DSC 40B

Theoretical Foundations II

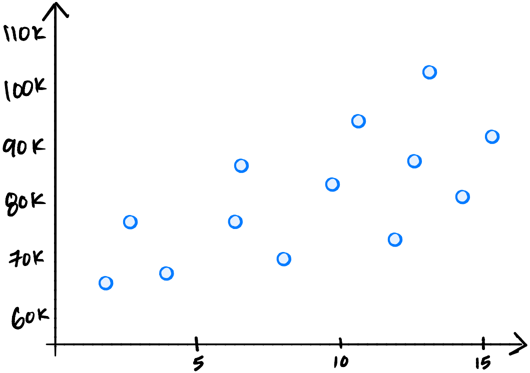
Lecture 1 | Part 1

What is DSC 40B?

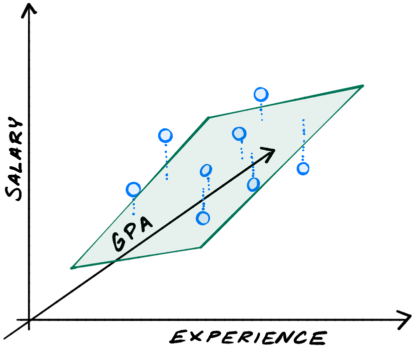
Recall DSC 40A...

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

Example: Predicting Salary



Example: Predicting Salary



The End

$$(X^T X)^{-1}\vec{w} = X^T\vec{b}$$

Wait...

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

An Algorithm?

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

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

Example: Minimize 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: Minimize Error

- **Solution**: The **median** of $x_1, \ldots, x_n$.
- But how do we actually **compute** the median?

DSC 40B

Theoretical Foundations II

Lecture 1 | Part 2

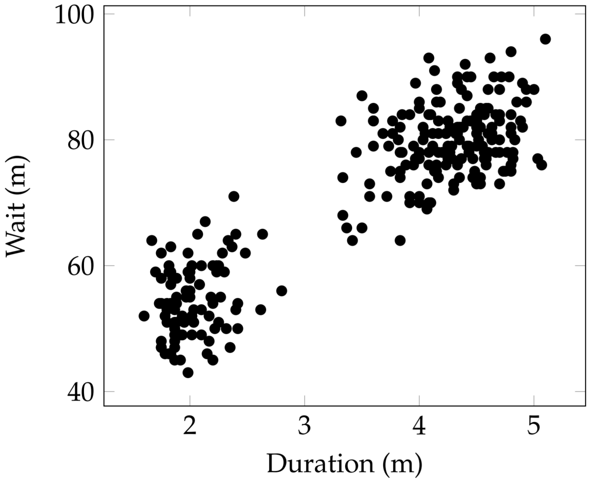
Example: Clustering

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

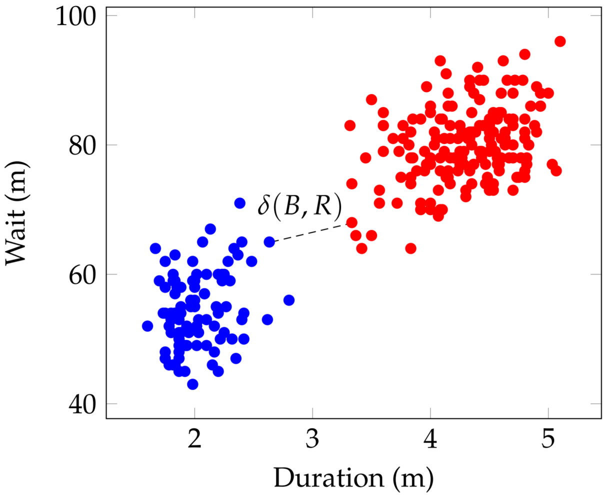
Example: Old Faithful



Clustering

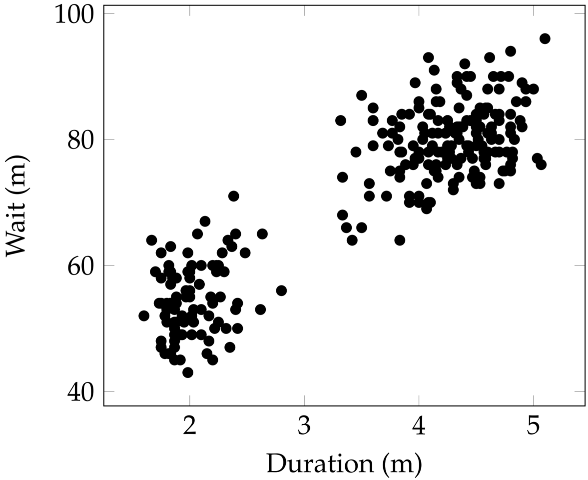
- Goal: for computer to identify the two groups in the data.

Example: Old Faithful



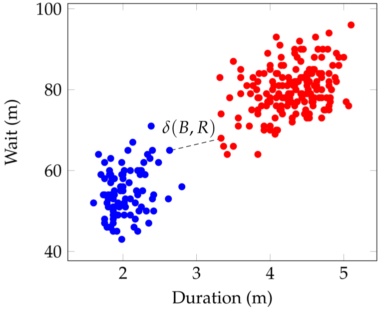
Clustering

- How do we turn this into something a **computer** can do?
- DSC 40A says: “Turn it into an optimization problem”.
- **Idea**: develop a way of quantifying the “goodness” of a clustering; find the **best**.



100
80
60
40
Wait (m)
2
3
4
5
Duration (m)

Quantifying Separation



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 $\delta(B, R)$.

The End

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 End

Wait...

- How long will this take to run if there are n points?
- How many clusterings of n things are there?

Combinatorics

- How many ways are there to assign R or B to n objects?
- Two choices [1] for each object: $2 \times 2 \times \ldots \times 2 = 2^n$.

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

Time

- Suppose it takes at least *1 nanosecond* to check a single clustering.
  - One *billionth* of a second.
- If there are n points, it will take *at least* 2^n nanoseconds to check all clusterings.

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.

Algorithm Design

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

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 3

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`

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 4

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
```

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 $T(n)$.
- **Idea**:
  - Assume certain basic operations (like adding two numbers) take a constant amount of time.
  - Count total time taken by basic operations.

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

Time/exec. **# of execs.**

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

Example: `mean`

- Total time:

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

- "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

Time/exec. **# of execs.**

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

Time/exec. **# of execs.**

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

Main Idea

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