idx)` or `graph.calc_shortest_path(from_idx, to_idx)`, the first step will always be to run `graph.set_distance_and_previous(from_idx)`. Once you have the `distance` and `previous` attributes set, you can use them to easily compute distances and shortest paths between nodes:

- `calc_distance(from_idx, to_idx)`
  Simply return the `distance` attribute of the "to" node.
- `calc_shortest_path(from_idx, to_idx)`

    1. Start at the "to" node and repeatedly go to the previous node until you get to the "from" node.
    2. Keep track of all the nodes you visit (this is the shortest path in reverse).
    3. Return the path in order from the "from" node to the "to" node. (You'll have to reverse the reversed path that you found in the previous step.)

Exercises

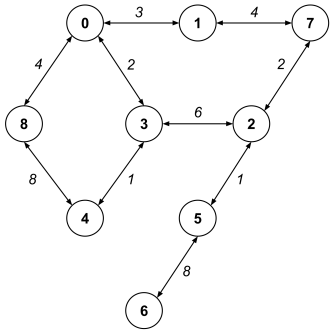
Extend your `Graph` class to include the methods `calc_distance` and `calc_shortest_path`. As always, be sure to write tests.

33. Dijkstra's Algorithm for Distance and Shortest Paths in Weighted Graphs

In a **weighted graph**, every edge is assigned a value called a **weight**.

Initializing a Weighted Graph

For example, consider the following weighted graph:



A weighted graph can be initialized with a weights dictionary instead of an edges list. The edges list just had a list of edges, whereas the weights dictionary will have its keys as edges and its values as the weights of those edges.

```
>>> weights = {
    (0,1): 3, (1,0): 3,
    (1,7): 4, (7,1): 4,
    (7,2): 2, (2,7): 2,
    (2,5): 1, (5,2): 1,
    (5,6): 8, (6,5): 8,
    (0,3): 2, (3,0): 2,
    (3,2): 6, (2,3): 6,
    (3,4): 1, (4,3): 1,
    (4,8): 8, (8,4): 8,
    (8,0): 4, (0,8): 4
}

>>> weighted_graph = WeightedGraph(weights)
```

Distance and Shortest Paths in Weighted Graphs

In a weighted graph, the distance between two nodes is the sum of the weights on the shortest path between them. For example, the distance from node 8 to node 4 is 7 because the shortest path is $8 \xrightarrow{4} 0 \xrightarrow{2} 3 \xrightarrow{1} 4$.

In particular, notice that the shortest path is NOT $8 \xrightarrow{8} 4$ because the distance along this path is 8.

```
>>> [weighted_graph.calc_distance(8,n) for n in range(9)
    ]
[4, 7, 12, 6, 7, 13, 21, 11, 0]

>>> weighted_graph.calc_shortest_path(8,4)
[8, 0, 3, 4]

>>> weighted_graph.calc_shortest_path(8,7)
[8, 0, 1, 7]

>>> weighted_graph.calc_shortest_path(8,6)
[8, 0, 3, 2, 5, 6]
```

Dijkstra's Algorithm for Distance

The underlying algorithms for `calc_distance` and `calc_shortest_path` are a bit more complicated for weighted graphs than for unweighted graphs.

To implement `calc_distance(from_idx, to_idx)` we need to use **Dijkstra's algorithm**, which works by assigning each node an initial guess for its distance and then repeatedly updating those guesses until they actually represent the distances to those nodes.

1. When setting initial guesses, the "from" node is assigned a distance value of 0, and all other nodes are assigned distance values of ∞ (just use a large number like 9999999999).
2. Set the `current_node` to be the "from" node.

3. Loop through the `current_node`'s unvisited children and update their distances as `child.distance = min(child.distance, current_node.distance + edge_weight)`.

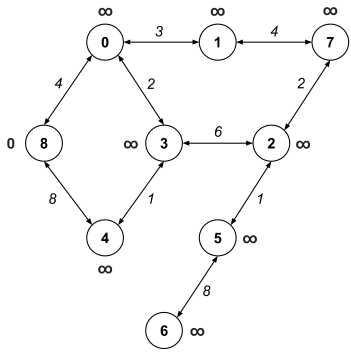
4. Update the `current_node` to be the *unvisited* node with the *smallest* distance value (not necessarily a child node).

5. If the ending node has not been visited yet, then return to step 3.

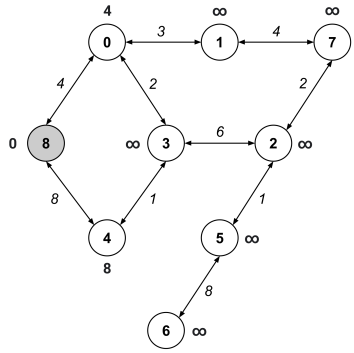
6. Return the `distance` attribute of the "to" node.

Let's demonstrate each iteration of Dijkstra's algorithm when computing the distance from node 8 to node 6.

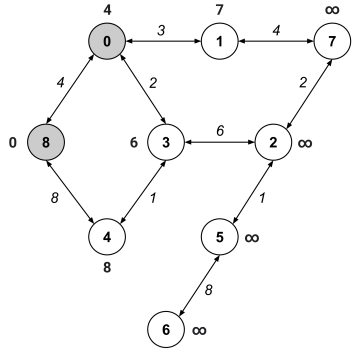
First, we set the initial guesses for the distance values. Since we're starting at node 8, we already know that it's a distance of 0 from itself. All the other nodes get initial guesses of infinity.



Now, we visit node 8 (the "from" node) and update the distance values on its children.



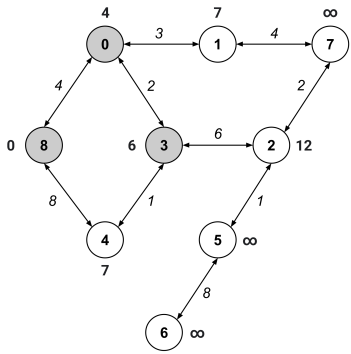
The next node we visit should be the unvisited node with the smallest distance value. This would be node 0, whose distance value is 4. We visit this node and update the distance values on its unvisited children.



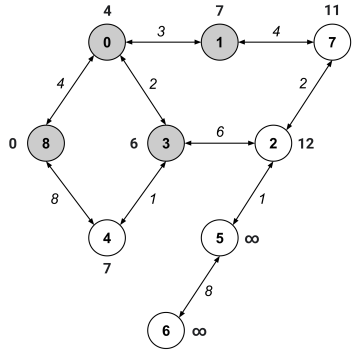
Again, we visit the unvisited node with the smallest distance value. This time, it's node 3.

Notice that when we update the distance values on its unvisited children, node 4's distance value decreases from 8 to 7. Whenever a distance value decreases like this, it means that the nodes we've traversed contain a shorter path than a path we found earlier.

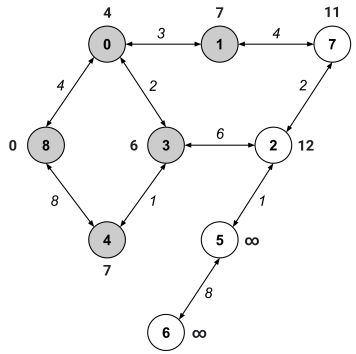
In our first iteration, we found a path $8 \xrightarrow{8} 4$ with distance 8. Now, we found a path $8 \xrightarrow{4} 0 \xrightarrow{2} 3 \xrightarrow{1} 4$ with distance 7.

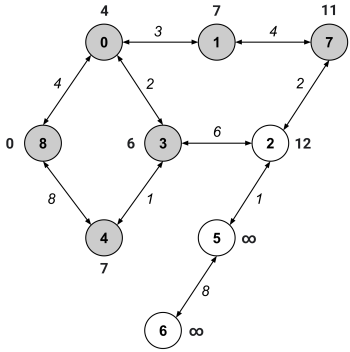


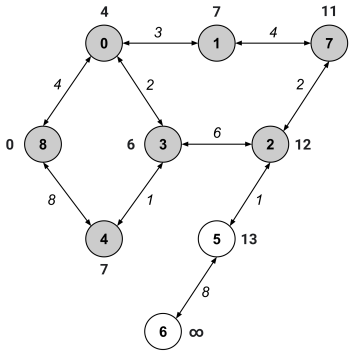
As usual, we visit the unvisited node with the smallest distance value. This time, we can visit either node 1 or node 4 (they are the unvisited nodes with the smallest distance values). We'll arbitrarily choose to visit node 1 and update the distance values of its children.

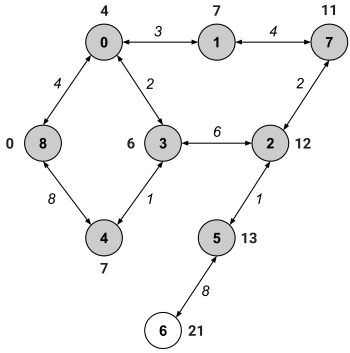


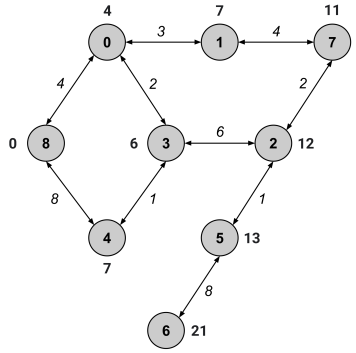
We keep on repeating this same procedure until the "to" node (node 6) has been visited.




4
7
11
0
3
1
4
7
4
2
2
0
8
6
3
6
2
12
8
1
1
4
5
∞
7
8
6
∞


4
7
11
0
3
1
4
7
4
2
2
0
8
6
3
6
2
12
8
1
1
4
5
13
7
8
6
∞


4
7
11
0
3
1
4
7
4
2
2
0
8
6
3
6
2
12
8
1
1
4
5
13
7
8
6
21

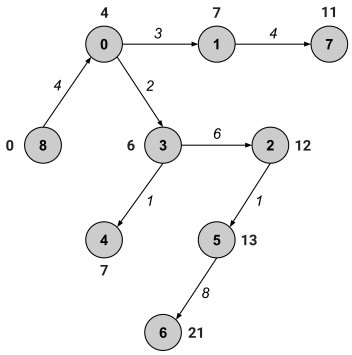


We've visited node 6, and its distance value is 21. This means that the distance from node 8 to node 6 is 21.

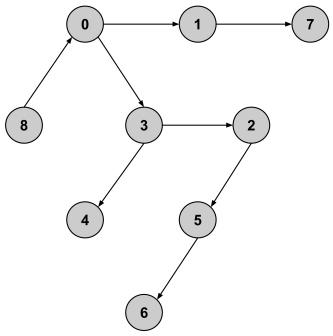
Computing Shortest Paths via the Shortest-Path Tree

But what is the specific shortest path from node 8 to node 6 that gives us this distance of 21?

To find the shortest path, we first construct the **shortest-path tree** by discarding any edge `(a,b)` whose weight does not match the corresponding difference between distance values, `b.distance - a.distance`.



Once we have the shortest-path tree, we've effectively reduced our problem down to a problem that we've already solved: finding the shortest path between two nodes in an unweighted graph.



Indeed, we can see that the shortest path from node 8 to node 6 in the tree above is given by $8 \to 0 \to 3 \to 2 \to 5 \to 6$. And indeed, in the weighted graph, the sum of weights along this path is 21.

Exercises

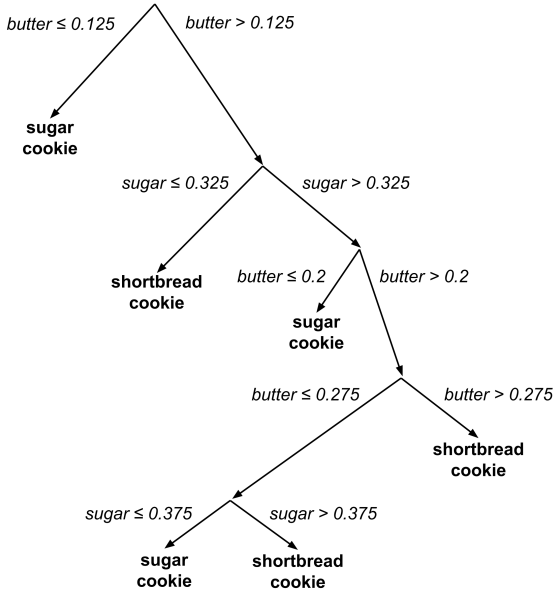
Implement a `WeightedGraph` class with the methods `calc_distance` and `calc_shortest_path`. As always, be sure to write tests.

34. Decision Trees

A **decision tree** is a graphical flowchart that represents a sequence of nested "if-then" decision rules. To illustrate, first recall the following cookie data set that was introduced during the discussion of k-nearest neighbors:

Cookie Type	Portion Butter	Portion Sugar
Shortbread	0.15	0.2
Shortbread	0.15	0.3
Shortbread	0.2	0.25
Shortbread	0.25	0.4
Shortbread	0.3	0.35
Sugar	0.05	0.25
Sugar	0.05	0.35
Sugar	0.1	0.3
Sugar	0.15	0.4
Sugar	0.25	0.35

The following decision tree was algorithmically constructed to classify an unknown cookie as a shortbread cookie or sugar cookie based on its portions of butter and sugar.



Using a Decision Tree

To use the decision tree to classify an unknown cookie, we start at the top of the tree and then repeatedly go downwards and left or right depending on the values of x and y.

For example, suppose we have a cookie with 0.25 portion butter and 0.35 portion sugar. To classify this cookie, we start at the top of the tree and then go

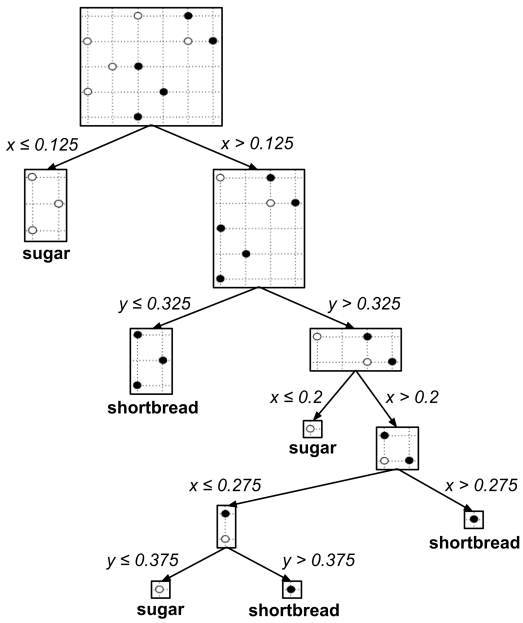
1. right $(\text{butter} > 0.125)$,
2. right $(\text{sugar} > 0.325)$,
3. right $(\text{butter} > 0.2)$,

4. left (butter ≤ 0.275),
5. left (sugar ≤ 0.375),

reaching the prediction that the cookie is a sugar cookie.

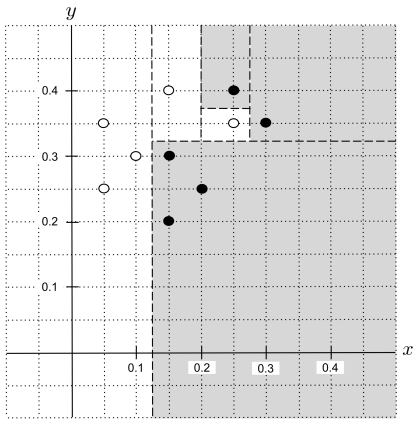
Classification Boundary

Let's take a look at how the decision tree classifies the points in our data set:



We can visualize this in the plane by drawing the **classification boundary**, shading the regions whose points would be classified as

shortbread cookies and keeping unshaded the regions whose points would be classified as sugar cookies. Each dotted line corresponds to a **split** in the tree.



Building a Decision Tree: Reducing Impurity

The algorithm for building a decision tree is conceptually simple. The goal is to make the simplest tree such that the leaf nodes *pure* in the sense that they only contain data points from one class. So, we repeatedly split *impure* leaf nodes in the way that most quickly reduces the impurity.

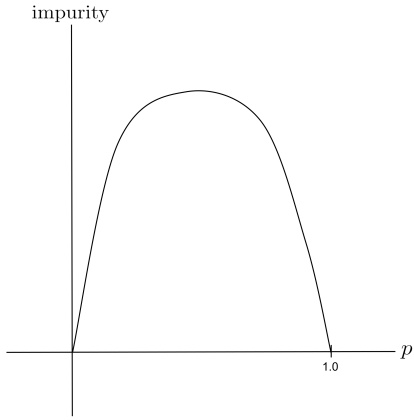
Intuitively, a node has 0 impurity when all of its data points come from one class. On the other hand, a node has maximum impurity when an equal amount of its data points come from each class.

To quantify a node's impurity, all we have to do is count up the proportion p of the node's data points that are from one particular

class and then apply a function that transforms p into a measure of impurity.

- If $p = 0$ or $p = 1$, then the node has no impurity since its data points are entirely from one class.
- If $p = 0.5$, then the node has maximum impurity since half of its data points come from one class and the other half comes from the other class.

Graphically, our function should look like this:



Two commonly used functions that yield the above graph are **Gini impurity**, defined as

$$G(p) = 1 - p^2 - (1 - p)^2,$$

and **information entropy**, defined as

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p).$$

Although these functions may initially look a little complicated, note that their forms permit them to be easily generalized to situations where we have more than two classes:

$$G = 1 - \sum_i p_i^2$$

$$H = -\sum_i p_i \log_2 p_i,$$

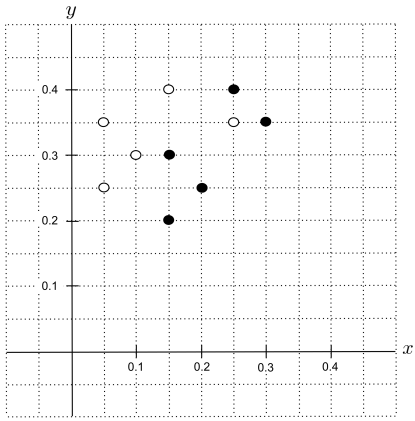
where p_i is the proportion of the ith class. (In our situation we only have two classes with proportions $p_1 = p$ and $p_2 = 1 - p$.)

Worked Example: Split o

As we walk through the algorithm for building our decision tree, we'll use Gini impurity since it simplifies nicely in the case of two classes, making it more amenable to manual computation:

$$\begin{aligned} G(p) &= 1 - p^2 - (1-p)^2 \\ &= 2p(1-p) \end{aligned}$$

Initially, our decision tree is just a single root node, i.e. a "stump" with no splits. It contains our full data set, shown below.



Worked Example: Split 1

Remember that our goal is to repeatedly split *impure* leaf nodes in the way that most quickly reduces the impurity. To find the split that most quickly reduces the impurity, we loop over all possible splits and compare the impurity before the split to the impurity after the split.

The impurity before the split is the same for all possible splits, so we will calculate it first. In the graph above there are 5 points that represent shortbread cookies and 5 points that represent sugar cookies, so $p = \frac{5}{5+5} = \frac{1}{2}$ and the impurity is computed as

$$\begin{aligned} G_{\text{before}} &= G\left(\frac{1}{2}\right) \\ &= 2\left(\frac{1}{2}\right)\left(1-\frac{1}{2}\right) \\ &= 0.5. \end{aligned}$$

Now, let's find all the possible splits. To find the values of x that could be chosen for splits, we first find all the distinct values of x that are hit by points and put them in order:

$$x = 0.05, 0.1, 0.15, 0.2, 0.25, 0.3$$

The possible splits along the x-axis are the midpoints between consecutive entries in the list above:

$$x_{\text{split}} = 0.075, 0.125, 0.175, 0.225, 0.275$$

Performing the same process for y-coordinates, we get the following:

$$y = 0.2, 0.25, 0.3, 0.35, 0.4$$

$$y_{\text{split}} = 0.225, 0.275, 0.325, 0.375$$

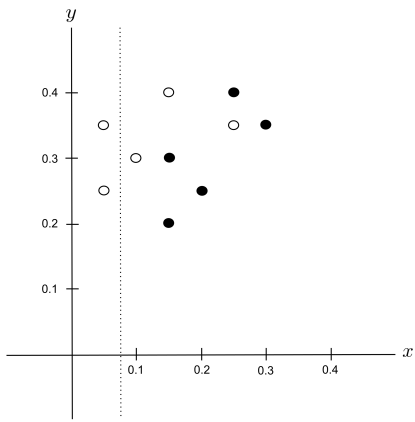
Let's go through each possible split and measure the impurity after the split. In general, the impurity after the split is measured as a weighted average of the new leaf nodes resulting from the split:

$$\begin{aligned}\text{impurity after} = &\ (\text{portion data points in} \leq \text{node}) \times (\text{impurity of} \leq \text{node}) \\ &+ (\text{portion data points in} > \text{node}) \times (\text{impurity of} > \text{node})\end{aligned}$$

The formula above can be represented more concisely as

$$G_{\text{after}} = p_{\leq} G_{\leq} + p_{>} G_{>}.$$

Possible Split: $x_{\text{split}} = 0.075$



The $x \leq 0.05$ node would be pure with 2 sugar cookies, giving an impurity of

$$G_{\leq} = 2\left(\frac{0}{2}\right)\left(\frac{2}{2}\right) = 0.$$

On the other hand, the $x > 0.05$ node would contain 5 shortbread cookies and 3 sugar cookies, giving an impurity of

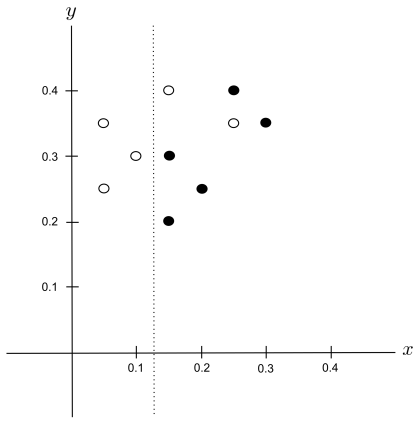
$$G_{>} = 2\left(\frac{5}{8}\right)\left(\frac{3}{8}\right) = \frac{30}{64}.$$

The $\leq$ node would contain 2 points while the $>$ node would contain 8 points, giving proportions $p_{\leq} = \frac{2}{10}$ and $p_{>} = \frac{8}{10}$.

Finally, the impurity after the split would be

$$\begin{aligned} G_{\text{after}} &= p_{\leq}G_{\leq} + p_{>}G_{>} \\ &= \left(\frac{2}{10}\right)(0) + \left(\frac{8}{10}\right)\left(\frac{30}{64}\right) \\ &= 0.375. \end{aligned}$$

Possible Split: $x_{\text{split}} = 0.125$



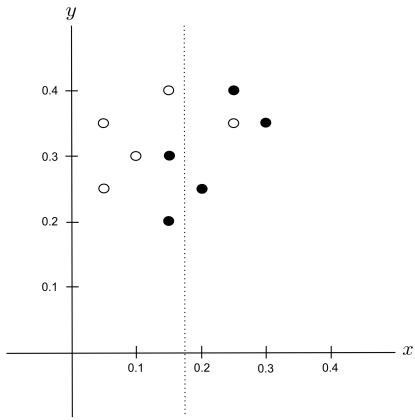
Repeating the same process, we have $p_{\leq} = \dfrac{3}{10}$ and $p_{>} = \dfrac{7}{10}$ get we get the following impurities:

$$G_{\leq} = 2\left(\frac{0}{3}\right)\left(\frac{3}{3}\right) = 0$$

$$G_{>} = 2\left(\frac{5}{7}\right)\left(\frac{2}{7}\right) = \frac{20}{49}$$

$$G_{\text{after}} = \left(\frac{3}{10}\right)(0) + \left(\frac{7}{10}\right)\left(\frac{20}{49}\right) \approx 0.286.$$

Possible Split: $x_{\text{split}} = 0.175$

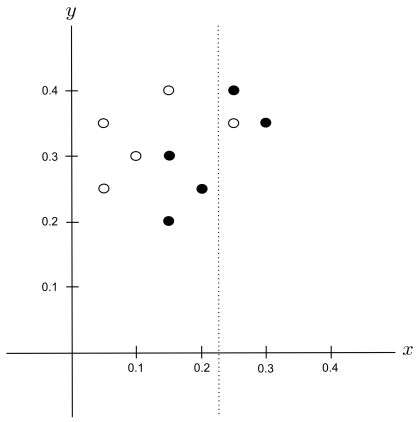


$$G_{\leq} = 2\left(\frac{2}{6}\right)\left(\frac{4}{6}\right) = \frac{16}{36}$$

$$G_{>} = 2\left(\frac{3}{4}\right)\left(\frac{1}{4}\right) = \frac{6}{16}$$

$$G_{\text{after}} = \left(\frac{6}{10}\right)\left(\frac{16}{36}\right) + \left(\frac{4}{10}\right)\left(\frac{6}{16}\right) \approx 0.417$$

Possible Split: $x_{\text{split}} = 0.225$

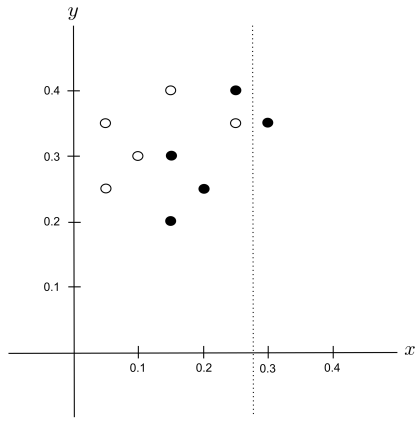


$$G_{\leq} = 2\left(\frac{3}{7}\right)\left(\frac{4}{7}\right) = \frac{24}{49}$$

$$G_{>} = 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) = \frac{4}{9}$$

$$G_{\text{after}} = \left(\frac{7}{10}\right)\left(\frac{24}{49}\right) + \left(\frac{3}{10}\right)\left(\frac{4}{9}\right) \approx 0.476$$

Possible Split: $x_{\text{split}} = 0.275$

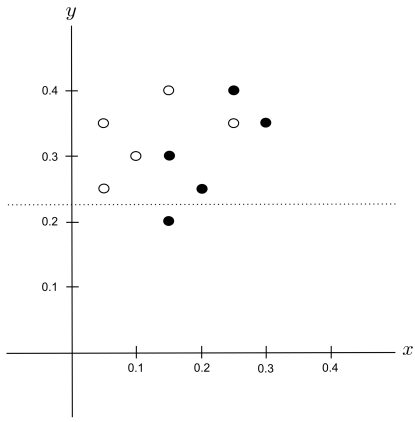


$$G_{\leq} = 2\left(\frac{4}{9}\right)\left(\frac{5}{9}\right) = \frac{40}{81}$$

$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{\text{after}} = \left(\frac{9}{10}\right)\left(\frac{40}{81}\right) + \left(\frac{1}{10}\right)(0) \approx 0.444$$

Possible Split: $y_{\text{split}} = 0.225$

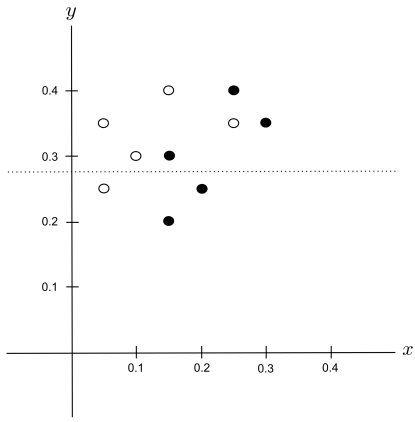


$$G_{\leq} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{>} = 2\left(\frac{4}{9}\right)\left(\frac{5}{9}\right) = \frac{40}{81}$$

$$G_{\text{after}} = \left(\frac{1}{10}\right)(0) + \left(\frac{9}{10}\right)\left(\frac{40}{81}\right) \approx 0.444$$

Possible Split: $y_{\text{split}} = 0.275$

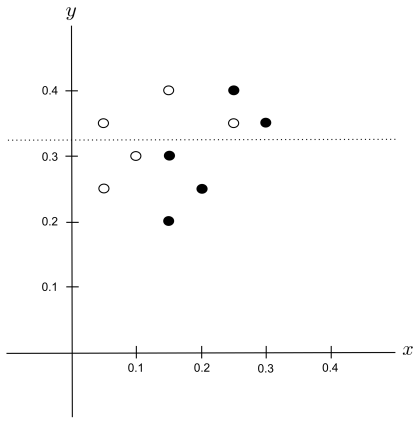


$$G_{\leq} = 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) = \frac{4}{9}$$

$$G_{>} = 2\left(\frac{3}{7}\right)\left(\frac{4}{7}\right) = \frac{24}{49}$$

$$G_{\text{after}} = \left(\frac{3}{10}\right)\left(\frac{4}{9}\right) + \left(\frac{7}{10}\right)\left(\frac{24}{49}\right) \approx 0.476$$

Possible Split: $y_{\text{split}} = 0.325$

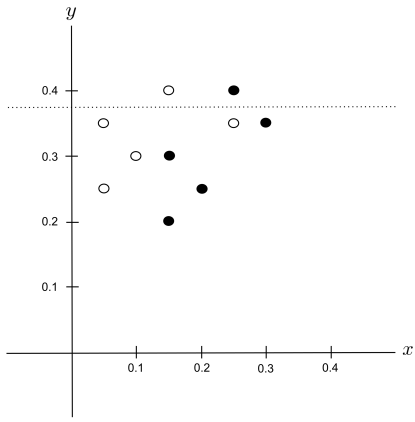


$$G_{\leq} = 2\left(\frac{3}{5}\right)\left(\frac{2}{5}\right) = \frac{12}{25}$$

$$G_{>} = 2\left(\frac{2}{5}\right)\left(\frac{3}{5}\right) = \frac{12}{25}$$

$$G_{\text{after}} = \left(\frac{5}{10}\right)\left(\frac{12}{25}\right) + \left(\frac{5}{10}\right)\left(\frac{12}{25}\right) = 0.48$$

Possible Split: $y_{\text{split}} = 0.375$



$$G_{\leq} = 2\left(\frac{4}{8}\right)\left(\frac{4}{8}\right) = \frac{32}{64}$$

$$G_{>} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{2}{4}$$

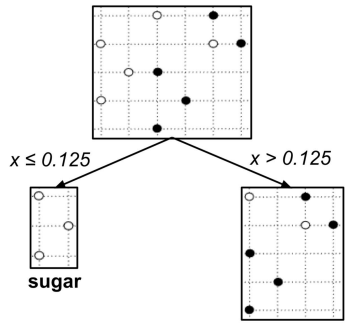
$$G_{\text{after}} = \left(\frac{8}{10}\right)\left(\frac{32}{64}\right) + \left(\frac{2}{10}\right)\left(\frac{2}{4}\right) = 0.5$$

Best Split

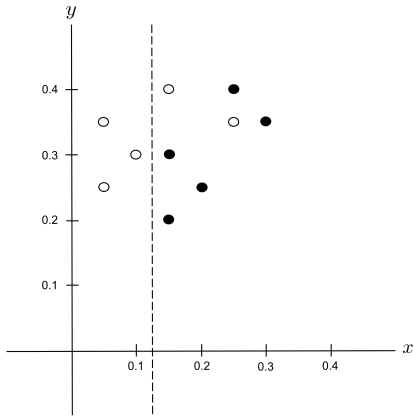
Remember that the initial impurity before splitting was $G_{\text{before}} = 0.5$. Let's compute how much each potential split would decrease the impurity:

Split	$G_{\text{before}} - G_{\text{after}}$	
$x_{\text{split}} = 0.075$	$0.5 - 0.375$	$= 0.125$
$x_{\text{split}} = 0.125$	$0.5 - 0.286$	$= \mathbf{0.214}$
$x_{\text{split}} = 0.175$	$0.5 - 0.417$	$= 0.083$
$x_{\text{split}} = 0.225$	$0.5 - 0.476$	$= 0.024$
$x_{\text{split}} = 0.275$	$0.5 - 0.444$	$= 0.056$
$y_{\text{split}} = 0.225$	$0.5 - 0.444$	$= 0.056$
$y_{\text{split}} = 0.275$	$0.5 - 0.476$	$= 0.024$
$y_{\text{split}} = 0.325$	$0.5 - 0.48$	$= 0.02$
$y_{\text{split}} = 0.375$	$0.5 - 0.5$	$= 0$

According to the table above, the best split is $x_{\text{split}} = 0.125$ since it decreases the impurity the most. We integrate this split into our decision tree:



This decision tree can be visualized in the plane as follows:



Worked Example: Split 2

Again, we repeat the process and split any impure leaf nodes in the tree. There is exactly one impure leaf node ($x > 0.125$) and it contains 5 shortbread and 2 sugar cookies, giving an impurity of

$$\begin{aligned} G_{\text{before}} &= G\left(\frac{5}{7}\right) \\ &= 2\left(\frac{5}{7}\right)\left(\frac{2}{7}\right) \\ &\approx 0.408 \end{aligned}$$

To find the possible splits, we first find the distinct values of x and y that are hit by points in this node and put them in order:

$$x = 0.15, 0.2, 0.25, 0.3$$

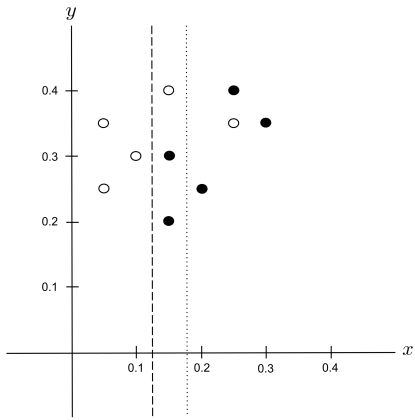
$$y = 0.2, 0.25, 0.3, 0.35, 0.4$$

The possible splits are the midpoints between consecutive entries in the list above:

$$x_{\text{split}} = 0.175, 0.225, 0.275$$

$$y_{\text{split}} = 0.225, 0.275, 0.325, 0.375$$

Possible Split: $x_{\text{split}} = 0.175$



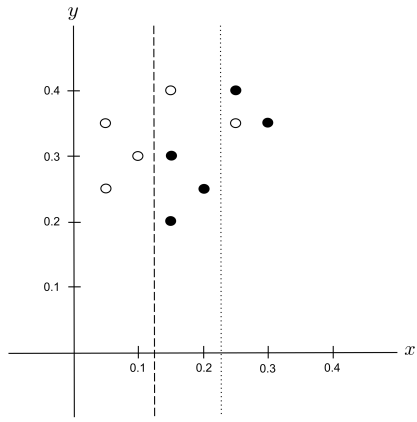
Remember that we are only splitting the region covered by the $x > 0.125$ node, which contains 7 data points. We can ignore the 3 data points left of the hard dotted line, since they are not contained within the node that we are splitting.

$$G_{\leq} = 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) = \frac{4}{9}$$

$$G_{>} = 2\left(\frac{3}{4}\right)\left(\frac{1}{4}\right) = \frac{6}{16}$$

$$G_{\text{after}} = \left(\frac{3}{7}\right)\left(\frac{4}{9}\right) + \left(\frac{4}{7}\right)\left(\frac{6}{16}\right) \approx 0.405$$

Possible Split: $x_{\text{split}} = 0.225$

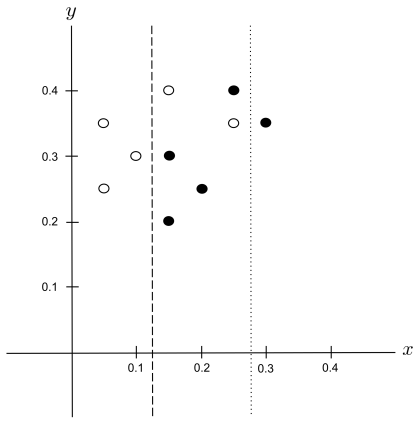


$$G_{\leq} = 2\left(\frac{3}{4}\right)\left(\frac{1}{4}\right) = \frac{6}{16}$$

$$G_{>} = 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) = \frac{4}{9}$$

$$G_{\text{after}} = \left(\frac{4}{7}\right)\left(\frac{6}{16}\right) + \left(\frac{3}{7}\right)\left(\frac{4}{9}\right) \approx 0.405$$

Possible Split: $x_{\text{split}} = 0.275$

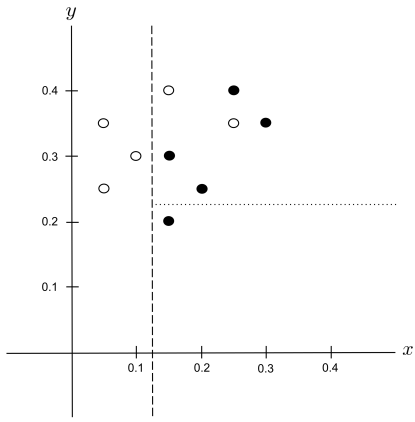


$$G_{\leq} = 2\left(\frac{4}{6}\right)\left(\frac{2}{6}\right) = \frac{16}{36}$$

$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{\text{after}} = \left(\frac{6}{7}\right)\left(\frac{16}{36}\right) + \left(\frac{1}{7}\right)(0) \approx 0.381$$

Possible Split: $y_{\text{split}} = 0.225$

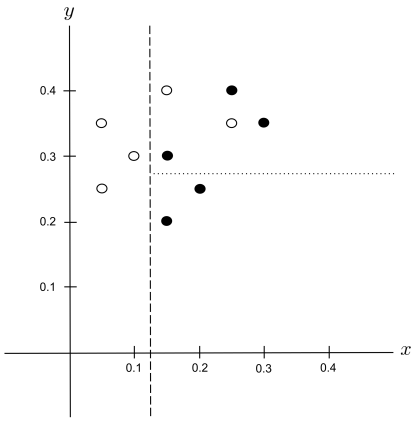


$$G_{\leq} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{>} = 2\left(\frac{4}{6}\right)\left(\frac{2}{6}\right) = \frac{16}{36}$$

$$G_{\text{after}} = \left(\frac{1}{7}\right)(0) + \left(\frac{6}{7}\right)\left(\frac{16}{36}\right) \approx 0.381$$

Possible Split: $y_{\text{split}} = 0.275$

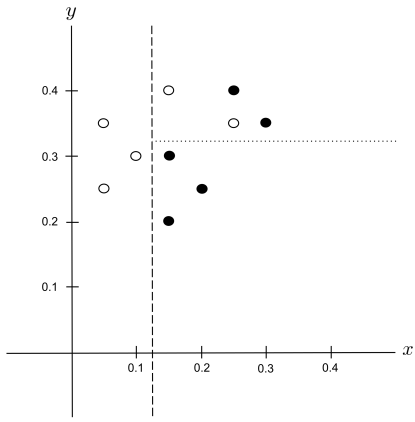


$$G_{\leq} = 2\left(\frac{2}{2}\right)\left(\frac{0}{2}\right) = 0$$

$$G_{>} = 2\left(\frac{3}{5}\right)\left(\frac{2}{5}\right) = \frac{12}{25}$$

$$G_{\text{after}} = \left(\frac{2}{7}\right)(0) + \left(\frac{5}{7}\right)\left(\frac{12}{25}\right) \approx 0.343$$

Possible Split: $y_{\text{split}} = 0.325$

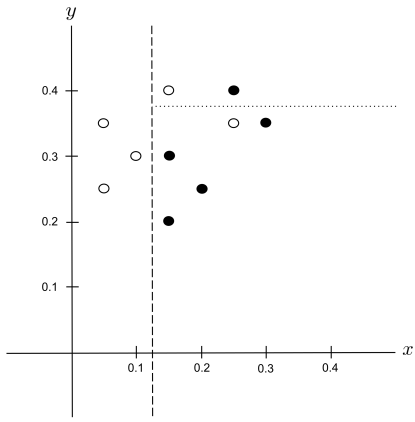


$$G_{\leq} = 2\left(\frac{3}{3}\right)\left(\frac{0}{3}\right) = 0$$

$$G_{>} = 2\left(\frac{2}{4}\right)\left(\frac{2}{4}\right) = \frac{8}{16}$$

$$G_{\text{after}} = \left(\frac{3}{7}\right)(0) + \left(\frac{4}{7}\right)\left(\frac{8}{16}\right) \approx 0.286$$

Possible Split: $y_{\text{split}} = 0.375$



$$G_{\leq} = 2\left(\frac{4}{5}\right)\left(\frac{1}{5}\right) = \frac{8}{25}$$

$$G_{>} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{1}{2}$$

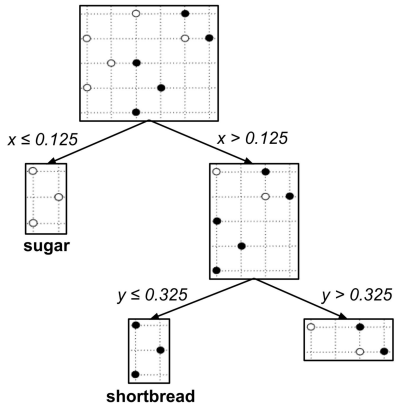
$$G_{\text{after}} = \left(\frac{5}{7}\right)\left(\frac{8}{25}\right) + \left(\frac{2}{7}\right)\left(\frac{1}{2}\right) \approx 0.371$$

Best Split

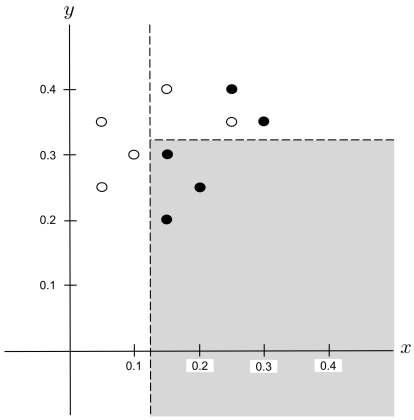
The best split is $y_{\text{split}} = 0.325$ since it decreases the impurity the most.

Split	$G_{\text{before}} - G_{\text{after}}$	
$x_{\text{split}} = 0.175$	$0.408 - 0.405$	$= 0.003$
$x_{\text{split}} = 0.225$	$0.408 - 0.405$	$= 0.003$
$x_{\text{split}} = 0.275$	$0.408 - 0.381$	$= 0.027$
$y_{\text{split}} = 0.225$	$0.408 - 0.381$	$= 0.027$
$y_{\text{split}} = 0.275$	$0.408 - 0.343$	$= 0.065$
$y_{\text{split}} = 0.325$	$0.408 - 0.286$	$= \mathbf{0.122}$
$y_{\text{split}} = 0.375$	$0.408 - 0.371$	$= 0.037$

We integrate this split into our decision tree:



This decision tree can be visualized in the plane as follows:



Worked Example: Split 3

Again, we repeat the process and split any impure leaf nodes in the tree. There is exactly one impure leaf node ($x > 0.125 \to y > 0.325$) and it contains 2 shortbread and 2 sugar cookies, giving an impurity of

$$\begin{aligned} G_{\text{before}} &= G\left(\frac{2}{4}\right) \\ &= 2\left(\frac{2}{4}\right)\left(\frac{2}{4}\right) \\ &= 0.5. \end{aligned}$$

To find the possible splits, we first find the distinct values of x and y that are hit by points in this node and put them in order:

$$x = 0.15, 0.25, 0.3$$

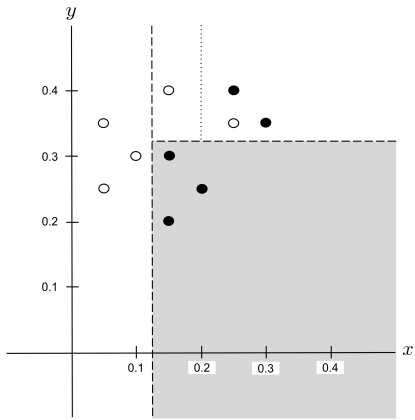
$$y = 0.35, 0.4$$

The possible splits are the midpoints between consecutive entries in the list above:

$$x_{\text{split}} = 0.2, 0.275$$

$$y_{\text{split}} = 0.375$$

Possible Split: $x_{\text{split}} = 0.2$



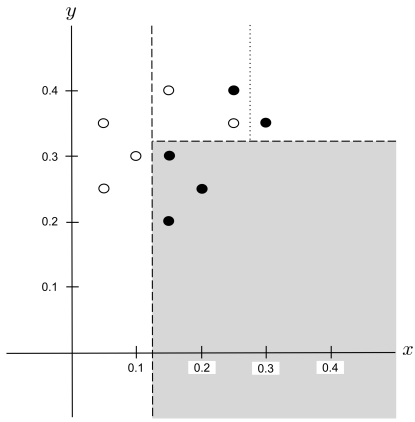
Remember that we are only splitting the region covered by the $x > 0.125 \rightarrow y > 0.325$ node, which contains 4 data points. We can ignore the 6 data points outside of this region, since they are not contained within the node that we are splitting.

$$G_{\leq} = 2\left(\frac{0}{1}\right)\left(\frac{1}{1}\right) = 0$$

$$G_{>} = 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) = \frac{4}{9}$$

$$G_{\text{after}} = \left(\frac{1}{4}\right)(0) + \left(\frac{3}{4}\right)\left(\frac{4}{9}\right) \approx 0.333$$

Possible Split: $x_{\text{split}} = 0.275$

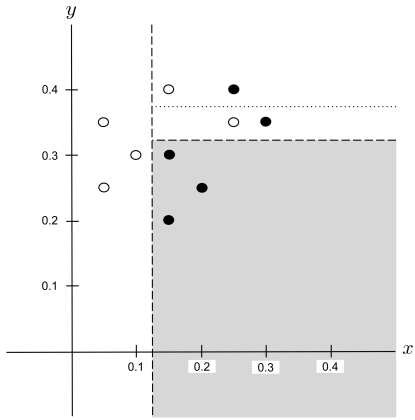


$$G_{\leq} = 2\left(\frac{1}{3}\right)\left(\frac{2}{3}\right) = \frac{4}{9}$$

$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{\text{after}} = \left(\frac{3}{4}\right)\left(\frac{4}{9}\right) + \left(\frac{1}{4}\right)(0) \approx 0.333$$

Possible Split: $y_{\text{split}} = 0.375$



$$G_{\leq} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{2}{4}$$

$$G_{>} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{2}{4}$$

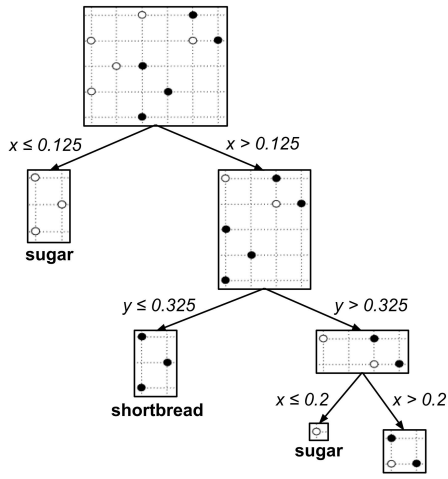
$$G_{\text{after}} = \left(\frac{2}{4}\right)\left(\frac{2}{4}\right) + \left(\frac{2}{4}\right)\left(\frac{2}{4}\right) = 0.5$$

Best Split

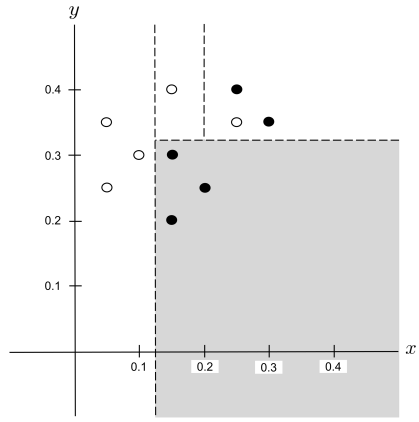
This time, there is a tie for the best split: $x_{\text{split}} = 0.2$ and $x_{\text{split}} = 0.275$ both decrease impurity the most.

Split	$G_{\text{before}} - G_{\text{after}}$	
$x_{\text{split}} = 0.2$	$0.5 - 0.333$	$= \mathbf{0.167}$
$x_{\text{split}} = 0.275$	$0.5 - 0.333$	$= \mathbf{0.167}$
$y_{\text{split}} = 0.375$	$0.5 - 0.5$	$= 0$

When ties like this occur, it does not matter which split we choose. We will arbitrarily choose the split that we encountered first, $x_{\text{split}} = 0.2$, and integrate this split into our decision tree:



This decision tree can be visualized in the plane as follows:



Worked Example: Split 4

Again, we repeat the process and split any impure leaf nodes in the tree. There is exactly one impure leaf node ($x > 0.125 \to y > 0.325 \to x > 0.2$) and it contains 2 shortbread and 1 sugar cookie, giving an impurity of

$$\begin{aligned} G_{\text{before}} &= G\left(\frac{2}{3}\right) \\ &= 2\left(\frac{2}{3}\right)\left(\frac{1}{3}\right) \\ &\approx 0.444. \end{aligned}$$

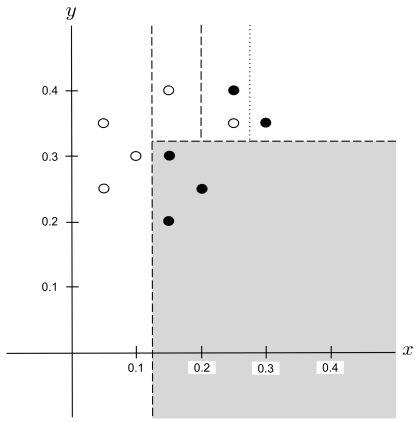
To find the possible splits, we first find the distinct values of x and y that are hit by points in this node and put them in order:

$$\begin{aligned} x &= 0.25, 0.3 \\ y &= 0.35, 0.4 \end{aligned}$$

The possible splits are the midpoints between consecutive entries in the list above:

$$\begin{aligned} x_{\text{split}} &= 0.275 \\ y_{\text{split}} &= 0.375 \end{aligned}$$

Possible Split: $x_{\text{split}} = 0.275$

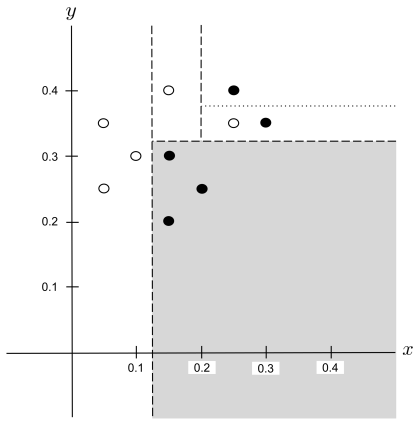


$$G_{\leq} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{2}{4}$$

$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{\text{after}} = \left(\frac{2}{3}\right)\left(\frac{2}{4}\right) + \left(\frac{1}{3}\right)(0) \approx 0.333$$

Possible Split: $y_{\text{split}} = 0.375$



$$G_{\leq} = 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = \frac{2}{4}$$

$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

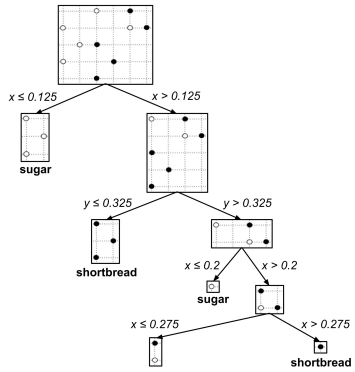
$$G_{\text{after}} = \left(\frac{2}{3}\right)\left(\frac{2}{4}\right) + \left(\frac{1}{3}\right)(0) \approx 0.333$$

Best Split

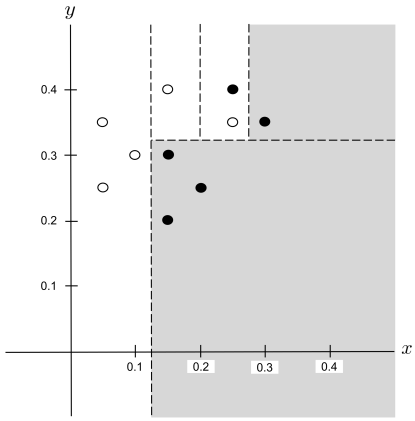
Again, there is a tie for the best split: $x_{\text{split}} = 0.275$ and $y_{\text{split}} = 0.375$ both decrease impurity the most.

Split	$G_{\text{before}} - G_{\text{after}}$
$x_{\text{split}} = 0.275$	$0.444 - 0.333 = \mathbf{0.111}$
$y_{\text{split}} = 0.375$	$0.444 - 0.333 = \mathbf{0.111}$

We will arbitrarily choose the split that we encountered first, $x_{\text{split}} = 0.275$, and integrate this split into our decision tree:



This decision tree can be visualized in the plane as follows:

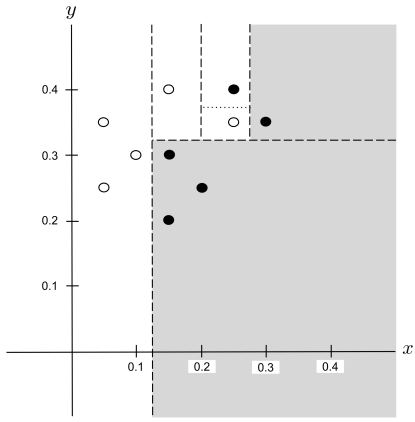


Worked Example: Split 5

There is only one possibility for the next split, $y_{\text{split}} = 0.375$, so it may be tempting to select it outright. But remember that we only want splits

that lead to a decrease in impurity. So, it's still necessary to compute the decrease in impurity before selecting this split.

$$\begin{aligned} G_{\text{before}} &= G\left(\frac{1}{2}\right) \\ &= 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right) \\ &= 0.5. \end{aligned}$$



$$G_{\leq} = 2\left(\frac{0}{1}\right)\left(\frac{1}{1}\right) = 0$$

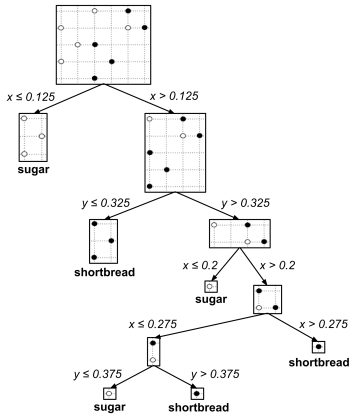
$$G_{>} = 2\left(\frac{1}{1}\right)\left(\frac{0}{1}\right) = 0$$

$$G_{\text{after}} = \left(\frac{1}{2}\right)(0) + \left(\frac{1}{2}\right)(0) = 0$$

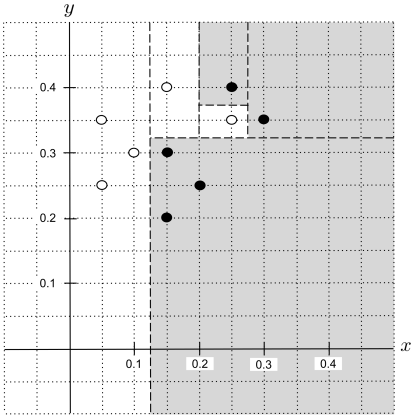
Indeed, the impurity decreases by a positive amount

$$\begin{aligned} G_{\text{before}} - G_{\text{after}} &= 0.5 - 0 \\ &= 0.5 \\ &> 0, \end{aligned}$$

so we select the split and integrate it into our decision tree:



This decision tree can be visualized in the plane as follows:



No more splits are possible, so we're done.

Early Stopping

Note that when fitting decision trees, it's common to stop splitting early so that the tree doesn't overfit the data. This is often achieved by enforcing

- a *maximum depth* constraint (i.e. skip over any potential splits that would cause the tree to become deeper than some number of levels), or
- a *minimum split size* constraint (i.e. do not split any leaf node that contains fewer than some number of data points).

These parameters constrain how far the decision tree can read into the data, similar to how the degree parameter constrains a polynomial regression model and how k constrains a k-nearest neighbors model.

Also note that if we stop splitting early (or if the data set has duplicate points with different classes), we end up with impure leaf nodes. In such cases, impure leaf nodes are considered to predict the majority class of the data points they contain. If there is a tie, then we can go up a level and use the majority class of the parent node.

Random Forests

A common way to improve the performance of decision trees is to select a bunch of random subsets of the data (each containing, say, 50% the data), fit a separate decision tree on each random subset, and then aggregate them together into a hive mind called a **random forest**. The random forest makes its predictions by

1. allowing each individual decision tree to vote (i.e. make its own prediction), and then
2. choosing whichever prediction received the most votes.

This general approach is called **bootstrap aggregating** or **bagging** for short (because a random subset of the data is known as a *bootstrap sample*). Bootstrap aggregating can be applied to any model, though random forest is the most famous application.

Exercises

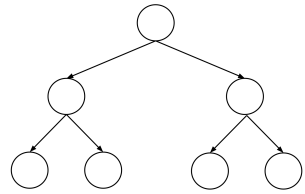
1. Implement the example that was worked out above.

2. Construct a leave-one-out cross-validation curve where a maximum depth constraint is varied between 1 and the number of points in the data set. When the maximum depth is 1, the resulting decision tree will contain only one node (a decision "stump" that simply predicts the majority class), and the leave-one-out accuracy will be 0 (since there will be fewer points in the class of the point that was left out). As usual, the leave-one-out cross-validation curve should reach a maximum somewhere between the endpoints (the endpoints correspond to underfitting or overfitting).

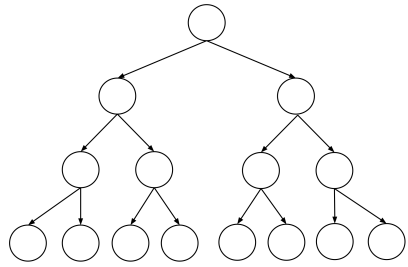
3. Construct a leave-one-out cross-validation curve where a minimum split size constraint is varied between 1 and one more than the number of points in the data set. It should look like a horizontal reflection of the curve for the maximum depth constraint because increasing the minimum split size has the same pruning effect as decreasing the maximum depth.

4. Construct a leave-one-out cross-validation curve for a random forest, where the number of trees in the forest is varied and each tree is trained on a random sample of 50% of the data. You should see the performance increase and asymptote off with the number of trees.

5. Construct a data set that leads to a decision tree that looks like the diagram shown below. Be sure to run your decision tree construction algorithm on the data set to verify the result.



6. Construct a data set that leads to a decision tree that looks like the diagram shown below. Be sure to run your decision tree construction algorithm on the data set to verify the result.



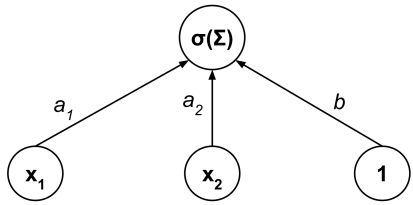
35. Introduction to Neural Network Regressors

It's common to represent models via *computational graphs*. For example, consider the following multiple logistic regression model:

$$f(x) = \frac{1}{1 + e^{-(a_1 x_1 + a_2 x_2 + b)}}$$

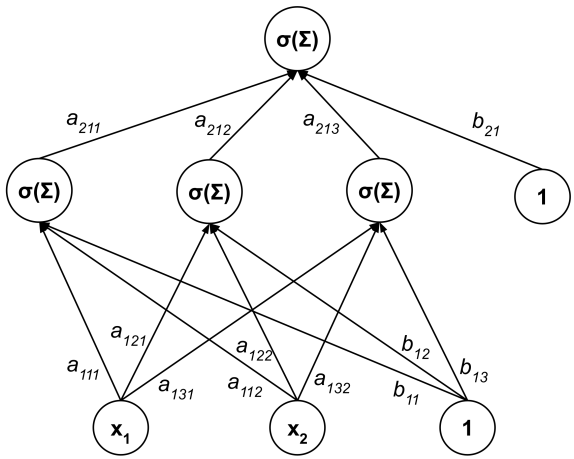
This model can be represented by the following computation graph, where

- $\Sigma = a_1 x_1 + a_2 x_2 + b$ is the sum of products of lower-node values and the edge weights, and
- $\sigma(\Sigma) = \frac{1}{1 + e^{-\Sigma}}$ is the sigmoid function.



Hierarchy and Complexity

Loosely speaking, the deeper or more "hierarchical" a computational graph is, the more complex the model that it represents. For example, consider the computational graph below, which contains an extra "layer" of nodes.



Whereas the first computational graph represented a simple model $f(x_1, x_2) = \sigma(a_1 x_1 + a_2 x_2 + b)$, this second computational graph represents a far more complex model:

$$f(x_1, x_2) = \sigma \left(\begin{array}{l} \quad a_{211}\sigma\left(a_{111}x_1 + a_{112}x_2 + b_{11}\right) \\ +a_{212}\sigma\left(a_{121}x_1 + a_{122}x_2 + b_{12}\right) \\ +a_{213}\sigma\left(a_{131}x_1 + a_{132}x_2 + b_{13}\right) \\ +b_{21} \end{array} \right)$$

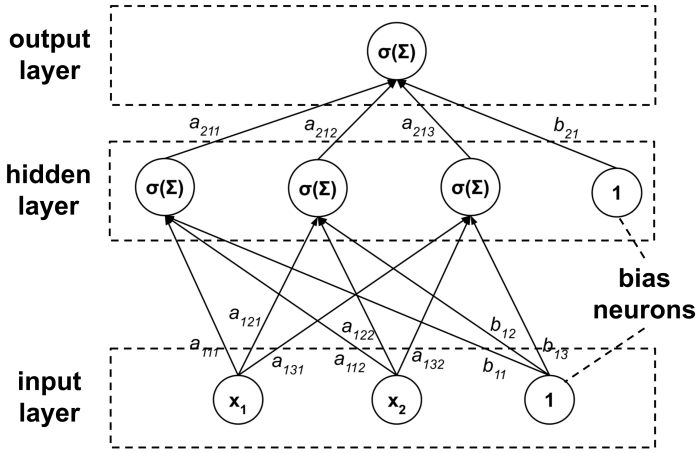
The subscripts in the coefficients may look a little crazy, but there is a consistent naming pattern:

- $a_{\ell ij}$ is the weight of the connection from the jth node in the ℓth layer to the ith node in the next layer.
- $b_{\ell i}$ is the weight of the connection from the bias node in the ℓth layer to the ith node in the next layer. (A **bias node** is a node whose output is always 1.)

Neural Networks

A **neural network** is a type of computational graph that is loosely inspired by the human brain. Each neuron in the brain receives input electrical signals from other neurons that connect to it, and the amount of signal that a neuron sends outward to the neurons it connects to depends on the total amount of electrical signal it receives as input. Each connection has a different strength, meaning that neurons

influence each other by different amounts. Additionally, neurons in key information-processing parts of the brain are sometimes arranged in layers.



Using neural network terminology, the computational graph above can be described as a neural network with 3 layers:

1. an **input layer** containing 2 linearly-activated neurons and a bias neuron,
2. a **hidden layer** containing 3 sigmoidally-activated neurons and a bias neuron, and
3. an **output layer** containing a single sigmoidally-activated neuron.

To say that a neuron is *sigmoidally-activated* means that to get the neuron's output we apply a sigmoidal **activation function** σ to the

neuron's input. Remember that the input Σ is the sum of products of lower-node values and the edge weights. By convention, a linear activation function is the identity function (i.e. the output is the same as the input).

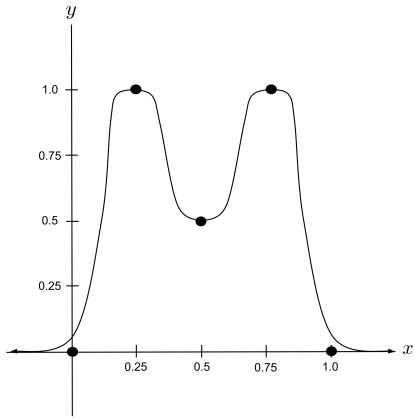
Neural networks are extremely powerful models. In fact, the *universal approximation theorem* states that given a continuous function $f : [0, 1]^n \to [0, 1]$ and an acceptable error threshold $\epsilon > 0$, there exists a sigmoidally-activated neural network with one hidden layer containing a finite number of neurons such that the error between the f and the neural network's output is less than ϵ.

Example: Manually Constructing a Neural Network

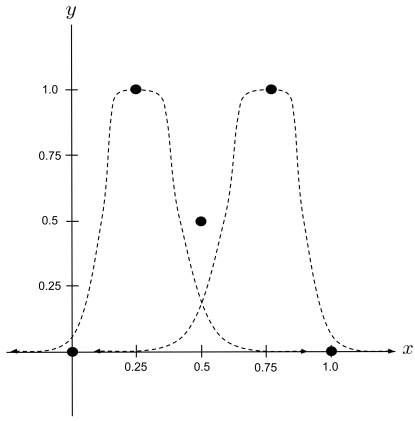
To demonstrate, let's set up a neural network that models the following data set:

$$[(0, 0), (0.25, 1), (0.5, 0.5), (0.75, 1), (1, 0)]$$

First, we'll draw a curve that approximates the data set. Then, we'll work backwards to combine sigmoid functions in a way that resembles the curve that we drew.

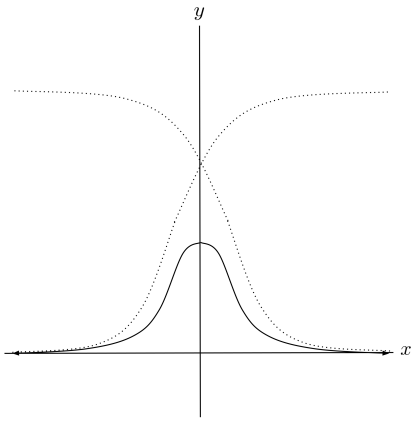


Loosely speaking, it appears that our curve can be modeled as the sum of two humps.



Notice that we can create a hump by adding two opposite-facing sigmoids (and shifting the result down to lie flat against the x-axis):

$$h(x) = \sigma(x + 1) + \sigma(-x + 1) - 1$$

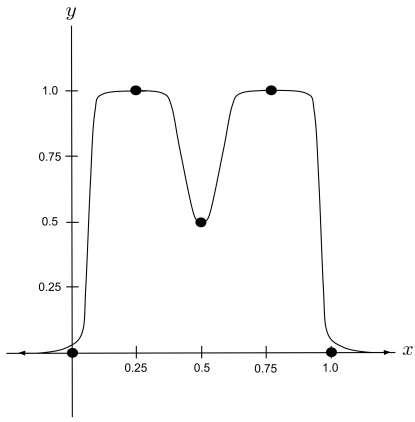


Remember that our neural network repeatedly applies sigmoid functions to sums of sigmoid functions, so we'll have to apply a sigmoid to the function above. The following composition will accomplish this while shaping our hump to be the correct width:

$$H(x) = \sigma(20h(10x) - 5)$$

Then, we can represent our final curve as the sum of two horizontally-shifted humps (again shifted downward to lie flat against the x axis and then wrapped in another sigmoid function).

$$\sigma(20H(x-0.25)+20H(x-0.75)-5)$$

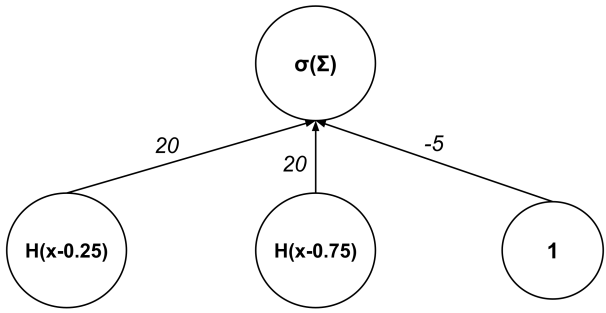


Now, let's work backwards from our final curve expression to figure out the architecture of the corresponding neural network.

Our output node represents the expression

$$\sigma(20H(x-0.25)+20H(x-0.75)-5),$$

so the previous layer should have nodes whose outputs are $H(x-0.25)$, $H(x-0.75)$, and 1 (the corresponding weights being 20, 20, and -5 respectively).

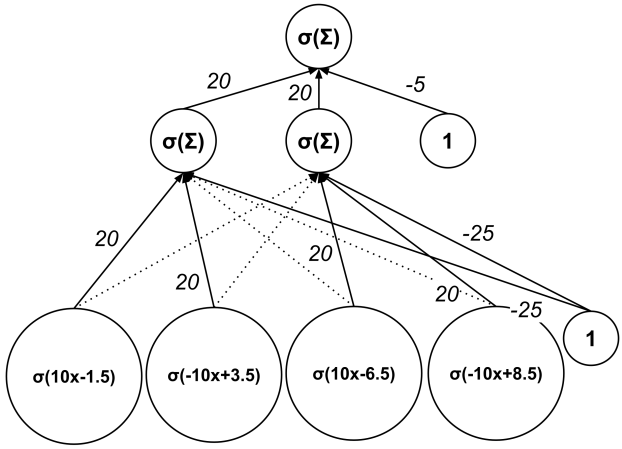


Expanding further, we have

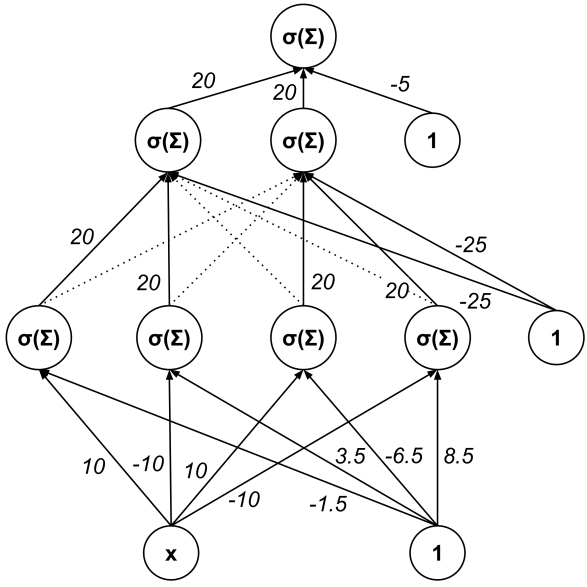
$$\begin{aligned} H(x-0.25) &= \sigma(20h(10x-2.5)-5) \\ &= \sigma(20(\sigma(10x-1.5)+\sigma(-10x+3.5)-1)-5) \\ &= \sigma(20\sigma(10x-1.5)+20\sigma(-10x+3.5)-25) \end{aligned}$$

$$\begin{aligned} H(x-0.75) &= \sigma(20h(10x-7.5)-5) \\ &= \sigma(20(\sigma(10x-6.5)+\sigma(-10x+8.5)-1)-5) \\ &= \sigma(20\sigma(10x-6.5)+20\sigma(-10x+8.5)-25), \end{aligned}$$

so the second-previous layer should have nodes whose outputs are $\sigma(10x-1.5)$, $\sigma(10x-6.5)$, $\sigma(-10x+3.5)$, $\sigma(-10x+8.5)$, and 1. (In the diagram below, edges with weight 0 are represented by soft dashed segments.)

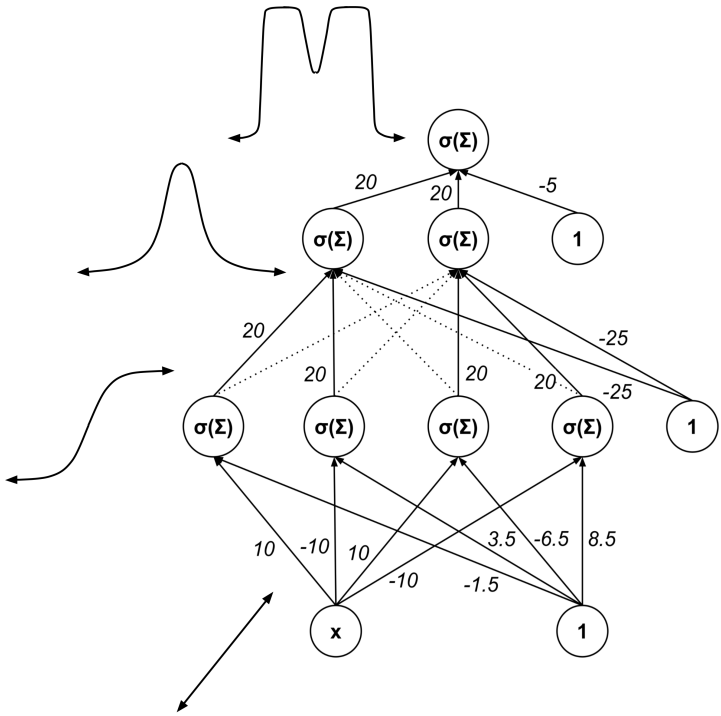


We can now sketch our full neural network as follows:



Hierarchical Representation

There is a clear hierarchical structure to the network. The first hidden layer transforms the linear intput into sigmoidal functions. The second hidden layer combines those sigmoids to generate humps. The output layer combines humps into the desired output.

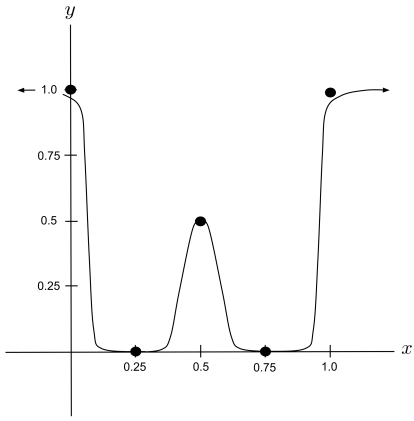


Hierarchical structure is ultimately the reason why neural networks can fit arbitrary functions to such high degrees of accuracy. Loosely speaking, each neuron in the network recognizes a different feature in the data, and deeper layers in the network synthesize elementary features into more complex features.

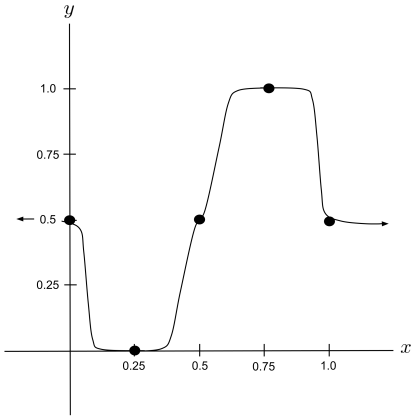
Exercises

1. Reproduce the example above by plotting the regression curve (as well as the data points).

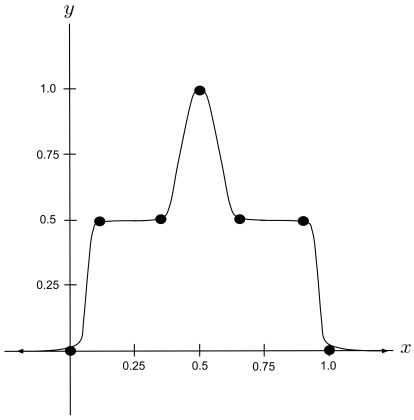
2. Tweak the neural network constructed in the discussion above so that the output resembles the following curve:



3. Tweak the neural network constructed in the discussion above so that the output resembles the following curve. (Hint: shift the equilibrium, flip one of the humps, and make the humps a little narrower.)



4. Tweak the neural network constructed in the discussion above so that the output resembles the following curve. (Hint: put a sharp peak on top of a wide plateau.)



36. Backpropagation

The most common method used to fit neural networks to data is gradient descent, just like we did have done previously for simpler models. The computations are significantly more involved for neural networks, but an algorithm called **backpropagation** provides a convenient framework for computing gradients.

Core Idea

The backpropagation algorithm leverages two key facts:

1. If you know $\dfrac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ for the output $\sigma(\Sigma)$ of a neuron, then you can easily compute $\dfrac{\partial \text{RSS}}{\partial w}$ for any weight w that the neuron receives from a neuron in the previous layer.
2. If you know $\dfrac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ for all neurons in a layer, then you can piggy-back off the result to compute $\dfrac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ for all neurons in the previous layer.

With these two facts in mind, the backpropagation algorithm consists of the following three steps:

1. *Forward propagate neuron activities.* Compute Σ and $\sigma(\Sigma)$ for all neurons in the network, starting at the input layer and repeatedly piggy-backing off the results to compute Σ and $\sigma(\Sigma)$ for all neurons in the next layer.

2. *Backpropagate neuron output gradients.* Compute $\frac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ for all neurons, starting with the output layer and then repeatedly piggy-backing off the results to compute $\frac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ in the previous layer.

3. *Expand neuron output gradients to weight gradients.* Compute $\frac{\partial \text{RSS}}{\partial w}$ for all weights in the neural network by piggy-backing off of $\frac{\partial \text{RSS}}{\partial \sigma(\Sigma)}$ for the neuron that receives the weight.

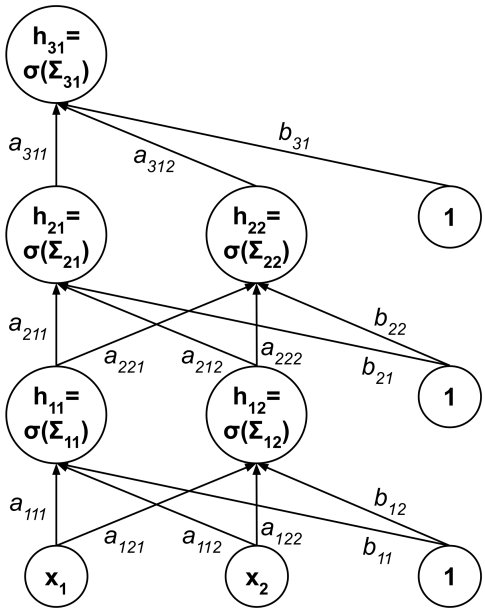
Forward Propagation of Neuron Activities

Let's formalize these steps mathematically. First, we denote the following quantities:

- $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$ are the inputs to the neural network, and $f(\vec{x})$ is the output of the neural network.

- $\vec{\Sigma}_\ell = \begin{bmatrix} \Sigma_{\ell 1} \\ \Sigma_{\ell 2} \\ \vdots \end{bmatrix}$ are the inputs to the neurons in the ℓth layer, and $\vec{h}_\ell = \begin{bmatrix} h_{\ell 1} \\ h_{\ell 2} \\ \vdots \end{bmatrix}$ are the outputs of the neurons in the ℓth layer. If the activation function of these neurons is σ, then $\vec{h}_\ell = \sigma\left(\vec{\Sigma}_\ell\right)$.
- The input layer is the 0th layer, there are L hidden layers between the input and output layers, and the output layer is the $(L+1)$th layer. Note that this means $\vec{h}_0 = \vec{x}$ and $h_{L+1} = f(\vec{x})$.
- $A_\ell = \begin{bmatrix} a_{\ell 11} & a_{\ell 12} & \cdots & \\ a_{\ell 21} & a_{\ell 22} & \cdots & \\ \vdots & & \vdots & \ddots \end{bmatrix}$ is the matrix of connection weights between the non-bias neurons in the ℓth layer and the next layer, and $\vec{b}_\ell = \begin{bmatrix} b_{\ell 1} \\ b_{\ell 2} \\ \vdots \end{bmatrix}$ are the connection weights between the bias neuron in the ℓth layer and the neurons in the next layer.

The following diagram may aid in remembering what each symbol represents.



Using the terminology introduced above, we can state the forward propagation step as follows:

$$
\begin{aligned}
\vec{\Sigma}_1 &= A_1 \vec{x} + \vec{b}_1 \\
\vec{h}_1 &= \sigma\left(\vec{\Sigma}_1\right) \\
\vec{\Sigma}_2 &= A_2 \vec{h}_1 + \vec{b}_2 \\
\vec{h}_2 &= \sigma\left(\vec{\Sigma}_2\right) \\
\vec{\Sigma}_3 &= A_3 \vec{h}_2 + \vec{b}_3 \\
\vec{h}_3 &= \sigma\left(\vec{\Sigma}_3\right) \\
&\vdots \\
\vec{\Sigma}_L &= A_L \vec{h}_{L-1} + \vec{b}_L \\
\vec{h}_L &= \sigma\left(\vec{\Sigma}_L\right) \\
\Sigma_{L+1} &= a_{(L+1)11} h_{L1} + a_{(L+1)12} h_{L2} + \cdots + b_{(L+1)1} \\
f\left(\vec{x}\right) &= \sigma\left(\Sigma_{L+1}\right)
\end{aligned}
$$

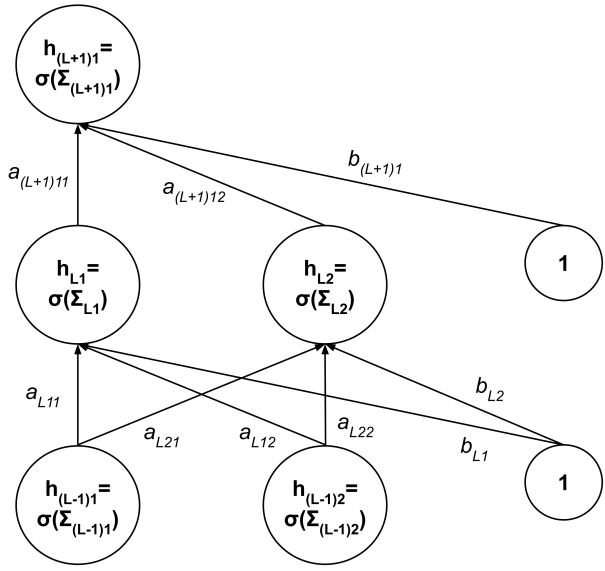
Note that the last two lines are written as scalars since the output layer contains only a single neuron, i.e. *the* output neuron.

Backpropagation of Neuron Output Gradients

Now, let's formalize the backpropagation step for a point $(\vec{x}, y)$. First, we compute the gradient with respect to the output neuron. Remember that the output of the output neuron is $f(\vec{x})$, which can also be denoted as $h_{(L+1)1}$ since the output layer is the $(L+1)$th layer.

$$\begin{aligned}
\frac{\partial \text{RSS}}{\partial h_{(L+1)i}} &= \frac{\partial}{\partial h_{(L+1)i}} \left[\left(f\left(\vec{x}\right) - y \right)^2 \right] \\
&= \frac{\partial}{\partial h_{(L+1)i}} \left[\left(h_{(L+1)i} - y \right)^2 \right] \\
&= 2 \left(h_{(L+1)i} - y \right)
\end{aligned}$$

Then, we backpropagate to the previous layer.



$$\begin{aligned}
\frac{\partial \text{RSS}}{\partial h_{Li}} &= \frac{\partial \text{RSS}}{\partial h_{(L+1)1}} \cdot \frac{\partial h_{(L+1)1}}{\partial h_{Li}} \\
&= \frac{\partial \text{RSS}}{\partial h_{(L+1)1}} \cdot \frac{\partial}{\partial h_{Li}} \left[\sigma\left(\Sigma_{(L+1)1}\right)\right] \\
&= \frac{\partial \text{RSS}}{\partial h_{(L+1)1}} \sigma'\left(\Sigma_{(L+1)1}\right) \cdot \frac{\partial}{\partial h_{Li}} \left[\Sigma_{(L+1)1}\right] \\
&= \frac{\partial \text{RSS}}{\partial h_{(L+1)1}} \sigma'\left(\Sigma_{(L+1)1}\right) \cdot \frac{\partial}{\partial h_{Li}} \begin{bmatrix} a_{(L+1)11} h_{L1} + a_{(L+1)12} h_{L2} \\ + \cdots + b_{(L+1)1} \end{bmatrix} \\
&= \frac{\partial \text{RSS}}{\partial h_{(L+1)1}} \sigma'\left(\Sigma_{(L+1)1}\right) a_{(L+1)1i}
\end{aligned}$$

Note that the quantity $\frac{\partial \text{RSS}}{\partial h_{(L+1)1}}$ was already computed, so we do not have to expand it out.

We continue backpropagating using the same approach. Note that hidden layers contain multiple nodes (unlike the output layer), so we need a term for each node.

$$\begin{aligned}\frac{\partial \text{RSS}}{\partial h_{(L-1)i}} &= \frac{\partial \text{RSS}}{\partial h_{L1}} \cdot \frac{\partial h_{L1}}{\partial h_{(L-1)i}} + \frac{\partial \text{RSS}}{\partial h_{L2}} \cdot \frac{\partial h_{L2}}{\partial h_{(L-1)i}} + \cdots \\ &= \cdots \\ &= \frac{\partial \text{RSS}}{\partial h_{L1}} \sigma'\left(\Sigma_{L1}\right) a_{L1i} + \frac{\partial \text{RSS}}{\partial h_{L2}} \sigma'\left(\Sigma_{L2}\right) a_{L2i} + \cdots\end{aligned}$$

Again, note that the quantities $\frac{\partial \text{RSS}}{\partial h_{L1}}$, $\frac{\partial \text{RSS}}{\partial h_{L2}}$, $\ldots$ were already computed, so we do not have to expand them out.

Also note that we can consolidate into vector form:

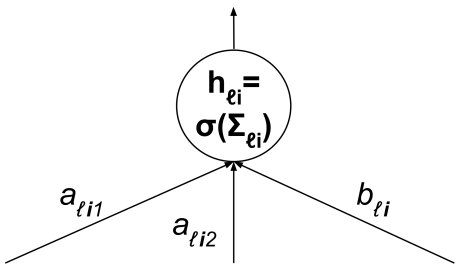
$$
\begin{aligned}
\frac{\partial \text{RSS}}{\partial \vec{h}_{L-1}} &= \begin{bmatrix} \frac{\partial \text{RSS}}{\partial h_{(L-1)1}} \\ \frac{\partial \text{RSS}}{\partial h_{(L-1)2}} \\ \vdots \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial \text{RSS}}{\partial h_{L1}} \sigma'\left(\Sigma_{L1}\right) a_{L11} + \frac{\partial \text{RSS}}{\partial h_{L2}} \sigma'\left(\Sigma_{L2}\right) a_{L21} + \cdots \\ \frac{\partial \text{RSS}}{\partial h_{L1}} \sigma'\left(\Sigma_{L1}\right) a_{L12} + \frac{\partial \text{RSS}}{\partial h_{L2}} \sigma'\left(\Sigma_{L2}\right) a_{L22} + \cdots \\ \vdots \end{bmatrix} \\
&= \begin{bmatrix} a_{L11} & a_{L21} & \cdots \\ a_{L12} & a_{L22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \frac{\partial \text{RSS}}{\partial h_{L1}} \sigma'\left(\Sigma_{L1}\right) \\ \frac{\partial \text{RSS}}{\partial h_{L2}} \sigma'\left(\Sigma_{L2}\right) \\ \vdots \end{bmatrix} \\
&= A_L^T \left(\frac{\partial \text{RSS}}{\partial \vec{h}_L} \circ \sigma'\left(\vec{\Sigma}_L\right) \right),
\end{aligned}
$$

where $\circ$ denotes the element-wise product.

We keep backpropagating using the same approach until we reach the input layer. At that point, we will have computed $\frac{\partial \text{RSS}}{\partial h_{\ell i}}$ for every neuron in the network.

Expansion of Neuron Output Gradients to Weight Gradients

Finally, we expand the neuron output gradients into weight gradients, i.e. coefficient gradients $\frac{\partial \text{RSS}}{\partial a_{\ell ij}}$ and bias gradients $\frac{\partial \text{RSS}}{\partial b_{\ell i}}$.



$$\begin{aligned}
\frac{\partial \text{RSS}}{\partial a_{\ell ij}} &= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \cdot \frac{\partial h_{\ell i}}{\partial a_{\ell ij}} \\
&= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \cdot \frac{\partial}{\partial a_{\ell ij}} \left[\sigma \left(\Sigma_{\ell i} \right) \right] \\
&= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \sigma' \left(\Sigma_{\ell i} \right) \cdot \frac{\partial}{\partial a_{\ell ij}} \left[\Sigma_{\ell i} \right] \\
&= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \sigma' \left(\Sigma_{\ell i} \right) \cdot \frac{\partial}{\partial a_{\ell ij}} \left[\begin{array}{c} a_{\ell i1} h_{(\ell-1)1} + a_{\ell i2} h_{(\ell-1)2} \\ + \cdots + b_{(\ell-1)i} \end{array} \right] \\
&= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \sigma' \left(\Sigma_{\ell i} \right) h_{(\ell-1)j}
\end{aligned}$$

By the same computation, we get

$$\frac{\partial \text{RSS}}{\partial b_{\ell i}} = \frac{\partial \text{RSS}}{\partial h_{\ell i}} \sigma' \left(\Sigma_{\ell i} \right).$$

Notice that the expression for $\frac{\partial \text{RSS}}{\partial b_{\ell i}}$ appears in the expression for $\frac{\partial \text{RSS}}{\partial a_{\ell ij}}$, so we can simplify:

$$\begin{aligned} \frac{\partial \text{RSS}}{\partial b_{\ell i}} &= \frac{\partial \text{RSS}}{\partial h_{\ell i}} \sigma' \left(\Sigma_{\ell i} \right) \\ \frac{\partial \text{RSS}}{\partial a_{\ell ij}} &= \frac{\partial \text{RSS}}{\partial b_{\ell i}} h_{(\ell-1)j} \end{aligned}$$

Again, we can consolidate into vector form:

$$\begin{aligned} \frac{\partial \text{RSS}}{\partial \vec{b}_\ell} &= \frac{\partial \text{RSS}}{\partial \vec{h}_\ell} \circ \sigma' \left(\vec{\Sigma}_\ell \right) \\ \frac{\partial \text{RSS}}{\partial A_\ell} &= \frac{\partial \text{RSS}}{\partial \vec{b}_\ell} \otimes \vec{h}_{\ell-1}, \end{aligned}$$

where $\otimes$ is the outer product.

Gradient Descent Update

Once we know all the weight gradients, we can update the weights using the usual gradient descent update:

$$A_\ell \to A_\ell - \alpha \frac{\partial \text{RSS}}{\partial A_\ell} \quad \text{where} \quad \frac{\partial \text{RSS}}{\partial A_\ell} = \sum_{(\vec{x},y)} \frac{\partial \text{RSS}}{\partial A_\ell}\bigg|_{(\vec{x},y)}$$

$$\vec{b}_\ell \to \vec{b}_\ell - \alpha \frac{\partial \text{RSS}}{\partial \vec{b}_\ell} \quad \text{where} \quad \frac{\partial \text{RSS}}{\partial \vec{b}_\ell} = \sum_{(\vec{x},y)} \frac{\partial \text{RSS}}{\partial \vec{b}_\ell}\bigg|_{(\vec{x},y)}$$

Pseudocode

The following pseudocode summarizes the backpropagation algorithm that was derived above.

1. Reset all gradient placeholders

$\forall \ell \in \{1, 2, ..., L\}$:

$$\frac{\partial \text{RSS}}{\partial A_\ell} = \begin{bmatrix} 0 & 0 & \cdots \\ 0 & 0 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

$$\frac{\partial \text{RSS}}{\partial \vec{b}_\ell} = \vec{0}$$

2. Loop over all data points

$\forall(\vec{x}, y)$:

2.1 Forward propagate neuron activities

$$\vec{\Sigma}_0 = \vec{x}$$

$$\vec{h}_0 = \vec{x}$$

$\forall \ell \in \{0, 1, \ldots, L\}$:

$$\vec{\Sigma}_{\ell+1} = A_{\ell+1}\vec{h}_\ell + \vec{b}_{\ell+1}$$

$$\vec{h}_{\ell+1} = \sigma\left(\vec{\Sigma}_{\ell+1}\right)$$

2.2 Backpropagate neuron output gradients

$$\frac{\partial \text{RSS}}{\partial h_{(L+1)1}} = 2\left(h_{(\ell+1)1} - y\right)$$

$\forall \ell \in \{L, L-1, \ldots, 1\}$:

$$\frac{\partial \text{RSS}}{\partial \vec{h}_\ell} = A_{\ell+1}^T \left(\frac{\partial \text{RSS}}{\partial \vec{h}_{\ell+1}} \circ \sigma'\left(\vec{\Sigma}_{\ell+1}\right)\right)$$

2.3 Expand to weight gradients

$$\forall \ell \in \{L+1, L, \ldots, 1\}:$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_\ell}\right|_{(\vec{x},y)} = \frac{\partial \text{RSS}}{\partial \vec{h}_\ell} \circ \sigma'\left(\vec{\Sigma}_\ell\right)$$

$$\left.\frac{\partial \text{RSS}}{\partial A_\ell}\right|_{(\vec{x},y)} = \left.\frac{\partial \text{RSS}}{\partial \vec{b}_\ell}\right|_{(\vec{x},y)} \otimes \vec{h}_{\ell-1}$$

$$\frac{\partial \text{RSS}}{\partial A_\ell} \to \frac{\partial \text{RSS}}{\partial A_\ell} + \left.\frac{\partial \text{RSS}}{\partial A_\ell}\right|_{(\vec{x},y)}$$

$$\frac{\partial \text{RSS}}{\partial \vec{b}_\ell} \to \frac{\partial \text{RSS}}{\partial \vec{b}_\ell} + \left.\frac{\partial \text{RSS}}{\partial \vec{b}_\ell}\right|_{(\vec{x},y)}$$

3. Update weights via gradient descent

$$\forall \ell \in \{1, 2, ..., L\}:$$

$$A_\ell \to A_\ell - \alpha \frac{\partial \text{RSS}}{\partial A_\ell}$$

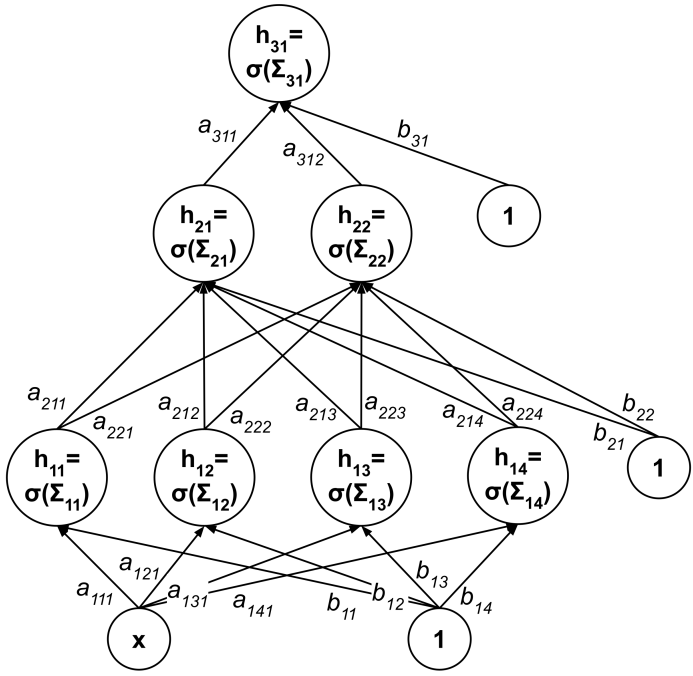
$$\vec{b}_\ell \to \vec{b}_\ell - \alpha \frac{\partial \text{RSS}}{\partial \vec{b}_\ell}$$

You might notice that steps 2.2 and 2.3 above can be combined more efficiently into a single step since $\frac{\partial \text{RSS}}{\partial \vec{h}_\ell} = A_{\ell+1}^T \frac{\partial \text{RSS}}{\partial \vec{b}_{\ell+1}}$. However, we will keep these steps separate for the sake of intuitive clarity. You are welcome to combine these steps in your own implementation.

Worked Example of a Single Iteration

Now, let's walk through an concrete example of fitting a neural network to a data set using the backpropagation algorithm. We will use the same data set and neural network architecture as the previous chapter:

$$[(0, 0), (0.25, 1), (0.5, 0.5), (0.75, 1), (1, 0)]$$



Because neural networks are hierarchical and high-dimensional (i.e. they have many parameters that are tightly coupled), they are vastly more difficult to train as compared to simpler non-hierarchical low-dimensional models like linear, logistic, and polynomial regressions.

Various tricks are often required to prevent the neural network from getting "stuck" in suboptimal local minima, which we will not cover here.

To provide a simple example that illustrates the training of a neural network to a high degree of accuracy while avoiding the need for more advanced tricks, we will intentionally choose the initial weights of our network to be similar to the weights that we arrived at when manually constructing a neural network in the previous chapter. (More specifically, they will be proportional by a factor of 0.5.) This will place us near a deep valley on the surface of RSS as a function of parameters of the neural network, and the proximity will allow elementary gradient descent to lead us down into the valley.

So, we will use the following initial weights:

$$A_1 = \begin{bmatrix} 5 \\ -5 \\ 5 \\ -5 \end{bmatrix} \qquad \vec{b}_1 = \begin{bmatrix} -0.75 \\ 1.75 \\ -3.25 \\ 4.25 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 10 & 10 & 0 & 0 \\ 0 & 0 & 10 & 10 \end{bmatrix} \qquad \vec{b}_2 = \begin{bmatrix} -12.5 \\ -12.5 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 10 & 10 \end{bmatrix} \qquad \vec{b}_3 = \begin{bmatrix} -2.5 \end{bmatrix}$$

Let's work out the first iteration of backpropagation by hand, using learning rate $\alpha = 0.01$. Note that the values shown are rounded to 6

decimal places, but intermediate values are not actually rounded in the implementation.

Point: $(\vec{x}, y) = ([0], 0)$

Forward propagation

$$\vec{\Sigma}_0 = \vec{x} = \begin{bmatrix} 0 \end{bmatrix}$$
$$\vec{h}_0 = \vec{x} = \begin{bmatrix} 0 \end{bmatrix}$$

$$\vec{\Sigma}_1 = A_1\vec{h}_0 + \vec{b}_1 = \begin{bmatrix} 5 \\ -5 \\ 5 \\ -5 \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix} + \begin{bmatrix} -0.75 \\ 1.75 \\ -3.25 \\ 4.25 \end{bmatrix} = \begin{bmatrix} -0.75 \\ 1.75 \\ -3.25 \\ 4.25 \end{bmatrix}$$

$$\vec{h}_1 = \sigma\left(\vec{\Sigma}_1\right) = \sigma\left(\begin{bmatrix} -0.75 \\ 1.75 \\ -3.25 \\ 4.25 \end{bmatrix}\right) = \begin{bmatrix} 0.320821 \\ 0.851953 \\ 0.037327 \\ 0.985936 \end{bmatrix}$$

$$\vec{\Sigma}_2 = A_2\vec{h}_1 + \vec{b}_2 = \begin{bmatrix} 10 & 10 & 0 & 0 \\ 0 & 0 & 10 & 10 \end{bmatrix} \begin{bmatrix} 0.320821 \\ 0.851953 \\ 0.037327 \\ 0.985936 \end{bmatrix} + \begin{bmatrix} -12.5 \\ -12.5 \end{bmatrix} = \begin{bmatrix} -0.772259 \\ -2.267367 \end{bmatrix}$$

$$\vec{h}_2 = \sigma\left(\vec{\Sigma}_2\right) = \sigma\left(\begin{bmatrix} -0.772259 \\ -2.267367 \end{bmatrix}\right) = \begin{bmatrix} 0.315991 \\ 0.093862 \end{bmatrix}$$

$$\vec{\Sigma}_3 = A_3\vec{h}_2 + \vec{b}_3 = \begin{bmatrix} 10 & 10 \end{bmatrix} \begin{bmatrix} 0.315991 \\ 0.093862 \end{bmatrix} + \begin{bmatrix} -2.5 \end{bmatrix} = \begin{bmatrix} 1.59852529 \end{bmatrix}$$
$$\vec{h}_3 = \sigma\left(\vec{\Sigma}_3\right) = \sigma\left(\begin{bmatrix} 1.598525 \end{bmatrix}\right) = \begin{bmatrix} 0.831812 \end{bmatrix}$$

Backpropagation

$$\frac{\partial \text{RSS}}{\partial h_{31}} = 2\left(h_{31} - y\right) = 2\left(0.831812 - 0\right) = 1.663624$$

$$\frac{\partial \text{RSS}}{\partial \vec{h}_2} = A_3^T\left(\frac{\partial \text{RSS}}{\partial \vec{h}_3} \circ \sigma'\left(\vec{\Sigma}_3\right)\right) = \begin{bmatrix}10\\10\end{bmatrix}\left(\begin{bmatrix}1.663624\end{bmatrix} \circ \sigma'\left(\begin{bmatrix}1.59852529\end{bmatrix}\right)\right) = \begin{bmatrix}2.327422\\2.327422\end{bmatrix}$$

$$\frac{\partial \text{RSS}}{\partial \vec{h}_1} = A_2^T\left(\frac{\partial \text{RSS}}{\partial \vec{h}_2} \circ \sigma'\left(\vec{\Sigma}_2\right)\right) = \begin{bmatrix}10 & 0\\10 & 0\\0 & 10\\0 & 10\end{bmatrix}\left(\begin{bmatrix}2.327422\\2.327422\end{bmatrix} \circ \sigma'\left(\begin{bmatrix}-0.772259\\-2.267367\end{bmatrix}\right)\right) = \begin{bmatrix}5.030502\\5.030502\\1.979515\\1.979515\end{bmatrix}$$

Expansion

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([0],0)} = \frac{\partial \text{RSS}}{\partial \vec{h}_3} \circ \sigma'\left(\vec{\Sigma}_3\right) = \begin{bmatrix}1.663624\end{bmatrix} \circ \sigma'\left(\begin{bmatrix}1.598525\end{bmatrix}\right) = \begin{bmatrix}0.232742\end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial A_3}\right|_{([0],0)} = \left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([0],0)} \otimes \vec{h}_2 = \begin{bmatrix}-0.000913\end{bmatrix} \otimes \begin{bmatrix}0.315991\\0.093862\end{bmatrix} = \begin{bmatrix}0.073544 & 0.021846\end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([0],0)} = \frac{\partial \text{RSS}}{\partial \vec{h}_2} \circ \sigma'\left(\vec{\Sigma}_2\right) = \begin{bmatrix}2.327422\\2.327422\end{bmatrix} \circ \sigma'\left(\begin{bmatrix}-0.772259\\-2.267367\end{bmatrix}\right) = \begin{bmatrix}0.503050\\0.197951\end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial A_2}\right|_{([0],0)} = \left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([0],0)} \otimes \vec{h}_1 = \begin{bmatrix}0.503050\\0.197951\end{bmatrix} \otimes \begin{bmatrix}0.320821\\0.851953\\0.037327\\0.985936\end{bmatrix} = \begin{bmatrix}0.161389 & 0.428575 & 0.018777 & 0.495976\\0.063507 & 0.168645 & 0.007389 & 0.195168\end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([0],0)} = \frac{\partial \text{RSS}}{\partial \vec{h}_1} \circ \sigma'\left(\vec{\Sigma}_1\right) = \begin{bmatrix}5.030502\\5.030502\\1.979515\\1.979515\end{bmatrix} \circ \sigma'\left(\begin{bmatrix}-0.75\\1.75\\-3.25\\4.25\end{bmatrix}\right) = \begin{bmatrix}1.096121\\0.634493\\0.071131\\0.027448\end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial A_1}\right|_{([0],0)} = \left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([0],0)} \otimes \vec{h}_0 = \begin{bmatrix}1.096121\\0.634493\\0.071131\\0.027448\end{bmatrix} \otimes \begin{bmatrix}0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$$

Point: $(\vec{x}, y) = ([0.25], 1)$

$$\vec{\Sigma}_0 = \begin{bmatrix} 0.25 \end{bmatrix} \qquad \vec{h}_0 = \begin{bmatrix} 0.25 \end{bmatrix}$$

$$\vec{\Sigma}_1 = \begin{bmatrix} 0.5 \\ 0.5 \\ -2 \\ 3 \end{bmatrix} \qquad \vec{h}_1 = \begin{bmatrix} 0.622459 \\ 0.622459 \\ 0.119203 \\ 0.952574 \end{bmatrix}$$

$$\vec{\Sigma}_2 = \begin{bmatrix} -0.050813 \\ -1.782230 \end{bmatrix} \qquad \vec{h}_2 = \begin{bmatrix} 0.487299 \\ 0.144028 \end{bmatrix}$$

$$\vec{\Sigma}_3 = \begin{bmatrix} 3.813274 \end{bmatrix} \qquad \vec{h}_3 = \begin{bmatrix} 0.978401 \end{bmatrix}$$

$$\frac{\partial \text{RSS}}{\partial \vec{h}_3} = \begin{bmatrix} -0.043198 \end{bmatrix}, \quad \frac{\partial \text{RSS}}{\partial \vec{h}_2} = \begin{bmatrix} -0.009129 \\ -0.009129 \end{bmatrix}, \quad \frac{\partial \text{RSS}}{\partial \vec{h}_1} = \begin{bmatrix} -0.022807 \\ -0.022807 \\ -0.011254 \\ -0.011254 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([0.25],1)} = \begin{bmatrix} -0.000913 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_3}\right|_{([0.25],1)} = \begin{bmatrix} -0.000445 & -0.000131 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([0.25],1)} = \begin{bmatrix} -0.002281 \\ -0.001125 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_2}\right|_{([0.25],1)} = \begin{bmatrix} -0.001420 & -0.001420 & -0.000272 & -0.002173 \\ -0.000701 & -0.000701 & -0.000134 & -0.001072 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([0.25],1)} = \begin{bmatrix} -0.005360 \\ -0.005360 \\ -0.001182 \\ -0.000508 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_1}\right|_{([0.25],1)} = \begin{bmatrix} -0.001340 \\ -0.001340 \\ -0.000295 \\ -0.000127 \end{bmatrix}$$

Point: $(\vec{x}, y) = ([0.5], 0.5)$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.020099 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_3}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.006351 & 0.006351 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.043443 \\ 0.043443 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_2}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.037011 & 0.013937 & 0.013937 & 0.037011 \\ 0.037011 & 0.013937 & 0.013937 & 0.037011 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.054794 \\ 0.094659 \\ 0.094659 \\ 0.054794 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_1}\right|_{([0.5],0.5)} = \begin{bmatrix} 0.027397 \\ 0.047330 \\ 0.047330 \\ 0.027397 \end{bmatrix}$$

Point: $(\vec{x}, y) = ([0.75], 1)$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([0.75],1)} = \begin{bmatrix} -0.000913 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_3}\right|_{([0.75],1)} = \begin{bmatrix} -0.000131 & -0.000445 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([0.75],1)} = \begin{bmatrix} -0.001125 \\ -0.002281 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_2}\right|_{([0.75],1)} = \begin{bmatrix} -0.001072 & -0.000134 & -0.000701 & -0.000701 \\ -0.002173 & -0.000272 & -0.001420 & -0.001420 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([0.75],1)} = \begin{bmatrix} -0.000508 \\ -0.001182 \\ -0.005360 \\ -0.005360 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_1}\right|_{([0.75],1)} = \begin{bmatrix} -0.000381 \\ -0.000886 \\ -0.004020 \\ -0.004020 \end{bmatrix}$$

Point: $(\vec{x}, y) = ([1], 0)$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_3}\right|_{([1],0)} = \begin{bmatrix} 0.232742 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_3}\right|_{([1],0)} = \begin{bmatrix} 0.021846 & 0.073544 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_2}\right|_{([1],0)} = \begin{bmatrix} 0.197951 \\ 0.503050 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_2}\right|_{([1],0)} = \begin{bmatrix} 0.195168 & 0.007389 & 0.168645 & 0.063507 \\ 0.495976 & 0.018777 & 0.428575 & 0.161389 \end{bmatrix}$$

$$\left.\frac{\partial \text{RSS}}{\partial \vec{b}_1}\right|_{([1],0)} = \begin{bmatrix} 0.027448 \\ 0.071131 \\ 0.634493 \\ 1.096121 \end{bmatrix} \qquad \left.\frac{\partial \text{RSS}}{\partial A_1}\right|_{([1],0)} = \begin{bmatrix} 0.027448 \\ 0.071131 \\ 0.634493 \\ 1.096121 \end{bmatrix}$$

Weight Updates

Summing up all the gradients we computed, we get the following:

$$\frac{\partial \text{RSS}}{\partial \vec{b}_3} = \begin{bmatrix} 0.483758 \end{bmatrix} \quad \frac{\partial \text{RSS}}{\partial A_3} = \begin{bmatrix} 0.101165 & 0.101165 \end{bmatrix}$$

$$\frac{\partial \text{RSS}}{\partial \vec{b}_2} = \begin{bmatrix} 0.741038 \\ 0.741038 \end{bmatrix} \quad \frac{\partial \text{RSS}}{\partial A_2} = \begin{bmatrix} 0.391076 & 0.448347 & 0.200388 & 0.593621 \\ 0.593621 & 0.200388 & 0.448347 & 0.391076 \end{bmatrix}$$

$$\frac{\partial \text{RSS}}{\partial \vec{b}_1} = \begin{bmatrix} 1.172495 \\ 0.793742 \\ 0.793742 \\ 1.172495 \end{bmatrix} \quad \frac{\partial \text{RSS}}{\partial A_1} = \begin{bmatrix} 0.053123 \\ 0.116235 \\ 0.677508 \\ 1.119371 \end{bmatrix}$$

Finally, applying the gradient descent updates $A_\ell \to A_\ell - \alpha \dfrac{\text{RSS}}{\partial A_\ell}$ and $\vec{b}_\ell \to \vec{b}_\ell - \alpha \dfrac{\text{RSS}}{\partial \vec{b}_\ell}$ with $\alpha = 0.01$, we get the following updated weights:

$$A_1 = \begin{bmatrix} 4.999469 \\ -5.001162 \\ 4.993225 \\ -5.011194 \end{bmatrix} \qquad \vec{b}_1 = \begin{bmatrix} -0.761725 \\ 1.742063 \\ -3.257937 \\ 4.238275 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 9.996089 & 9.995517 & -0.002004 & -0.005936 \\ -0.005936 & -0.002004 & 9.995517 & 9.996089 \end{bmatrix} \quad \vec{b}_2 = \begin{bmatrix} -12.507410 \\ -12.507410 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 9.998988 & 9.998988 \end{bmatrix} \qquad \vec{b}_3 = \begin{bmatrix} -2.504838 \end{bmatrix}$$

Demonstration of Many Iterations

Repeating this procedure over and over, we get the following results. Note that the values shown are rounded to 3 decimal places, but intermediate values are not actually rounded in the implementation.

Initial

- RSS ≈ 1.614
- Predictions $\approx 0.832, 0.978, 0.979, 0.978, 0.832$
  (Compare to $0, \quad 1, \quad 0.5, \quad 1, \quad 0$)

$$A_1 = \begin{bmatrix} 5 \\ -5 \\ 5 \\ -5 \end{bmatrix} \qquad \vec{b}_1 = \begin{bmatrix} -0.75 \\ 1.75 \\ -3.25 \\ 4.25 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 10 & 10 & 0 & 0 \\ 0 & 0 & 10 & 10 \end{bmatrix} \qquad \vec{b}_2 = \begin{bmatrix} -12.5 \\ -12.5 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 10 & 10 \end{bmatrix} \qquad \vec{b}_3 = \begin{bmatrix} -2.5 \end{bmatrix}$$

After 1 *iteration*

- RSS ≈ 1.525
- Predictions $\approx 0.812, 0.973, 0.973, 0.972, 0.801$

$$A_1 = \begin{bmatrix} 4.999 \\ -5.001 \\ 4.993 \\ -5.011 \end{bmatrix} \qquad \vec{b}_1 = \begin{bmatrix} -0.762 \\ 1.742 \\ -3.258 \\ 4.238 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 9.996 & 9.996 & -0.002 & -0.006 \\ -0.006 & -0.002 & 9.996 & 9.996 \end{bmatrix} \quad \vec{b}_2 = \begin{bmatrix} -12.507 \\ -12.507 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 9.999 & 9.999 \end{bmatrix} \qquad \vec{b}_3 = \begin{bmatrix} -2.505 \end{bmatrix}$$

After 2 *iterations*

- RSS ≈ 1.426
- Predictions $\approx 0.789, 0.967, 0.965, 0.962, 0.765$

$$A_1 \approx \begin{bmatrix} 4.999 \\ -5.002 \\ 4.986 \\ -5.023 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.774 \\ 1.734 \\ -3.267 \\ 4.226 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.992 & 9.991 & -0.004 & -0.012 \\ -0.012 & -0.004 & 9.991 & 9.992 \end{bmatrix} \qquad \vec{b}_2 \approx \begin{bmatrix} -12.515 \\ -12.515 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 9.998 & 9.998 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -2.510 \end{bmatrix}$$

After 3 *iterations*

- RSS ≈ 1.320
- Predictions $\approx 0.764, 0.959, 0.954, 0.949, 0.725$

$$A_1 \approx \begin{bmatrix} 4.998 \\ -5.004 \\ 4.978 \\ -5.035 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.787 \\ 1.725 \\ -3.276 \\ 4.213 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.987 & 9.986 & -0.006 & -0.019 \\ -0.019 & -0.006 & 9.986 & 9.988 \end{bmatrix} \qquad \vec{b}_2 \approx \begin{bmatrix} -12.524 \\ -12.524 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 9.997 & 9.997 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -2.516 \end{bmatrix}$$

After 10 *iterations*

- RSS ≈ 0.730
- Predictions $\approx 0.571, 0.844, 0.816, 0.774, 0.478$

$$A_1 \approx \begin{bmatrix} 4.992 \\ -5.015 \\ 4.933 \\ -5.101 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.873 \\ 1.66 \\ -3.331 \\ 4.142 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.957 & 9.953 & -0.022 & -0.064 \\ -0.058 & -0.022 & 9.958 & 9.959 \end{bmatrix} \quad \vec{b}_2 \approx \begin{bmatrix} -12.58 \\ -12.575 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 9.99 & 9.991 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -2.557 \end{bmatrix}$$

After 100 *iterations*

- RSS ≈ 0.496
- Predictions $\approx 0.362, 0.694, 0.730, 0.698, 0.356$

$$A_1 \approx \begin{bmatrix} 5.068 \\ -4.962 \\ 4.965 \\ -5.158 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.936 \\ 1.696 \\ -3.249 \\ 4.164 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.920 & 9.862 & -0.064 & -0.15 \\ -0.116 & -0.074 & 9.894 & 9.921 \end{bmatrix} \quad \vec{b}_2 \approx \begin{bmatrix} -12.708 \\ -12.683 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 9.978 & 9.981 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -2.735 \end{bmatrix}$$

After 1 000 *iterations*

- RSS ≈ 0.198
- Predictions $\approx 0.199, 0.788, 0.668, 0.788, 0.202$

$$A_1 \approx \begin{bmatrix} 5.491 \\ -5.101 \\ 5.343 \\ -5.152 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.518 \\ 2.008 \\ -3.081 \\ 4.546 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.744 & 9.666 & -0.301 & -0.425 \\ -0.442 & -0.301 & 9.668 & 9.721 \end{bmatrix} \quad \vec{b}_2 \approx \begin{bmatrix} -13.155 \\ -13.170 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 9.982 & 9.98 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -3.596 \end{bmatrix}$$

After 10 000 *iterations*

- RSS ≈ 0.0239
- Predictions $\approx 0.068, 0.922, 0.517, 0.915, 0.075$

$$A_1 \approx \begin{bmatrix} 6.96 \\ -6.033 \\ 6.632 \\ -5.806 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.279 \\ 2.766 \\ -3.307 \\ 5.478 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 9.88 & 9.555 & -0.835 & -0.865 \\ -0.932 & -0.806 & 9.647 & 9.781 \end{bmatrix} \quad \vec{b}_2 \approx \begin{bmatrix} -13.763 \\ -13.804 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 10.515 & 10.542 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -4.759 \end{bmatrix}$$

After 100 000 *iterations*

- RSS ≈ 0.0020
- Predictions $\approx 0.020, 0.979, 0.501, 0.976, 0.023$

$$A_1 \approx \begin{bmatrix} 8.407 \\ -7.103 \\ 7.786 \\ -6.971 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.523 \\ 3.289 \\ -3.906 \\ 6.412 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 10.437 & 9.708 & -1.367 & -1.063 \\ -1.179 & -1.311 & 9.922 & 10.416 \end{bmatrix} \qquad \vec{b}_2 \approx \begin{bmatrix} -14.075 \\ -14.139 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 11.607 & 11.800 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -5.433 \end{bmatrix}$$

After $1\,000\,000$ *iterations*

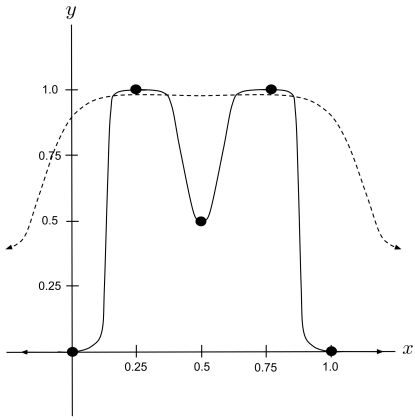
- RSS ≈ 0.0002
- Predictions $\approx 0.006, 0.993, 0.500, 0.993, 0.007$

$$A_1 \approx \begin{bmatrix} 9.527 \\ -7.895 \\ 8.602 \\ -7.956 \end{bmatrix} \qquad \vec{b}_1 \approx \begin{bmatrix} -0.738 \\ 3.651 \\ -4.384 \\ 7.219 \end{bmatrix}$$

$$A_2 \approx \begin{bmatrix} 11.006 & 9.855 & -1.801 & -1.214 \\ -1.405 & -1.730 & 10.129 & 11.100 \end{bmatrix} \qquad \vec{b}_2 \approx \begin{bmatrix} -14.323 \\ -14.440 \end{bmatrix}$$

$$A_3 \approx \begin{bmatrix} 12.902 & 13.223 \end{bmatrix} \qquad \vec{b}_3 \approx \begin{bmatrix} -6.178 \end{bmatrix}$$

Below is a graph of the regression curves before and after training (i.e. using the initial weights and using the weights after 1 000 000 iterations of backpropagation). The trained network does an even better job of passing through the leftmost and rightmost points than the network we constructed manually in the previous section!



Exercises

1. Implement the example that was worked out above.

2. Re-run the example that was worked out above, this time using initial weights drawn randomly from the normal distribution. Note that your RSS should gradually decrease but it may get "stuck" in a suboptimal local minimum, resulting in a regression curve that is a decent but not perfect fit.

Part VI

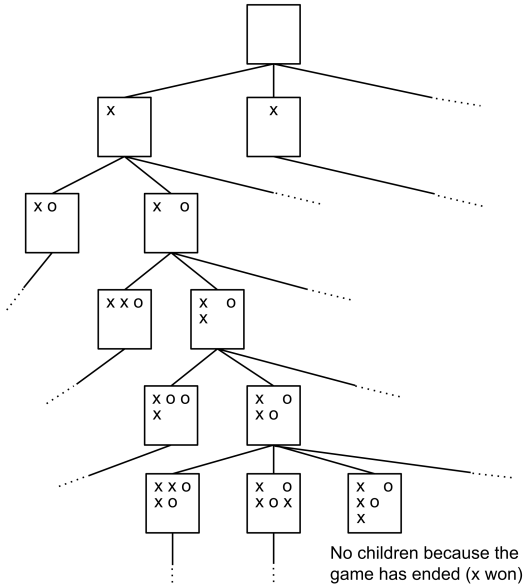
Games

37. Canonical and Reduced Game Trees for Tic-Tac-Toe

A **game tree** is a data structure that represents all the possible outcomes of a game. It is a graph where the nodes correspond to the states of the game, and the edges correspond to actions that cause the game to transition from one state to another. Game trees are commonly used when coding up strategies for autonomous game-playing agents.

Exercise: Tic-Tac-Toe Tree

Create a class `TicTacToeTree` that constructs a game tree for tic-tac-toe. Each node in the game tree has corresponds to a state of the game. The root node represents an empty board. It has 9 children, one for each move that the first player can make. Each of those 9 children have 8 children (after the first player has moved, there are 8 moves remaining for the second player). And so on.



There are 255 168 unique ways that a game of tic-tac-toe can play out, so you can check your tree by verifying that there are 255 168 *leaf nodes*.

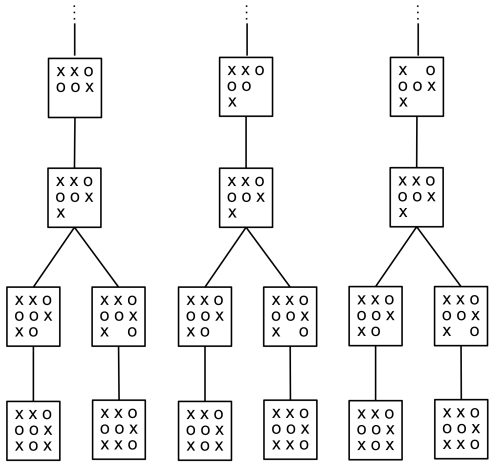
Here are some tips regarding the implementation:

- Each node should have a state attribute that holds the state of the tic-tac-toe game, a player attribute that says whose turn it is, and a winner attribute that says if someone has won.
- Instead of passing edges into the tree at initialization, you'll need to build up your tree algorithmically: start with a tree with a single node, and then repeatedly create child nodes until they reach a terminal state (i.e. a state where the game is finished).
- Ultimately this just comes down to a graph traversal (breadth-first or depth-first, doesn't matter which). Whenever a node's

game state is not terminal, create a child node for each possible next state.

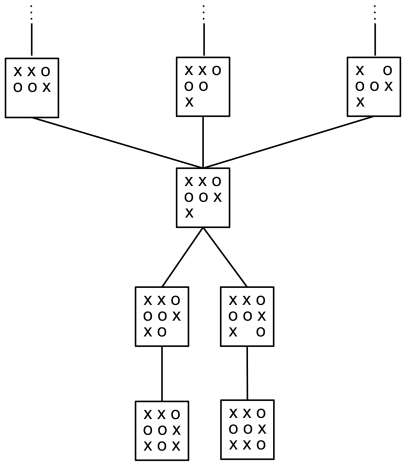
Exercise: Reduced Tic-Tac-Toe Tree

Once you've built your `TicTacToeTree` and verified that it has the correct number of leaf nodes, the next step is to make it more efficient. Notice that there are many redundancies where separate nodes represent the same state:



Although redundancies are included in the canonical conception of a game tree, we can greatly speed up the construction and reduce the size of our game tree if we use only one node per game state. To do this,

you'll need to make a slight tweak to your traversal so that whenever a node with the desired state already exists, you connect up that existing node as a child (instead of creating a new node).



Do not loop over the tree every time to check if a node with the desired child state already exists. That would be really inefficient! Instead, store nodes in a dictionary where the key represents the game state. That way, to check if a node with a particular game state already exists, you just need to look up that state in the dictionary.

There are $5\,478$ distinct possible game states in the game of tic-tac-toe, so you can check your reduced tree by verifying that there are $5\,478$ nodes *in total*.

38. Minimax Strategy

The **minimax strategy** is a powerful game-playing strategy that operates on game trees. It envisions the worst-case scenario that could possibly result from any given move, and then chooses the move that would result in the best (i.e. "least bad") worst-case scenario.

Minimax Algorithm

The minimax strategy chooses actions according to the following algorithm:

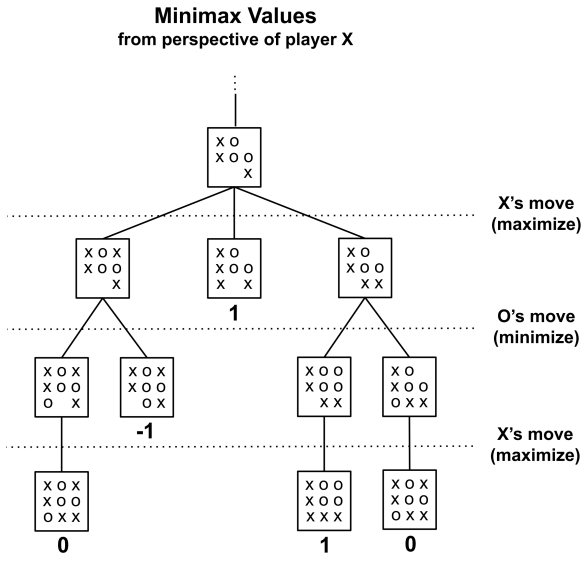
1. Create a game tree with all the states of the game.
2. Identify each node that represents a terminal state and assign it a minimax value of 1, -1, or 0 depending on whether it corresponds to a win, loss, or tie for you.
3. Repeatedly propagate those values up the tree to parent nodes, assuming that you will try to win (i.e. move into states that maximize your value) and your opponent will try to make you lose (i.e. move into states that minimize your value).

  - If the edge from the parent node corresponds to your turn, then the parent node's minimax value is the maximum of the child values (because you want to maximize your value).
  - Otherwise, if the edge from the parent node corresponds to your *opponent's* turn, then the parent node's minimax value is the *minimum* of the child values (because your opponent wants to *minimize* your value).

- Always choose the move that takes you to the next possible state with the highest minimax value. (You can break ties via random choice.)

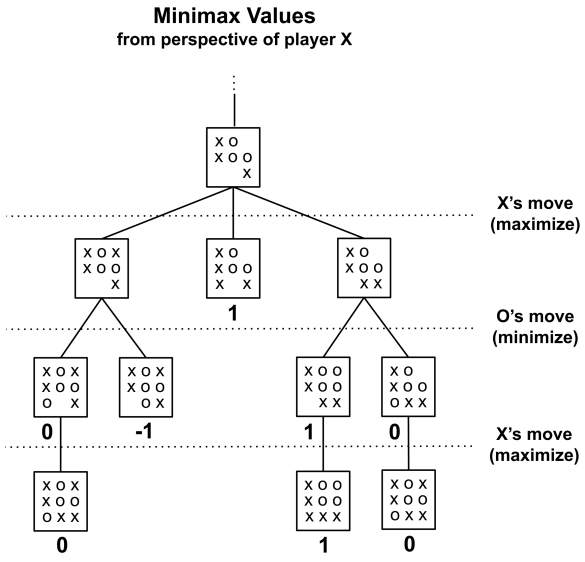
Worked Example

To illustrate the minimax algorithm in action, let's label part of a tic-tac-toe game tree with minimax values, from the perspective of player X (i.e. supposing we are player X). We always start by labeling the terminal states.

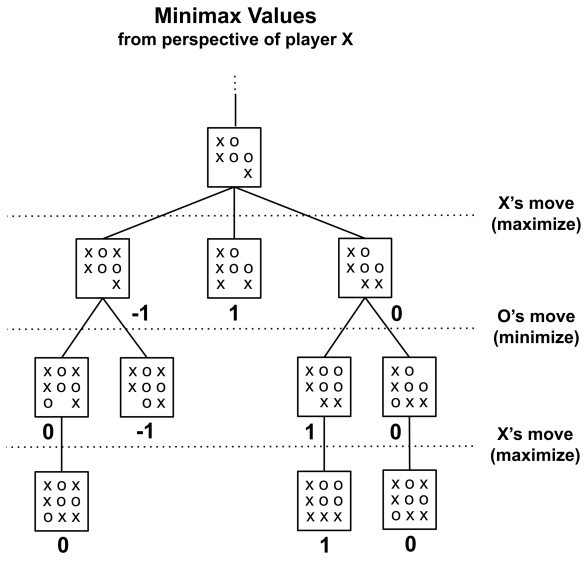


Then, we propagate these values up to parent nodes. But we can only compute the minimax value of a node once we've assigned minimax values to all its children.

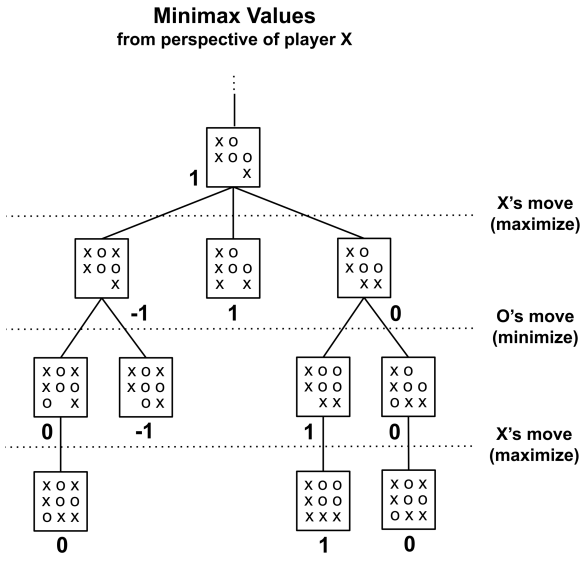
Here, there are 3 parents who do not have minimax values but whose children all do. The edges between these parents and their children all correspond to moves by X, which is us, and we want to maximize the minimax value. So, to each of these parents, we assign the maximum value of its children. (In this case, each of these parents has only one child, so the maximum value happens to be the only value.)



Now, we repeat the process. Now, there are 2 parents who do not have minimax values but whose children all do. The edges between these parents and their children all correspond to moves by O, which is our opponent, and our opponent wants to minimize the minimax value. So, to each of these parents, we assign the minimum value of its children.



Again, we repeat the process. There is a single parent (the top node) who does not have a minimax value but whose children all do. The edges between these parents and their children all correspond to moves by X, which is us, and we want to maximize the minimax value. So we assign it the maximum value of its children.



We've assigned minimax values to all the nodes in this part of the tree. The minimax value of the highest node is 1, which tells us that there is a guaranteed way to win from that game state (all we need to do is place an X in the bottom-left corner). Indeed, this action is accomplished by choosing the child node with the maximum value.

Exercises

1. Implement a **minimax player** for your tic-tac-toe game that automatically chooses actions based on the minimax strategy. (It goes without saying: don't rebuild and relabel the game tree on every move. That would be very inefficient and slow. Build

and label it once at the beginning, and then use the same tree throughout the rest of the game.)

2. Run your minimax player against a deterministic "top-left strategy" that always moves into the leftmost open space in the topmost row. At each of the minimax player's turns, print out the possible moves that the minimax player could possibly make as well as the associated minimax values of the states. Check the following:

   – Every minimax value should be either 1, -1, or 0.
   – Each of the minimax player's chosen moves should be associated with a maximum-value state.
   – Towards the end of the game, you should be able to inspect game states, manually sketch out the section of the game tree containing their progeny, and then manually compute and verify the minimax value of each state.
   – Make sure these checks still hold when the minimax player goes second.

3. Then, run your minimax player against a random player for many games (alternating who goes first). The minimax player should *never* lose. If you encounter any game where the minimax player loses, then you'll need to store the sequence of moves and step through the game to debug what went wrong.

4. Play two minimax players against each other for many games, alternating who goes first. Every game should result in a tie.

5. Play against the minimax player yourself. You shouldn't be able to win.

39. Reduced Search Depth and Heuristic Evaluation for Connect Four

In theory, we could solve any game by building a big game tree, labeling the terminal states as wins, losses, or ties, and then working backwards from that information to identify the minimax strategy. But in practice, game trees get so big so quickly that for all but the most simple games, game trees are too expensive to store in memory and take too long to traverse.

For example, consider the game of connect four, which is normally played on a 7×6 board. Whereas tic-tac-toe had $5\,478$, valid board states, connect four has $4\,531\,985\,219\,092$. Implementing a game tree of this size is infeasible – if you're not convinced, try running a simple "for" loop that loops over the numbers from 1 to $4\,531\,985\,219\,092$. If looping over a million numbers takes a few seconds, then looping over a trillion numbers will take weeks.

Reduced Search Depth

We can't build a full game tree for connect four. But what we can do instead is

1. **reduce the search depth** (i.e. build a shortened game tree up to some maximum depth), and
2. come up with some kind of **heuristic evaluation function** to rate how good or bad each leaf node state is.

Then we can apply the minimax strategy in an attempt to move in the direction of the best leaf node state.

This procedure for selecting a move can be outlined more explicitly as follows:

1. Build a game tree that is N layers deep from the current game state. (This is called an N-**ply** game tree.)
2. Use the heuristic evaluation function to assign minimax to the terminal nodes in the game tree.
3. Repeatedly propagate those values up to parent nodes using the minimax algorithm.
4. Choose your action in accordance with the standard minimax strategy (i.e. choose the move that takes you to the child state with the highest minimax value).

Note that this time, you'll have to relabel the game tree on every move because the terminal nodes of the tree will change (thereby changing the minimax values of the rest of the tree). But you don't need to rebuild the full game tree on every move – you can take the existing game tree, prune off nodes that are no longer relevant, and grow the additional nodes needed to bring you back to a search depth of N.

Heuristic Evaluation Function

Now, let's talk about the "secret sauce" in this recipe: the heuristic evaluation function. It takes a game state as input, and returns a number between -1 and 1 that represents how strongly we want or do not want to be in that game state. To write this function we use our human intuition about the game. Here are some rough guidelines:

- If we're 100% confident that a game state is a win or will result in a win, then the function should return 1.
- If we think that a win is more likely than a loss, then the function should return a decimal between 0 and 1, with higher win probabilities corresponding to higher decimals.
- If we have no idea whether a game state will result in a win or loss, or we think it will result in a tie, then the function should return 0.
- If we think that a win is less likely than a loss, then the function should return a decimal between 0 and -1, with lower win probabilities corresponding to more negative decimals.

- If we're 100% confident that a game state is a loss or will result in a loss, then the function should return -1.

For example, here is a simple heuristic function for tic-tac-toe:

1. If the game state is a definite win, tie, or loss, then return 1, 0, or -1 respectively.
2. Otherwise, count up the number of rows, columns, and diagonals where you occupy two spaces and the third space is empty. Then, subtract the number of rows, columns, and diagonals where your opponent occupies two spaces and the third space is empty. Finally, divide the result by 8 (which is the total number of rows, columns, and diagonals).

Exercises

Remember to alternate who goes first in the matchups described below.

1. Implement a 9-ply heuristic minimax tic-tac-toe player, i.e. it uses the heuristic evaluation function described above and a search depth of $N = 9$ (which happens to be the full tree). Then, run it against the perfect minimax player that you created previously. Every game should result in a tie.
2. Implement a 2-ply heuristic minimax tic-tac-toe player. Then, run it against your 9-ply heuristic player, as well as a purely random player. The 2-ply player should do better than the random player but worse than the 9-ply player.

3. Develop a heuristic minimax connect four player that uses as many ply as can be computed quickly, and verify that it performs better than a random player.

4. Verify that your heuristic minimax connect four player performs better than a "last-minute" player that moves randomly unless it has an opportunity to capture a win or block a loss on the next move. (If it doesn't, then you may need to improve your heuristic.)

5. Play against your heuristic minimax connect four player yourself.

40. Introduction to Blondie24 and Neuroevolution

Previously, we built strategic connect four players by constructing a pruned game tree, using heuristics to rate terminal states, and then applying the minimax algorithm. This was a combination of game-specific human intelligence (heuristics) and generalizable artificial intelligence (minimax on a game tree).

Blondie24

In the 1990s, a researcher named David Fogel managed to automate the process of rating states in pruned game trees without relying on heuristics or any other human input. In particular, he and his colleague Kumar Chellapilla created a computer program that achieved expert level checkers-playing ability by learning from scratch. They played it against other humans online under the username Blondie24, pretending to be a 24-year old blonde female college student.

Blondie24 was particularly noteworthy because other successful game-playing agents had been hand-tuned and/or trained on human-expert strategies. Unlike these agents, Blondie24 learned *without* having any access to information on human-expert strategies.

To automate the process of rating states in pruned game trees, Fogel turned it into a regression problem: given a game state, predict its value. Of course, the regression function is pretty complicated (e.g. changing one piece on a chess board can totally change the outcome of the game), so the natural choice was to use a neural network.

However, the usual method of training a neural network, backpropagation, does not work in this setting. Backpropagation relies on a dataset of pairs of inputs and outputs – which means that the model would need a data set of game states along with their correct rating, totally defeating the purpose of getting the model to learn this information from scratch. In this setting, the only feedback the computer gets is at the very end of the game, whether it won or lost (or tied).

Neuroevolution

To get around this issue, Fogel trained neural networks via **evolution**, which is often referred to as **neuroevolution** in the context of neural networks. Starting with a population of many neural networks with random weights, he repeatedly

1. played the networks against each other,

2. discarded the networks that performed worse than average,
3. duplicated the remaining networks, and then
4. randomly perturbed the weights of the duplicate networks.

This is analogous to the concept of evolution in biology in which weak organisms die and fit organisms survive to produce similar but slightly mutated offspring. By repeatedly running the evolutionary procedure, Fogel was able to evolve a neural network whose internal mapping from input state to output rating caused it to play the game of checkers in an intelligent way, without any sort of human input.

Exercise: Evolving a Neural Network Regressor

Before we reimplement Fogel's papers leading up to Blondie24, let's first gain some experience with neuroevolution in a simpler case. As a toy problem, consider the following data set:

```
[
  (0.0, 1.0), (0.04, 0.81), (0.08, 0.52), (0.12, 0.2),
      (0.17, -0.12),
  (0.21, -0.38), (0.25, -0.54), (0.29, -0.58), (0.33,
      -0.51), (0.38, -0.34),
  (0.42, -0.1), (0.46, 0.16), (0.5, 0.39), (0.54, 0.55),
       (0.58, 0.61),
  (0.62, 0.55), (0.67, 0.38), (0.71, 0.12), (0.75,
      -0.19), (0.79, -0.51),
  (0.83, -0.77), (0.88, -0.95), (0.92, -1.0), (0.96,
      -0.91), (1.0, -0.7)
]
```

We will fit the above data using a neural network regressor with the following architecture:

- *Input Layer:* 1 linearly-activated node and 1 bias node
- *First Hidden Layer:* 10 tanh-activated nodes and 1 bias node
- *Second Hidden Layer:* 6 tanh-activated nodes and 1 bias node
- *Third Hidden Layer:* 3 tanh-activated nodes and 1 bias node
- *Output Layer:* 1 tanh-activated node

Remember that hyperbolic tangent function is defined as

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

To train the neural network, use the following evolutionary algorithm (which is based on the Blondie24 approach):

1. Create a population of 15 neural networks with weights randomly drawn from $[-0.2, 0.2]$. Additionally, assign a mutation rate to each net, initially equal to $\alpha = 0.05$.

2. Each of the 15 parents replicates to produce a single child. The child is given mutation rate

$$\alpha^{\text{child}} = \alpha^{\text{parent}} e^{N(0,1)/\sqrt{2\sqrt{|W|}}}$$

   and weights

$$w_{ij}^{\text{child}} = w_{ij}^{\text{parent}} + \alpha^{\text{child}} N(0,1),$$

   where $N(0,1)$ is a random number drawn from the standard normal distribution and $|W|$ is the number of weights in the network. Be sure to draw a different random number for each instance of $N(0,1)$.

3. Compute the RSS for each net and select the 15 nets with the lowest RSS. These will be the parents in the next generation.

4. Go back to step 2.

Make a plot of the average RSS at each generation, run the algorithm until the graph levels off to nearly 0, and then plot the regression curves corresponding to the first and last generations of neural networks on a graph along with the data.

(The regression curve plot will contain 60 different curves drawn on the same plot: one curve from each of the 30 nets in the first generation, and one curve from each of the 30 nets in the last generation.)

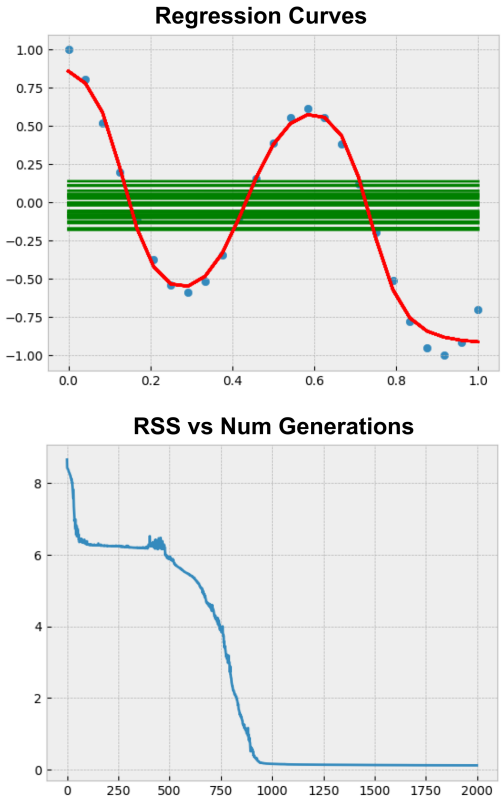
The first generation curves will not fit the data at all (they will appear flat), but the final generation of regression curves should fit the data remarkably well. Note that the training process may require on the order of a thousand generations.

Exercise: Hyperparameter Tuning

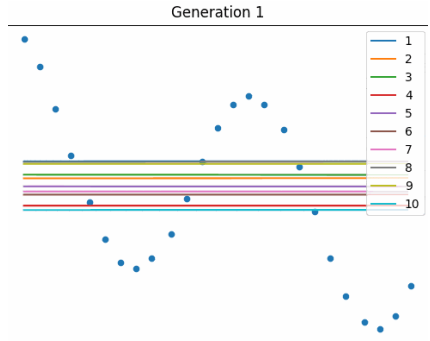
Once you've got this working, try tuning hyperparameters to get the RSS to converge to nearly 0 as quickly as possible. You can tweak the mutation rate, initial weight distribution, number of neural networks, and neural network architecture (i.e. number of hidden layers and their sizes).

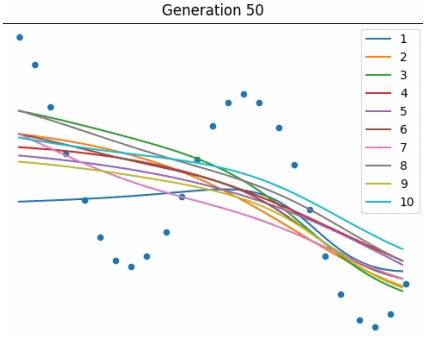
Visualization

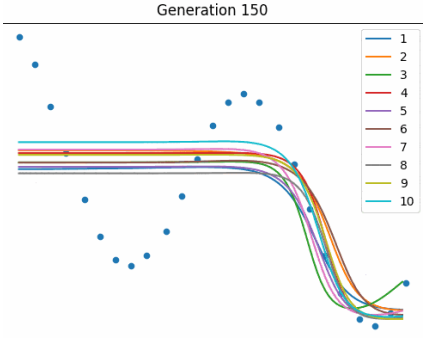
Courtesy of Maia Dimas, below is an illustration of the first and last generations of neural nets, along with a graph of RSS versus the number of generations.

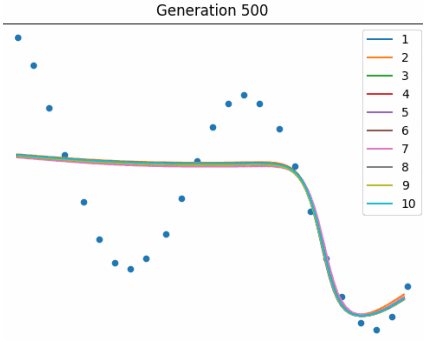


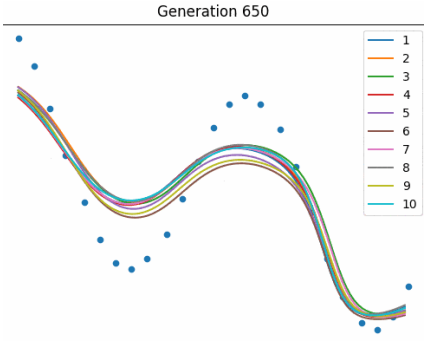
Courtesy of Elias Gee, below some intermediate illustrations of a small population (10 nets) as they learn to fit the curve over many generations.

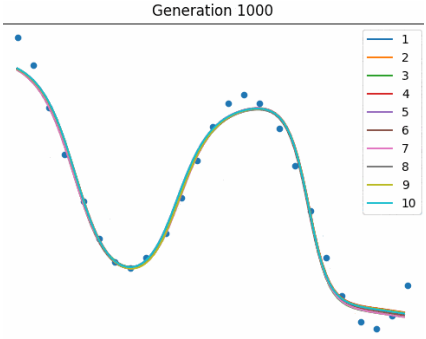



Generation 50
1
2
3
4
5
6
7
8
9
10


Generation 150
1
2
3
4
5
6
7
8
9
10


Generation 500
1
2
3
4
5
6
7
8
9
10


Generation 650
1
2
3
4
5
6
7
8
9
10


Generation 1000
1
2
3
4
5
6
7
8
9
10

41. Reimplementing Fogel's Tic-Tac-Toe Paper

The goal of this section is to reimplement the paper *Using Evolutionary Programming to Create Neural Networks that are Capable of Playing Tic-Tac-Toe* by David Fogel in 1993. This paper preceded Blondie24, and many of the principles introduced in this paper were extended in Blondie24. As such, reimplementing the paper provides good scaffolding as we work our way up to reimplement Blondie24.

The information needed to reimplement this paper is outlined below.

Neural Network Architecture

The neural net consists of the following layers:

- *Input Layer:* 9 linearly-activated nodes and 1 bias node
- *Hidden Layer:* H sigmoidally-activated nodes and 1 bias node (H is variable, as will be described later)
- *Output Layer:* 9 sigmoidally-activated nodes

Converting Board to Input

A tic-tac-toe board is converted to input by flattening it into a vector and replacing X with 1, empty squares with 0, and O with -1.

For example, given a board

$$\begin{bmatrix} \text{X} & \text{O} & \square \\ \square & \text{X} & \text{O} \\ \square & \square & \square \end{bmatrix},$$

we first concatenate consecutive rows to flatten the board into the following 9-element vector:

$$\langle \text{X}, \text{O}, \square, \square, \text{X}, \text{O}, \square, \square, \square \rangle ,$$

Then, we replace X with 1, empty squares with 0, and O with -1 to get the final input vector:

$$\langle 1, -1, 0, 0, 1, -1, 0, 0, 0 \rangle$$

Converting Output to Action

The output layer consists of 9 nodes, one for each board square. To convert the output values into an action, we do the following:

1. Discard any values that correspond to a board square that has already been filled. (This will prevent illegal moves.)
2. Identify the empty board square with the maximum value. We move into this square.

Evolution Procedure

The initial population consists of 50 networks. In each network, the number of hidden nodes H is randomly chosen from the range $\{1, 2, \ldots, 10\}$ and the initial weights are randomly chosen from the range $[-0.5, 0.5]$.

Replication

A network replicates by making a copy of itself and then modifying the copy as follows:

- Each weight is incremented by $N(0, 0.05^2)$, a value drawn from the normal distribution with mean 0 and standard deviation 0.05.
- With 0.5 probability, we modify the network architecture. If we modify the architecture, then we do so by randomly choosing between adding or deleting a hidden node. If we add a node, then we initialize its associated weights with values of 0.

Note that when modifying the architecture, we abort any decision that would lead the number of hidden nodes to exit the range $\{1, 2, \ldots, 10\}$. More specifically:

- We abort the decision to delete a hidden node if the number of hidden nodes is $H = 1$.
- We abort the decision to add a hidden node if the number of hidden nodes is $H = 10$.

Evaluation

In each generation, each network plays 32 games against a near-perfect but beatable opponent. The evolving network is always allowed to move first, and receives a payoff of 1 for a win, 0 for a tie, and -10 for a loss.

The near-perfect opponent follows the strategy below:

```
With 10% chance:
 - Randomly choose an open square to move into.

Otherwise:
 - If the next move can be chosen to win the game, do so
     .

 - Otherwise, if the next move can be chosen to block
   the opponent's win, do so.

 - Otherwise, if there are 2 open squares in a line with
   the opponent's marker, randomly move into one of
       those
   squares.

 - Otherwise, randomly choose an open square to move
     into.
```

Once the total payoff (over 32 games against the near-perfect opponent) has been computed for each of the 100 networks (the 50 parents and their 50 children), a second round of evaluation occurs to select the networks that will proceed to the next generation and replicate.

In the second round of evaluation, each network is given a score that represents how its total payoff (from matchups with the near-perfect opponent) compares to the total payoffs of some other networks in

the same generation. Specifically, each network is compared to 10 other networks randomly chosen from the generation, and its score is incremented for each other network that has a lower total payoff.

The top 50 networks from the second round of evaluation are selected to proceed to the next generation and replicate.

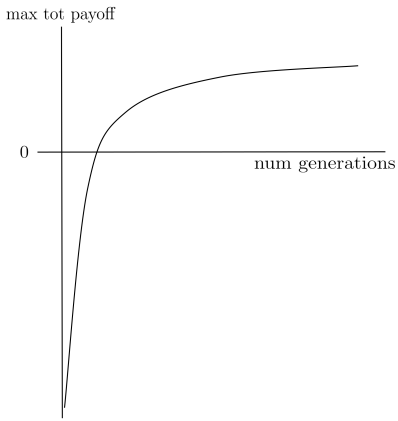
Note: The actual paper states that the networks with the highest performance from the second round of evaluation are selected. Interpreted formally, it would suggest to only select those network(s) that had the absolute maximum performance. But this would lead the generations to rapidly shrink in size, resulting in premature convergence. The informal interpretation (top 50 networks) leads the generation sizes to stay the same, which avoids the issue.

Performance Curve

Generate a performance curve as follows:

1. Run the above procedure for 800 generations, keeping track of the maximum total payoff (i.e. the best player's total payoff) at each generation.
2. Then repeat 19 more times, for a total of 20 trials of 800 generations each.
3. Finally, plot the mean maximum total payoff (averaged over the 20 trials) as a function of the number of generations.

The resulting curve should resemble the following shape:



Once you're able to generate a good performance curve, store the neural network parameters for the best player from the last generation so that you can play it against a random player and verify its intelligence. (It should beat the random player the vast majority of the time.)

Debugging: Sanity Checks

Before you attempt to run the full 20 trials of 800 generations with 50 parent networks in each generation, run a quicker sanity check (say, 5 trials of 50 generations with 10 parent networks in each generation) to make sure that the curve looks like it's moving in the correct direction (upward).

Given the complexity of this project, your plot will likely not come out correct the first time you try to run it, even if you feel certain that you have correctly implemented the specifications of the paper.

If and when this happens, you will need to add plenty of validation to your implementation. A non-exhaustive list of validation checks and unit tests are provided below for the first half of the paper specifications. You will need to come up validation checks for the second half on your own.

1. Each generation, check that each individual neural network has a valid number of nodes in each layer.

2. Whenever you convert a board to input for a neural network, check that the resulting entries are all in the set $\{-1, 0, 1\}$ add up to 0 before the neural network player makes its next move.

3. Create a unit test takes a variety of possible board states and output vectors from the neural network and then verifies that the player makes the appropriate move.

4. Check that the initial weights are between -0.5 and 0.5, no weight stays the same after replication, some weights increase and some weights decrease after replication (i.e. they don't all move in the same direction), and no weight changes by more than 0.3 (which is 6 standard deviations).

5. Check that after replication, some networks have an additional hidden layer node, other networks have one less hidden layer node, and other networks have an unchanged number of hidden layer nodes.

6. ...

If the plot still doesn't look right even after your sanity checks are implemented and passing, then create a log of the results using the template shown below.

(Sometimes there can be issues that you don't anticipate in your sanity checks, that become apparent when you take a birds-eye view and manually look at concrete numbers.)

```
HYPERPARAMETERS
   Networks per generation: 20
   Selection percentage: 0.5

ABBREVIATIONS
   star (*) if selected
   H = number of non-bias neurons in hidden layer
   [min weight, mean weight, max weight]
   (wins : losses : ties)
   (IDs won against | IDs lost against | IDs tied
       against)

GENERATION 1

  NN 1 (parent, H=7, [-0.42, 0.02, 0.37])
    payoff -300  (0:30:2)
    score 2  (7,17|2,7,16,12,13|8,11,19)

* NN 2 (parent, H=3, [-0.49, -0.05, 0.45])
    payoff -190  (10:20:2)
    score=7  (3,8,9,11,12,14,19|15|4,10)

  ...

* NN 20 (child of 10, H=5, [-0.45, 0.03, 0.48])
    payoff -200  (10:21:1)
    score 8  (7,17|2,7,16,12,13|8,11,19)]
```

```
GENERATION 2

  NN 2 (parent, ...)
    payoff ...
    score ...

* NN 4 (parent, ...)
    payoff ...
    score ...

  ...

  NN 20 (parent, ...)
    payoff ...
    score ...

* NN 21 (child of 2, ...)
    payoff ...
    score ...

  NN 40 (child of 20, ...)
    payoff ...
    score ...

  ...
```

42. Reimplementing Blondie24

Fogel and Chellapilla's Blondie24 was published over the course of two papers. Here we shall address the first paper, *Evolving Neural Networks to Play Checkers without Relying on Expert Knowledge*, published in 1999.

This first version of Blondie24 operated under similar principles as Fogel's tic-tac-toe player, described previously. However, there are a number of important differences that are detailed below.

Neural Network Architecture

The neural net consists of the following layers:

- *Input Layer:* 32 linearly-activated nodes and 1 bias node (checkers board has 64 squares but only half of them are used)
- *First Hidden Layer:* 40 tanh-activated nodes and 1 bias node
- *Second Hidden Layer:* 10 tanh-activated nodes and 1 bias node
- *Output Layer:* 1 tanh-activated node

Additionally, there is a special node called a *piece difference node* whose activity is the sum of the 32 input nodes. The piece difference node connects directly to the output node, bypassing all the layers. The connection, of course, has a variable weight that is learned by the network.

In total, there are 1742 weights (including the weight of the piece difference node).

Converting Board to Input

This is similar to tic-tac-toe in that the player's own regular pieces are labeled with 1, empty squares with 0, and opponent regular pieces with -1. However, the player's own king pieces are labeled with K, and

the opponent's with $-K$, where K is a variable that is learned by the network.

An action is chosen via the minimax algorithm using the following heuristic evaluation function. As the network learns, this heuristic evaluation function will become more accurate.

1. If a board state is a win or a loss, return 1 or -1 respectively.
2. Otherwise, pass the board state as input to the neural network and return the activity of the output node.

The search depth is set to $d = 4$ to allow for reasonable execution times.

Evolution Procedure

The initial population consists of 15 networks with initial weights randomly chosen from the range $[-0.2, 0.2]$, mutation rates set to $\alpha = 0.05$, and $K = 2$. Each network is initialized with the same number of nodes (as described earlier), which remains constant throughout the course of evolution (i.e. nodes are not added nor deleted).

Replication

The evolution procedure follows the same rules as those described previously when evolving neural network regressors.

However, in addition to updating the mutation rate and weights, K is also updated through the following rule:

$$K^{\text{child}} = K^{\text{parent}} e^{N(0,1)/\sqrt{2}}$$

Note that K is constrained to the range $[1, 3]$, meaning that

- if K falls below 1 then it is immediately set to 1, and
- if K rises above 3 then it is immediately set to 3.

Evaluation

Each of the 30 networks in a generation plays a game of checkers against 5 other networks randomly selected (with replacement) from the generation. The network is allowed to move first during each game, and it receives a payoff of 1 for a win, 0 for a tie, and -2 for a loss. (A tie is declared after 100 moves by each player with no winner.)

The 15 networks with the highest total payoffs are selected as the parents of the next generation.

Performance Curve

In their paper, Fogel and Chellapilla did not create a curve to demonstrate performance as a function of number of generations. Instead, they played their final network against human players online and demonstrated that it achieved an impressive performance rating.

Here, we will create a performance curve by playing the evolving networks against an external algorithmic strategy and measuring their performance. This can be accomplished as follows:

1. Develop a heuristic checkers player by hand that plays slightly intelligently. It should capitalize on obvious opportunities to move its pieces forward and jump opponent pieces, but it should not attempt to plan into the future.
2. During each generation of the evolutionary procedure, before replication, play each of the 15 parent networks against your heuristic player and compute the average payoff.
3. Keep evolving new generations until the average payoff levels off.

The resulting plot should show that the average payoff increases with the number of generations (up to some point), demonstrating the that the evolving networks are learning to play checkers intelligently.

Keep in mind that your hand-crafted heuristic checkers player is not actually used during the evolution procedure – it is only used to measure how the evolved networks perform against an external

opponent. So, anything that the evolving networks "learn" is organic and self-taught, not tailored to the specifics of your hand-crafted player.

43. Reimplementing Blondie24: Convolutional Version

Fogel and Chellapilla followed up their 1999 Blondie24 paper with another paper, *Evolving an Expert Checkers Playing Program without Using Human Expertise*, published in 2001.

Convolutional Layer

This paper was very similar to the 1999 paper, but it had one key difference that improved the performance of the evolved players: they inserted a **convolutional layer** between the input layer and first hidden layer in their neural network.

- *Input Layer:* 32 linearly-activated nodes and 1 bias node (checkers board has 64 squares but only half of them are used)
- ***Convolutional Layer:*** one tanh-activated node for each $N \times N$ subsquare of the checkers board with $N = 3, 4, 5, 6, 7, 8$. These nodes also receive input from the bias node in the input layer. Also, this convolutional layer contains 1 bias node that connects to the next layer.
- *First Hidden Layer:* 40 tanh-activated nodes and 1 bias node
- *Second Hidden Layer:* 10 tanh-activated nodes and 1 bias node
- *Output Layer:* 1 tanh-activated node. Note that this node also receives input from the piece difference node.

Recall that the checkers board has dimensions 8×8. So, there is a single 8×8 subsquare (namely, the entire board). Likewise, there are four 7×7 subsquares, nine 6×6 subsquares, sixteen 5×5 subsquares, sixteen 5×5 subsquares, twenty-five 4×4 subsquares, and thirty-six 3×3 subsquares. Including the bias node, the total number of nodes in the convolutional layer is

$$1 + 4 + 9 + 16 + 25 + 36 + 1 = 92.$$

These nodes receive 945 weights from the input layer (including the input bias node):

$$
\begin{aligned}
&1\left(\frac{8^2}{2}+1\right)+4\left(\frac{7^2}{2}+1\right)+9\left(\frac{6^2}{2}+1\right) \\
&\quad+16\left(\frac{5^2}{2}+1\right)+25\left(\frac{4^2}{2}+1\right)+36\left(\frac{3^2}{2}+1\right) \\
&=945
\end{aligned}
$$

The convolutional layer is also known as a *spatial preprocessing layer* because it allows the network to perceive spatial characteristics of the board at different levels of "zoom". Today, most modern image classification systems leverage convolutional neural networks.

With the addition of the convolutional layer, the total number of weights in the Blondie24 neural network increases to 5047, including the weight from the piece difference node to the output layer. These weights are all variable and are learned through the process of evolution.

King Value Update

There was one more minor difference in the 2001 paper. The king value was updated in a slightly different way:

$$K^{\text{child}} = K^{\text{parent}} + \delta$$

where δ is randomly chosen from $\{-0.1, 0, 0.1\}$. The updated value of K is still constrained to range $[1, 3]$.

Performance Curve

Generate a performance curve the same way you did for the previous (non-convolutional) implementation of Blondie24, playing your evolved networks against your heuristic strategy. The curve should look fairly similar but should ideally level off to a slightly higher level of performance.