DSC40B: Theoretical Foundations of Data Science II

Lecture 1: *Welcome, introduction and examples*

Instructor: Yusu Wang

Prelude

Course Information

- Course webpage:
  - http://dsc40b.com
- Office hours:
  - Tue@ *9:00am – 10:00am*
  - always feel free to send me emails with questions / concerns
    - yusuwang@ucsd.edu
  - using *Campuswire* message board may be faster to get an answer

Course Information

- Homework and Exams:
  - Around 9 labs, and 8 homework (roughly one per week other than the last)
    - Check out Homework Redemption policy on course webpage
  - 2 midterms:
    - Midterm 1: Feb 9; Midterm 2: March 9
    - Check out Midterm Redemption policy on course webpage
  - Redemption midterms
    - March 23
  - 1 Super-homework (as substitute for final exam):
    - Due in the final exam week

Course Information

- Grading:
  - 12.5%: Labs
  - 30%: Homework Assignments
  - 7.5%: Final super-homework
  - 25%: Midterm 01
    - (or Redemption Midterm 01, whichever is larger)
  - 25%: Midterm 02
    - (or Redemption Midterm 02, whichever is larger)
  - Passing threshold: ≥60% average in two midterm in order to pass!

Course Information

- Slip Days
  - You have 5 "slip days" to use throughout the quarter. A slip day extends the deadline of any one homework by 24 hours. Slip days can't be "stacked“. Slip days are applied automatically at the end of the quarter, but it's your responsibility to keep track of how many you have left.
    - Slips days cannot be used for the homework prior to the two midterms

- Homework collaboration
  - You may discuss homework with your classmates – in fact, it is sometimes encouraged. It is very important that you write up your solutions individually.

Course Materials

- Couse note by *Justin Eldridge* (*JE*)
  - https://ucsd-ets.github.io/dsc40b-2021-wi-public/published/default/notes/book.pdf

- Other optional textbooks:
  - Cormen, Leiserson, Rivest, Stein (*CLRS*), *Intruction to Algorithms*
  - Dasgupta, Papadimitriou, Vazirani, *Algorithms*

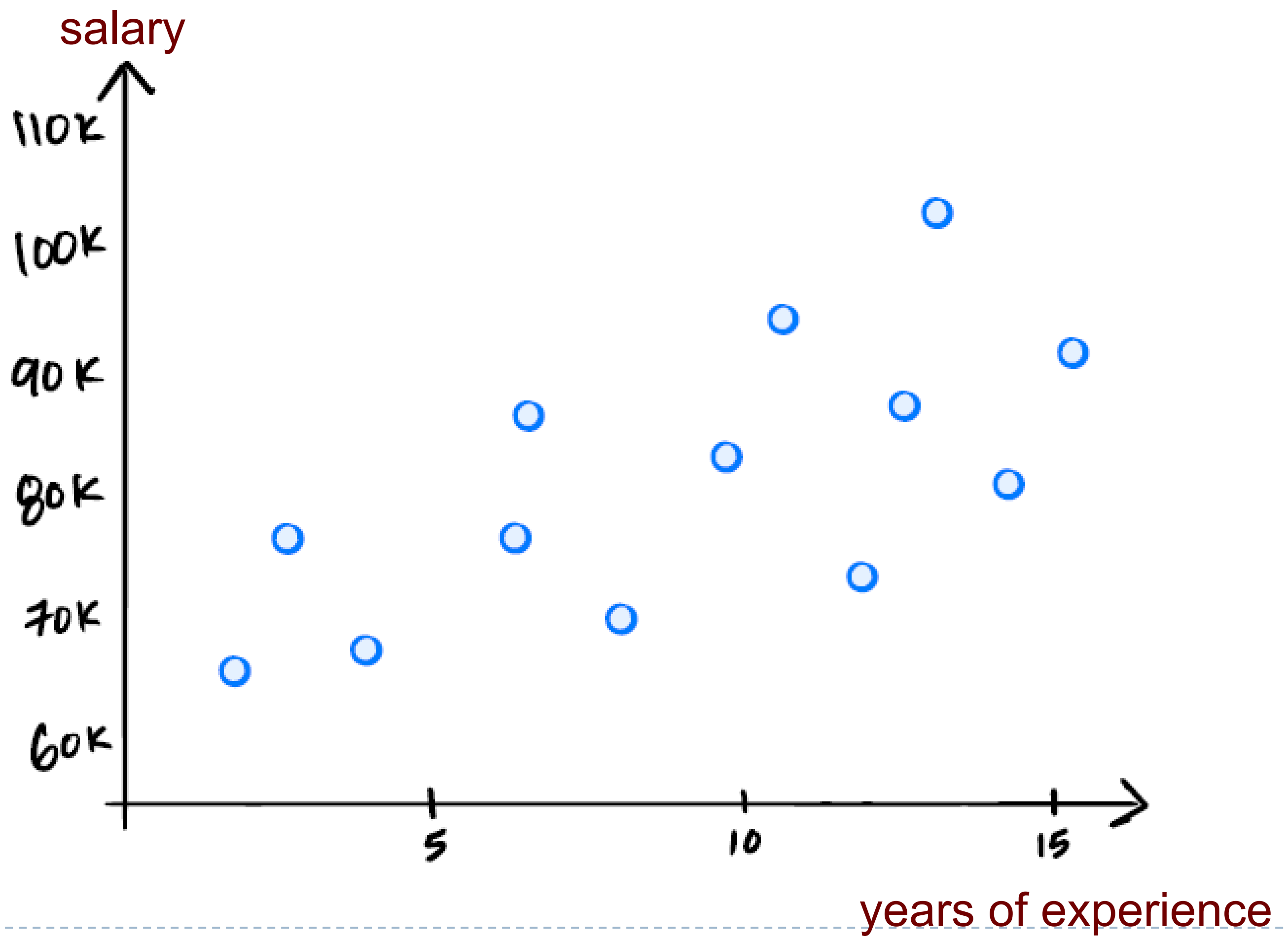
Introduction to the course

Recall in DSC 40 A

- How do we formalize learning from data
- How do we model it in a precise way so that a computer could potentially tackle it

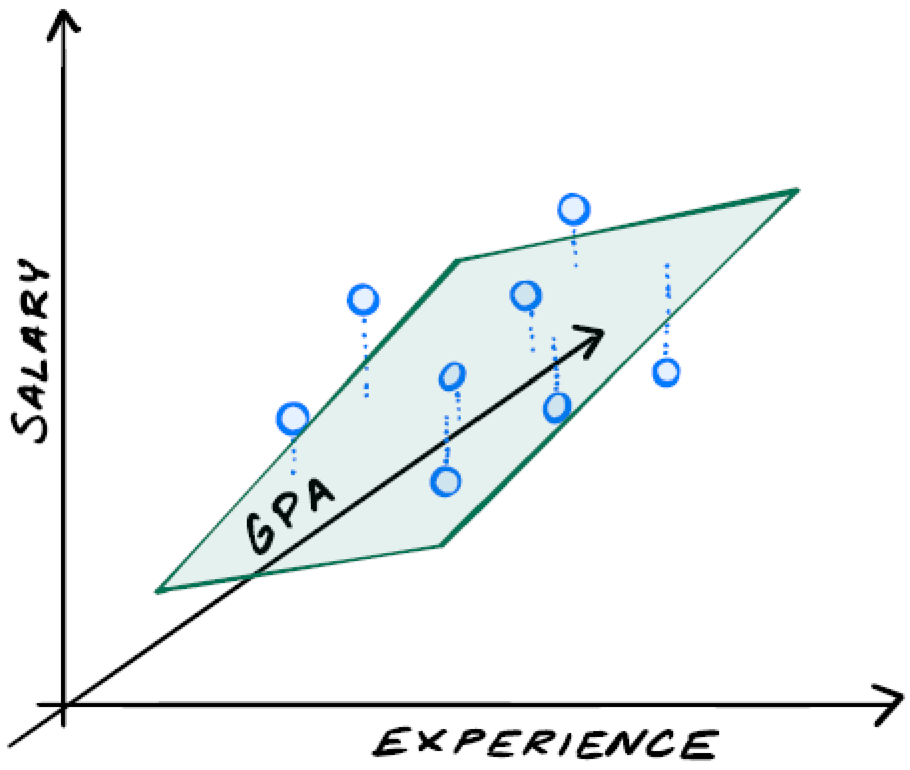
A simple example

- Salary prediction from existing data



A simple example

- Salary prediction from existing data



- Formulation (linear regression):
  - Find the best (hyper-)plane fitting these points with least total error (sum-square distances)
  - $X^T X \vec{\omega} = X^T \vec{b}$

Is that the end?

But ....

- How do we really compute it?
- How do we ask the computer to compute it for us?

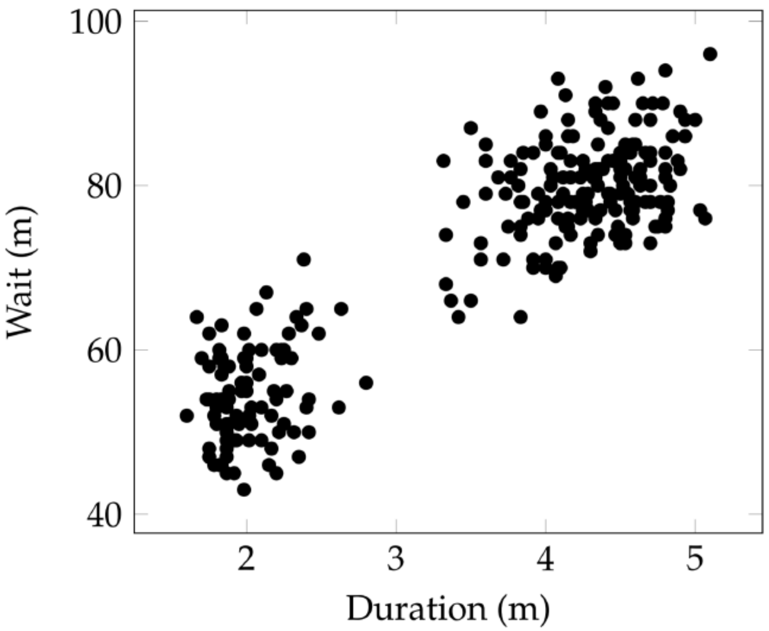
```
>>> import numpy as np
>>> w = np.linalg.solve(X.T @ X, X.T @ b)
```

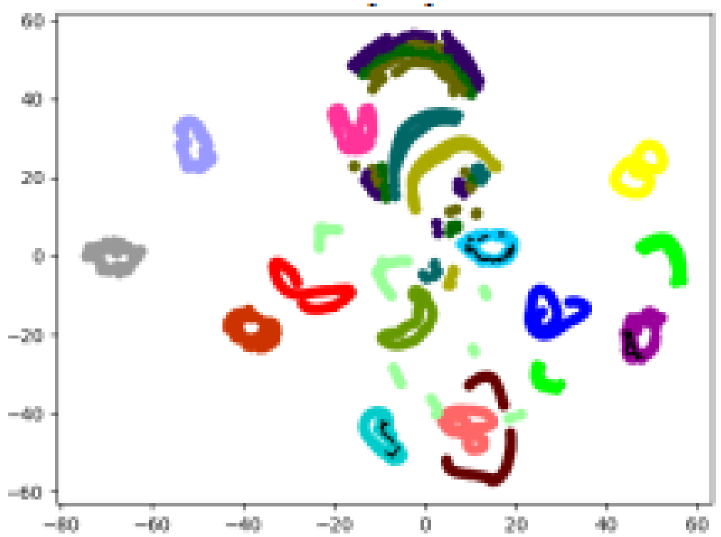
- This is an algorithm
  - a sequence of steps / operations to achieve a goal

- How do we know this is a “good” algorithm?
  - How fast does it run on 1,000 points?
  - How does it scale to 1,000,000 points?
  - Can we come up with better algorithms for this?

A second example

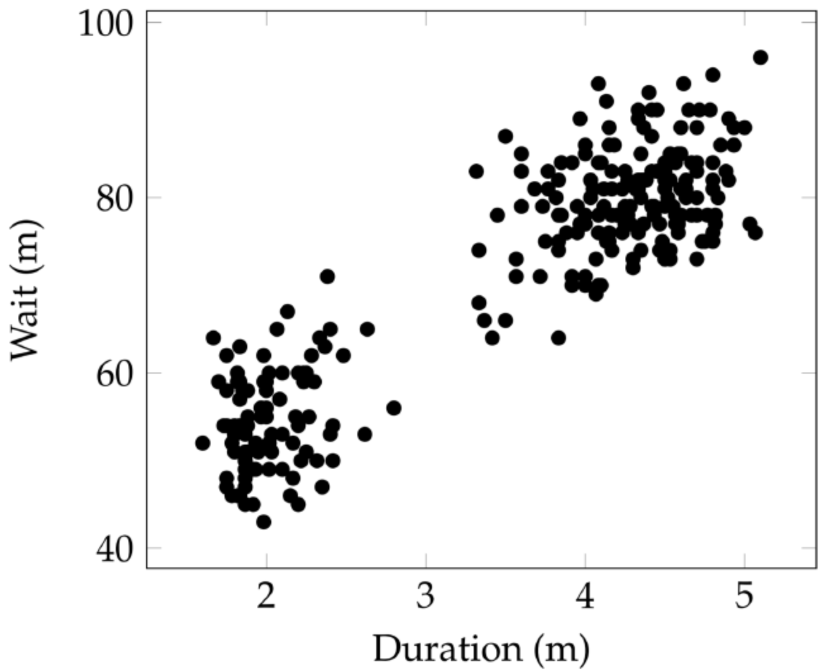
- Clustering
  - Given a set of data, identify "groups" (clusters) such that "similar" data points are grouped together
  - Ubiquitous across science and engineering





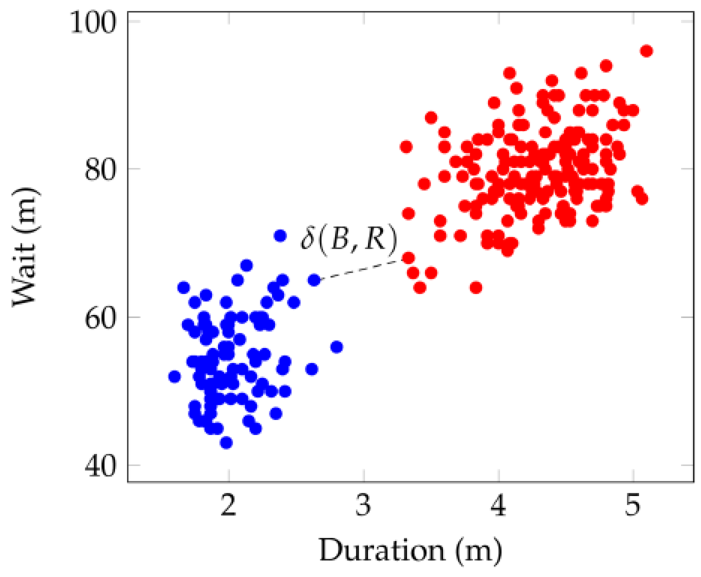
Old Faithful Geyser

- What's the pattern behind its eruption?
- How to define as well as find the two clusters?



DSC40A says

- Let's model this as an optimization problem
  - first develop a way to measure / quantify the "goodness" of different grouping
  - then compute the best grouping with highest goodness score



- A grouping candidate:
  - assign each point to be either blue or red
- Quality (goodness):
  - min separation distance $\delta(B, R)$
    - smallest distance between a red or blue point
- Goal:
  - given points $X = \{x_1, \ldots, x_n\}$
  - return the assignment of $X = B \cup R$ with largest separation distance $\delta(B, R)$

Are we done?

- But… how do we really solve this optimization problem?

- What is an algorithm to achieve this?
  - The algorithm needs to be correct
- If we develop a correct algorithm, is this algorithm good?
  - How do we know? (aka: how to analyze our algorithm?)
- How do we design better algorithms?

A first algorithm for clustering

- Intuition: to compute min-separation grouping
  - try all possible assignment of all input data points
  - compute separation distance for each assignment (grouping)
  - return the one with largest separation distance (the best)

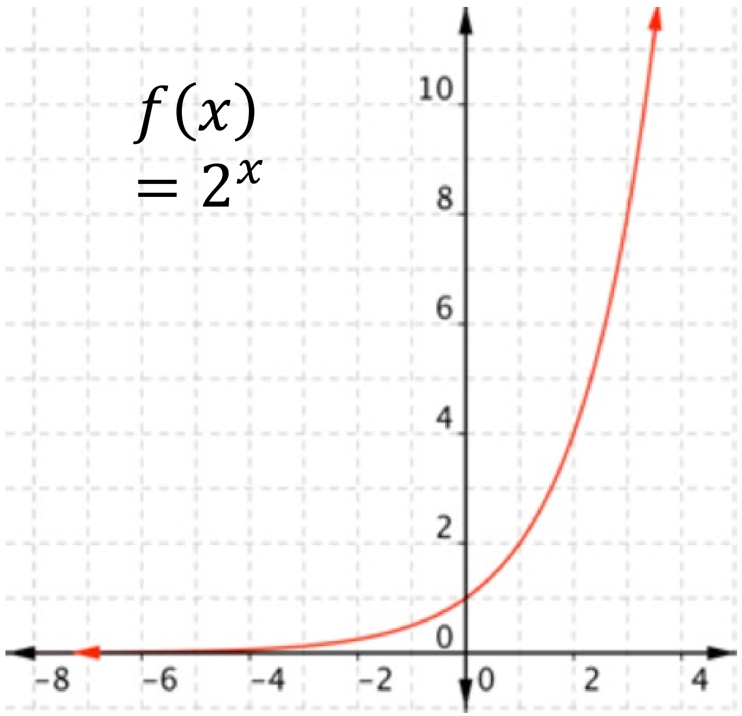
```
best_separation = float('inf') # Python for "infinity"
best_clustering = None

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

print(best_clustering)
```

Running time?

- Precise time:
  - depends on the computer
- Rough idea to make measuring time "computer-independent":
  - Let's count how many operations we would need!
    - How many possible assignment do we have?
      - assigning *R* or *B* to each input point
      - $2 \times 2 \times \cdots \times 2 = 2^n$
  - Suppose it takes 1 nanosecond to check one grouping
  - Takes 2^n nanoseconds in total

- 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



- Clearly not efficient. Can we do better? How to design better algorithms?

This course: DSC40B

- DSC40A how to model / formulate a problem to tackle an input task
- DSC40B: how we then solve it efficiently!

- Focus on "algorithms"
  - What is an algorithm?
    - step by step strategy to solve problems
    - should terminate and return correct answer for any input instance

This course: DSC40B

- Often, obvious algorithms might not be "efficient"
  - Doesn't mean that the problem at hand is hard!

- Various issues involved in designing good algorithms
  - What do we mean by "good"?
    - algorithm analysis, asymptotic language used to measure performance
  - How to design good algorithms
    - data structures, algorithm paradigms
  - Fundamental problems
    - graph traversal, shortest path etc.

How to measure efficiency?
Time complexity

Efficiency

- An algorithm takes an input and performs a task (which could produce an output)
- Efficiency matters!
  - Running time?
  - How much memory it needs?
  - An algorithm is only useful when it is efficient enough
- How do we measure time efficiency?
- Note: running time
  - Depends on computing environments
  - depends on size of input
  - depends on specific input too

Scenario

- Suppose you are building a least-square regression model to predict oxygen level of a patient based on various parameters measured about her
- You developed an algorithm and trained it on $1{,}000$ patients
  - it runs 1 second
- What if you now want to train on $1{,}000{,}000$ patients?
  - is it 1000 seconds?
- We should analyze how the runtime scales
  - Growth of running time w.r.t. input size
  - But how?

- We will answer the question of "time complexity" in coming lectures
- Today:
  - Some simple examples to provide intuition

Approach #1: just time it!

- How about we time it
  - e.g, using Jupyter tools: *time* and *timeit*

- Disadvantages:
  - depending on the machine
  - to get the growth, need to perform multiple runs and plot the time to obtain the pattern
    - one timing doesn't tell how the algorithm scales
  - even then, the time may depend on specific input
    - input of the same size could incur different running time

Approach 2: Time complexity analysis

- Count intuitively just operations
  - eliminate the precise running time of a specific computer
- Determine it by analyzing the code without running it
- Obtain a function (a formula) on how the ``count'' changes with input size (i.e, growth w.r.t input size)
  - instead of timing or plotting to find how time scales

An example

- Consider the following code
  - input is an array A

```
def Func(A):
        total = 0
        n = len(A)
        for i in range(n):
                total += A[i]
        return total / n
```

An example

- Consider the following code
  - input is an array A of numbers

```
def Mean(A):
	total = 0
	n = len(A)
	for i in range(n):
		total += A[i]
	return total / n
```

Time complexity

- First abstraction: depending on input size
  - consider function $T(n)$,
  - aim to obtain a formula for $T(n)$ to capture growth of the running time w.r.t input size

- Second abstraction:
  - Assume that basic operations take constant time
    - e.g, all operations such as addition/multiplication/division takes constant time each
    - e.g, for an array A, access $A[i]$ takes constant time

Back to our example

- Consider the following code
  - input is an array A

```
def mean(A):
        total = 0
        n = len(A)
    for i in [0,n-1]:
            total += A[i]
    return total / n
```

c_1

c_2

c_3

- $T(n) = c_1 + \sum_{i=0}^{n-1} c_2 + c_3 = \mathrm{c}_2\mathrm{n} + \mathrm{c}_1 + \mathrm{c}_3$

Time complexity

- First abstraction: depending on input size
  - Consider function $T(n)$, and aim to obtain a formula for $T(n)$
- Second abstraction:
  - Assume that basic operations take constant time
    - e.g, all operations such as addition/multiplication/division takes constant time each
    - for an array A and index i, access $A[i]$ takes constant time
- Third abstraction:
  - Ignore constants as well as lower order terms,
    - focus on dominating terms to capture relative growth w.r.t. n
  - $T(n) = c_1 + \sum_{i=1}^{n} c_2 + c_3 = c_2 n + c_1 + c_3 = \Theta(n)$

Putting everything together

- Consider the following code
  - input is an array A

```
def mean(A):
        total = 0
        n = len(A)
    for i in [0,n-1]:
            total += A[i]
    return total / n
```

c_1 (total = 0, n = len(A))

c_2 (total += A[i])

c_3 (return total / n)

Algorithm mean(A) has time complexity $T(n) = \Theta(n)$

- $T(n) = c_1 + \sum_{i=0}^{n-1} c_2 \ + c_3 = c_2 n + c_1 + c_3 = \Theta(n)$

Caution

- Note, it is not true that each single command line in the code will take constant time
- Example:

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

- $T(n) = a_1 n + a_2 + a_3 = \Theta(n)$

A simple exercise

- Input: an array A of n numbers
- Output: return the largest number in A
- What is the time complexity of your algorithm?

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

Time complexity

- First abstraction:
  - aim to obtain $T(n)$ where n is size of input
- Second abstraction:
  - assume that basic operations take constant time
- Third abstraction:
  - Ignore constants as
  - $T(n) = \Theta(n)$ for p

Next time
We will learn what exactly are asymptotic time complexity and go through more examples

- $\Theta(\cdot)$ is called asymptotic notation, and a time complexity of this form is called asymptotic time complexity

FIN