DSC 40B

Theoretical Foundations II

Lecture 1 | Part 1

Administrative Stuff

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 1

Administrative Stuff

Syllabus

- All course materials, the syllabus, etc., can be found at **course website**[1].

[1] https://akbarrafiey.github.io/DSC40B-SP24/

Syllabus

- All course materials, the syllabus, etc., can be found at **course website**.
- News:
  - No discussion tomorrow.

Syllabus

- All course materials, the syllabus, etc., can be found at **course website**.

- News:
  - No discussion tomorrow.

  - Lab 01 posted, due on Sunday.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 2

What is DSC 40B?

Recall DSC 40A...

- How do we **formalize** learning from data?
- How do we turn it into something a **computer** can do?

Example 1: Minimize Absolute Error

- **Goal**: summarize a collection of numbers, $x_1, \dots, x_n$:
- **Idea**: find number M minimizing the total absolute error:

$$\sum_{i=1}^{n} |M - x_i|$$

Example 1: Minimize Absolute Error

- **Solution**: The **median** of $x_1, \ldots, x_n$.

The End

The End?

Example 1: Minimize Absolute Error

- How do we actually **compute** the median?

Exercise

Suppose you're on a desert island with no internet connection, but a basic installation of Python. For some reason, you need to compute the median of a million numbers to get off of the island.

How do you do it?

Main Idea

Our work doesn't stop once we solve the math problem (*a la* DSC 40A).

We still need to **compute** the answer.

We need an **algorithm**.

Main Idea

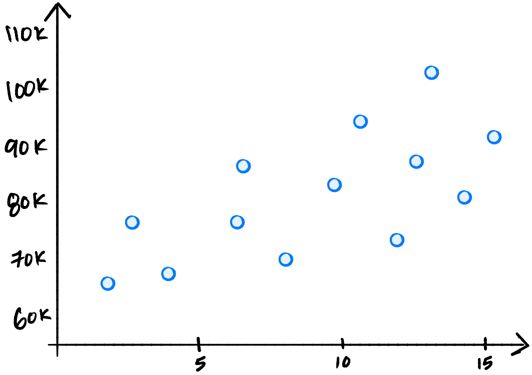
Our work doesn't stop once we solve the math problem (*a la* DSC 40A).

We still need to **compute** the answer.

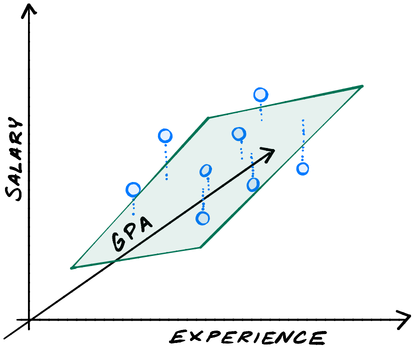
We need an **algorithm**.

More than that, we need an **implementation** of that algorithm (that is: code).

Example 2: Least Squares Regression



Example 2: Least Squares Regression



The End

$$(X^T X)\vec{w} = X^T\vec{b}$$

Wait...

- We actually need to **compute** the answer...
- We need an **algorithm**.

An Algorithm?

- Let's say we have `numpy` installed.
- It provides an implementation of an algorithm:

```
>>> import numpy as np
>>> w = np.linalg.solve(X.T @ X, X.T @ b)
```

But...

- Will it work for 1,000,000 data points?
- What about for 1,000,000 features?

Main Idea

Having an algorithm isn't enough – we need to know about its performance. Otherwise, it may be useless for our particular problem.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 3

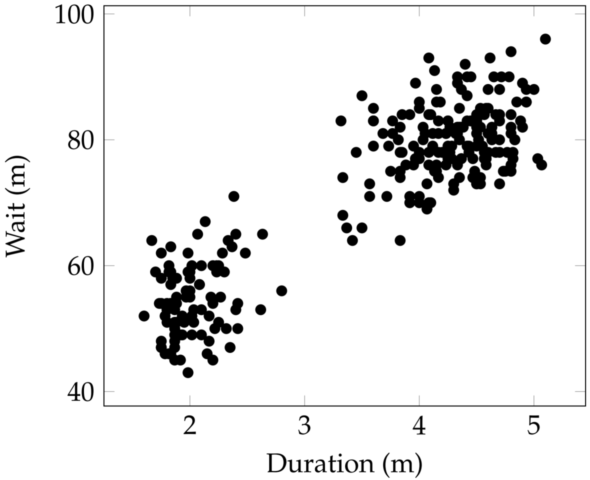
Example: Clustering

Clustering

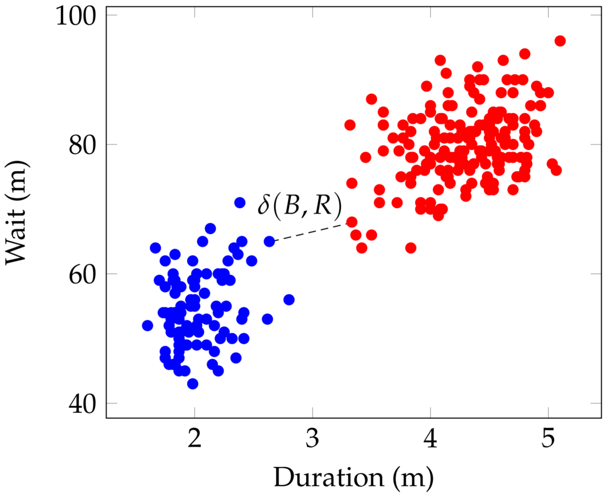
- Given a pile of data, discover similar groups.
- Examples:
  - Find political groups within social network data.
  - Given data on COVID-19 symptoms, discover groups that are affected differently.
  - Find the similar regions of an image (**segmentation**).
- Most useful when data is high dimensional...

Example: Old Faithful

Example: Old Faithful



Example: Old Faithful

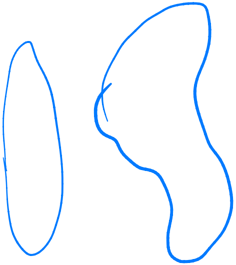


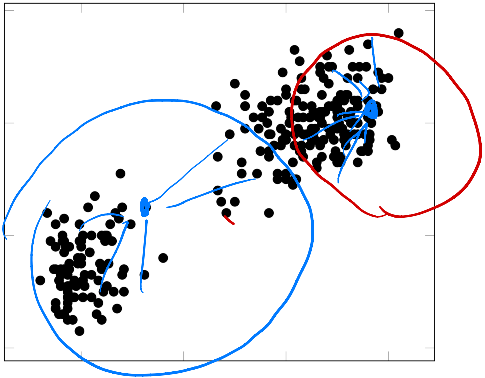
Clustering

- Goal: for computer to identify the two groups in the data.
- A clustering is an assignment of a color to each data point.
- There are many possible clusterings.

Clustering

- How do we turn this into something a **computer** can do?
- DSC 40A says: “Turn it into an optimization problem”.
- Idea: **design** a way of quantifying the “goodness” of a clustering; find the **best**.
  - Design a **loss function**.
  - There are many possibilities, tradeoffs!

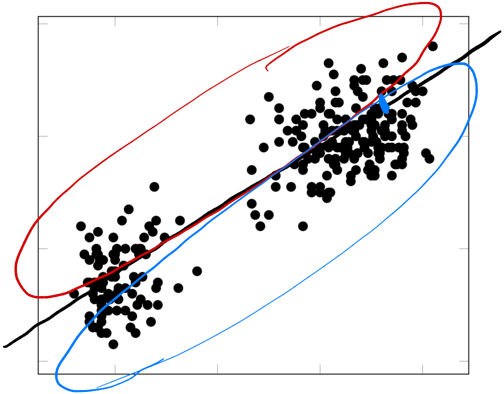




Exercise

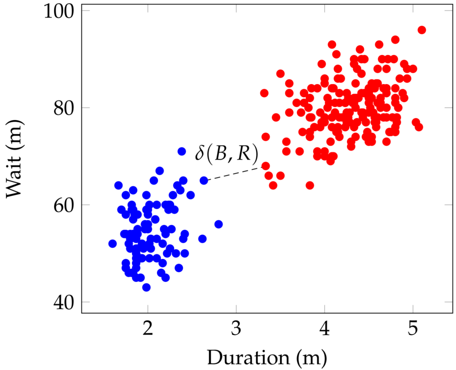
What's a good loss function for this problem? It should assign small loss to a **good** clustering.

Quantifying Separation



Idea: Define the "separation" $\delta(B, R)$ to be the *smallest* distance between a blue point and red point.

Quantifying Separation



Idea: Define the "separation" $\delta(B, R)$ to be the *smallest* distance between a blue point and red point.

The Problem

- **Given**: n points $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$.
- **Find**: an assignment of points to clusters R and B so as to maximize $\delta(B, R)$.

DSC 40A: “The End”

DSC 40A: “The End”

DSC 40B: “The Beginning”

The “Brute Force” Algorithm

- There are finitely-many possible clusterings.
- **Algorithm**: Try each possible clustering, return that with largest separation, $\delta(B, R)$.
- This is called a **brute force** algorithm.

```python
best_separation = -float('inf') # Python for "infinity"
best_clustering = None

for clustering in all_clusterings(data):
    sep = calculate_separation(clustering)
    if sep > best_separation:
        best_separation = sep
        best_clustering = clustering

print(best_clustering)
```

The Algorithm

- We have an **algorithm**!
- But how long will this take to run if there are n points?
- How many clusterings of n things are there?

Exercise

How many ways are there of assigning R or B to n points?

Solution

- Two choices [2] for each object: $2 \times 2 \times \ldots \times 2 = 2^n$.

[2] Small nitpick: actual color doesn't matter, 2^{n-1}.

Time

- Suppose it takes at least *1 nanosecond*[3] to check a single clustering.
  - One *billionth* of a second.
  - Time it takes for light to travel 1 foot.

- If there are n points, it will take *at least* 2^n nanoseconds to check all clusterings.

---

[3] This is an *extremely* optimistic estimate. It's actually much slower, and scales with n.

Time Needed

n	Time
1	1 nanosecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days
60	36 years

Time Needed

n	Time
1	1 nanosecond
10	1 microsecond
20	1 millisecond
30	1 second
40	18 minutes
50	13 days
60	36 years
70	37,000 years

Example: Old Faithful

- The Old Faithful data set has 270 points.

- Brute force algorithm will finish in 6×10^{64} years.

Example: Old Faithful

- The Old Faithful data set has 270 points.
- Brute force algorithm will finish in 6×10^{64} years.

Algorithm Design

- Often, most obvious algorithm is **unusably slow**.

Algorithm Design

- Often, most obvious algorithm is **unusably slow**.
- Does this mean our problem is too hard?
  - Direct result of our choice of loss function.

Algorithm Design

- Often, most obvious algorithm is **unusably slow**.
- Does this mean our problem is too hard?
  - Direct result of our choice of loss function.
- We'll see an efficient solution by the end of the quarter.

Main Idea

Just having an algorithm isn't enough – it must also be reasonably **efficient**. Otherwise, it might be useless for our particular problem.

DSC 40B

- Assess the efficiency of algorithms.
- Understand why and how common algorithms work.
- Develop faster algorithms using design strategies and data structures.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 4

Measuring Efficiency by Timing

Efficiency

- Speed matters, *especially* with large data sets.
- An algorithm is only useful if it runs **fast enough**.
  - That depends on the size of your data set.
- How do we measure the efficiency of code?
- How do we know if a method will be fast enough?

Scenario

- You're building a least squares regression model to predict a patient's blood oxygen level.
- You've trained it on 1,000 people.
- You have a full data set of 100,000 people.
- How long will it take? How does it **scale**?

Example: Scaling

- Your code takes 5 seconds on 1,000 points.
- How long will it take on 100,000 data points?
- 5 seconds × 100 = 500 seconds?
- More? Less?

Coming Up

- We’ll answer this in coming lectures.
- Today: start with simpler algorithms for the mean, median.

Approach #1: Timing

- How do we measure the efficiency of code?
- Simple: time it!
- Useful Jupyter tools: `time` and `timeit`

In [1]:
```
numbers = range(1000)
```

In [3]:
```
%%time
sum(numbers)
```

```
CPU times: user 13 µs, sys: 0 ns, total: 13 µs
Wall time: 13.8 µs
```

Out[3]:
```
499500
```

In [4]:
```
%%timeit
sum(numbers)
```

```
10.8 µs ± 509 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Disadvantages of Timing

1. Time depends on the computer.

Disadvantages of Timing

1. Time depends on the computer.

2. Depends on the particular input, too.

Disadvantages of Timing

1. Time depends on the computer.
2. Depends on the particular input, too.
3. One timing doesn’t tell us how algorithm **scales**.

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 5

Measuring Efficiency by Counting Operations

Approach #2: Time Complexity Analysis

- Determine efficiency of code **without** running it.
- Idea: find a formula for time taken as a function of input size.

Advantages of Time Complexity

1. Doesn’t depend on the computer.
2. Reveals which inputs are “hard”, which are “easy”.
3. Tells us how algorithm scales.

Exercise

Write a function `mean` which takes in a NumPy array of floats and outputs their mean.

```python
def mean(numbers):
    total = 0
    n = len(numbers)
    for x in numbers:
        total += x
    return total / n
```

total = total + x

Time Complexity Analysis

- How long does it take `mean` to run on an array of size n? Call this $T(n)$.
- We want a formula for $T(n)$.

Counting Basic Operations

- Assume certain basic operations (like adding two numbers) take a constant amount of time.
  - `x + y` doesn't take more time if `numbers` is bigger.
  - So `x + y` takes "constant time"
  - Compare to `sum(numbers)`. **Not** a basic operation.

- **Idea**: Count the number of basic operations. This is a measure of time.

Exercise

Which of the below array operations takes constant time?

- accessing an element: `arr[i]` yes
- asking for the length: `len(arr)` yes
- finding the max: `max(arr)` no

Basic Operations with Arrays

We'll assume that these operations on NumPy arrays take **constant time**.

- accessing an element: `arr[i]`
- asking for the length: `len(arr)`

Example

```python
def mean(numbers):
    total = 0
    n = len(numbers)
    for x in numbers:
        total += x
    return total / n
```

Line	Time/exec.	# of execs.
`total = 0`	c_1	1
`n = len(numbers)`	c_2	1
`for x in numbers:`	c_3	$n+1$
`total += x`	c_4	n
`return total / n`	c_5	1

$$T(n) = c_1 + c_2 + c_3(n+1) + c_4 \cdot n + c_5 = (c_4 + c_3)n + (c_1 + c_2 + c_3 + c_5)$$

Example: `mean`

- Total time:

$$
\begin{aligned}
T(n) &= c_3(n+1) + c_4 n + (c_1 + c_2 + c_5) \\
&= (c_3 + c_4)n + (c_1 + c_2 + c_3 + c_5)
\end{aligned}
$$

- "Forgetting" constants, lower-order terms with "Big-Theta": $T(n) = \Theta(n)$.

- $\Theta(n)$ is the **time complexity** of the algorithm.

Main Idea

Forgetting constant, lower order terms allows us to focus on how the algorithm **scales**, independent of which computer we run it on.

Careful!

- Not always the case that a single line of code takes constant time per execution!

Example

```
def mean_2(numbers):
    total = sum(numbers)
    n = len(numbers)
    return total / n
```

Time/exec.	# of execs.
	1
$c_1 n + c_0$	
c_2	1
c_3	1

$$T(n) = c_1 n + c_0 + c_2 + c_3 \longrightarrow T(n) = \Theta(n)$$

Example: `mean_2`

- Total time:

$$T(n) = c_1 n + (c_0 + c_2 + c_3)$$

- "Forgetting" constants, lower-order terms with "Big-Theta": $T(n) = \Theta(n)$.

Exercise

Write an algorithm for finding the maximum of an array of n numbers. What is its time complexity?

```python
def maximum(numbers):
    current_max = -float('inf')
    for x in numbers:
        if x > current_max:
            current_max = x
    return current_max
```

	Time/exec.	# of execs.
`current_max = -float('inf')`	c_1	1
`for x in numbers:`	c_2	$n+1$
`if x > current_max:`	c_3	n
`current_max = x`	c_4	$1 \ldots n$
`return current_max`	c_5	1

$$T(n) = \Theta(n)$$

Main Idea

Using Big-Theta allows us not to worry about *exactly* how many times each line runs.

By the way...

Approximate timing for various operations on a typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

From Peter Norvig's essay, "Teach Yourself Programming in Ten Years"
`http://norvig.com/21-days.html`

Remaining Questions

- What if the code is more complex?
  - For example, nested loops.
- What is this notation anyways?

Remaining Questions

- What if the code is more complex?
  - For example, nested loops.
- What is this notation anyways?
- Next time in DSC 40B.