

# Algorithms for Data Science

## CSOR W4246

Eleni Drinea  
*Computer Science Department*

Columbia University

More dynamic programming: sequence alignment

# Outline

- 1 Recap
  - A Dynamic Programming solution
- 2 Sequence alignment

# Today

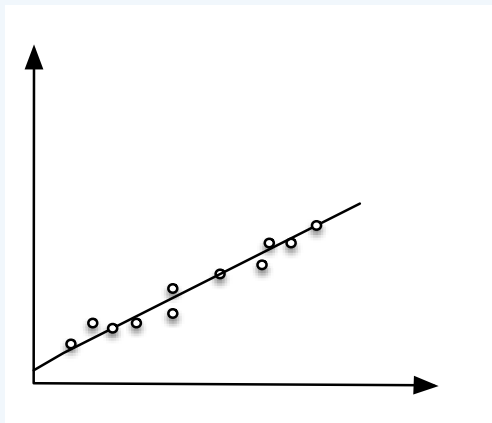
## 1 Recap

- A Dynamic Programming solution

## 2 Sequence alignment

# Linear least squares fitting

A foundational problem in statistics: find a line of *best fit* through some data points.



## A first problem: linear least squares fitting

**Input:** a set  $P$  of  $n$  data points  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ ; we assume  $x_1 < x_2 < \dots < x_n$ .

**Output:** the line  $L$  defined as  $y = ax + b$  that **minimizes** the *sum of the vertical distances of the points from the line*:

$$\text{err}(L, P) = \sum_{i=1}^n (y_i - ax_i - b)^2 \quad (1)$$

# Linear least squares fitting: solution

Given a set  $P$  of data points, we can use calculus to show that the line  $L$  given by  $y = ax + b$  that minimizes

$$\text{err}(L, P) = \sum_{i=1}^n (y_i - ax_i - b)^2 \quad (2)$$

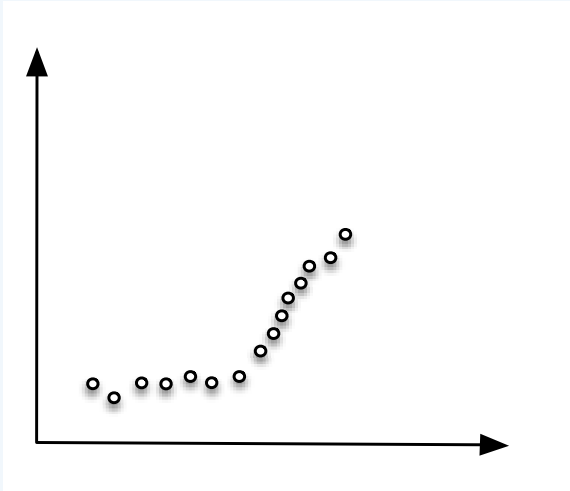
satisfies

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \quad (3)$$

$$b = \frac{\sum_i y_i - a \sum_i x_i}{n} \quad (4)$$

*How fast can we compute  $a, b$ ?*

*What if the data changes direction?*



# Formalizing Segmented Least Squares

**Input:** data set  $P = \{p_1, \dots, p_n\}$  of points on the plane.

- ▶ A **segment**  $S = \{p_i, p_{i+1}, \dots, p_j\}$  is a contiguous subset of  $P$ .
- ▶ Let  $\mathcal{A}$  be a **partition** of  $P$  into  $m_{\mathcal{A}}$  segments  $S_1, S_2, \dots, S_{m_{\mathcal{A}}}$ .  
For every segment  $S_k$ , use (2), (3), (4) to compute a line  $L_k$  that minimizes  $err(L_k, S_k)$ .
- ▶ Let  $C > 0$  be a fixed multiplier. The **cost** of partition  $\mathcal{A}$  is

$$\sum_{S_k \in \mathcal{A}} err(L_k, S_k) + m_{\mathcal{A}} \cdot C$$

**Output:** a partition of minimum cost, and its cost.



# A recurrence for the optimal solution

**Notation:** let  $e_{i,j} = \text{err}(L, \{p_i, \dots, p_j\})$ , for  $1 \leq i \leq j \leq n$ .

- ▶  $OPT(n) = \min_{1 \leq i \leq n} \{e_{i,n} + C + OPT(i-1)\}.$
- ▶ Applying the above expression recursively to remove the last segment, we obtain the recurrence

$$OPT(j) = \min_{1 \leq i \leq j} \{e_{i,j} + C + OPT(i-1)\} \quad (5)$$

## Remark 1.

1. We can precompute and store all  $e_{i,j}$  using equations (2), (3), (4) in  $O(n^3)$  time. *Can be improved to  $O(n^2)$ .*
2. The natural recursive algorithm arising from recurrence (5) is **not** efficient (think about its recursion tree!).

# Elements of DP in segmented least squares

1. **Overlapping subproblems**
2. **An easy-to-compute recurrence (5)** for combining solutions to the smaller subproblems into a solution to a larger subproblem in  $O(n)$  time (once smaller subproblems have been solved).
3. **Iterative, bottom-up computations:** compute the subproblems from smallest (0 points) to largest ( $n$  points), iteratively.
4. Small number of subproblems: we only need to solve  $n$  subproblems.

# A dynamic programming approach

$$OPT(j) = \min_{1 \leq i \leq j} \left\{ e_{i,j} + C + OPT(i-1) \right\}$$

- ▶ The optimal solution to the subproblem on  $p_1, \dots, p_j$  contains optimal solutions to smaller subproblems.
- ▶ Recurrence 5 provides an **ordering** of the subproblems from **smaller to larger**: the subproblem of size 0 is the smallest, the subproblem of size  $n$  is the largest.
- ▶ Boundary condition:  $OPT(0) = 0$ .
- ⇒ There are  $n + 1$  subproblems in total. Solving the  $j$ -th subproblem requires  $\Theta(j) = O(n)$  time.
- ⇒ The overall running time is  $O(n^2)$ .
- ▶ Segment  $p_k, \dots, p_j$  appears in the optimal solution when the minimum in the expression above is achieved for  $i = k$ .

# An iterative algorithm for segmented least squares

Let  $M$  be an array with  $n$  entries such that

$M[i]$  = cost of optimal partition of the first  $i$  data points

**SegmentedLS**( $n, P$ )

$M[0] = 0$

**for** all pairs  $i \leq j$  **do**

    Compute  $e_{i,j}$  for segment  $p_i, \dots, p_j$  using (2), (3), (4)

**end for**

**for**  $j = 1$  to  $n$  **do**

$M[j] = \min_{1 \leq i \leq j} \{e_{i,j} + C + M[i - 1]\}$

**end for**

Return  $M[n]$

**Running time:** time required to fill in dynamic programming array  $M$  is  $O(n^3) + O(n^2)$ . **Can be brought down to  $O(n^2)$ .**

# Reconstructing an optimal segmentation

We can reconstruct the optimal partition **recursively**, using array  $M$  and error matrix  $e$ .

OPTSegmentation( $j$ )

**if** ( $j == 0$ ) **then** return

**else** // find the first point of the segment where  $p_j$  belongs

    Find  $1 \leq i \leq j$  such that  $M[j] = e_{i,j} + C + M[i - 1]$

    OPTSegmentation( $i - 1$ )

    Output segment  $\{p_i, \dots, p_j\}$

**end if**

- ▶ Initial call: OPTSegmentation( $n$ )
- ▶ *Running time?*

# Obtaining efficient algorithms using DP

1. **Optimal substructure**: the optimal solution to the problem contains optimal solutions to the subproblems.
2. A **recurrence** for the overall optimal solution in terms of optimal solutions to appropriate subproblems. The recurrence should provide a natural ordering of the subproblems from smaller to larger and require polynomial work for combining solutions to the subproblems.
3. **Iterative, bottom-up** computation of subproblems, from smaller to larger.
4. Small number of subproblems (polynomial in  $n$ ).

# Dynamic programming vs Divide & Conquer

- ▶ They both combine solutions to subproblems to generate the overall solution.
- ▶ However, divide and conquer starts with a large problem and divides it into small pieces.
- ▶ While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.

# Today

## 1 Recap

- A Dynamic Programming solution

## 2 Sequence alignment



# String similarity

This problem arises when comparing **strings**.

**Example:** consider an online dictionary.

- ▶ **Input:** a word, e.g., “ocurrance”
- ▶ **Output:** *did you mean* “occurrence” ?

**Similarity:** intuitively, two words are similar if we can “almost” line them up by using **gaps** and **mismatches**.

# Aligning strings using gaps and mismatches

We can align “ocurrence” and “occurrence” using

- ▶ one gap and one mismatch

o	c	—	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

- ▶ or, three gaps

o	—	c	u	r	r	—	a	n	c	e
o	c	c	u	r	r	e	—	n	c	e

# Strings in biology

- ▶ Similarity of english words is rather intuitive.
- ▶ Determining similarity of biological strings is a central computational problem for molecular biologists.
  - ▶ Chromosomes again: an organism's genome (set of genetic material) consists of chromosomes (giant linear DNA molecules)
  - ▶ We may think of a chromosome as an enormous linear tape containing a string over the alphabet  $\{A, C, G, T\}$ .
  - ▶ The string encodes instructions for building protein molecules.

# Why similarity?

*Why are we interested in similarity of biological strings?*

- ▶ Roughly speaking, the sequence of symbols in an organism's genome determines the properties of the organism.
- ▶ So similarity can guide decisions about biological experiments.

*How do we define similarity between two strings?*

# Similarity based on the notion of “lining up” two strings

Informally, an **alignment** between two strings tells us which pairs of positions will be lined up with one another.

Example:  $X = \text{GCAT}$ ,  $Y = \text{CATG}$

$x_1$	$x_2$	$x_3$	$x_4$	
G	C	A	T	-
-	C	A	T	G
	$y_1$	$y_2$	$y_3$	$y_4$

The set of pairs  $\{(2, 1), (3, 2), (4, 3)\}$  is an **alignment** of  $X, Y$ : these are the pairs of positions in  $X, Y$  that are **matched**.

# Definition of alignment of two strings

An **alignment**  $L$  of  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  is a set of **ordered** pairs of indices  $(i, j)$  with  $i \in [1, m]$ ,  $j \in [1, n]$  such that the following two properties hold:

- P1. every  $i \in [1, m]$ ,  $j \in [1, n]$  appears at most once in  $L$ ;
- P2. pairs do not *cross*: if  $(i, j), (i', j') \in L$  and  $i < i'$ , then  $j < j'$ .

Example:  $X = \text{GCAT}$ ,  $Y = \text{CATG}$

$x_1$	$x_2$	$x_3$	$x_4$	
G	C	A	T	-
-	C	A	T	G
	$y_1$	$y_2$	$y_3$	$y_4$

1.  $\{(2, 1), (3, 2), (4, 3)\}$  is an alignment; but
2.  $\{(2, 1), (3, 2), (4, 3), (1, 4)\}$  is **not** an alignment (violates P2).

# Cost of an alignment

Let  $L$  be an alignment of  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$ .

1. **Gap penalty**  $\delta$ : there is a cost  $\delta$  for every position of  $X$  and every position of  $Y$  that is not matched.
2. **Mismatch cost**: there is a cost  $\alpha_{pq}$  for every pair of alphabet symbols  $p, q$  that are matched in  $L$ .
  - ▶ So every pair  $(i, j) \in L$  incurs a cost of  $\alpha_{x_i y_j}$ .
  - ▶ **Assumption:**  $\alpha_{pp} = 0$  for every symbol  $p$  (matching a symbol with itself incurs no cost).

The **cost** of alignment  $L$  is the sum of all the gap and the mismatch costs.

# Cost of alignment in symbols

In symbols, given alignment  $L$ , let

- ▶  $X_i^L = 1$  **iff** position  $i$  of  $X$  is not matched (gap),
- ▶  $Y_j^L = 1$  **iff** position  $j$  of  $Y$  is not matched (gap).

Then the cost of alignment  $L$  is given by

$$\text{cost}(L) = \sum_{1 \leq i \leq m} X_i^L \delta + \sum_{1 \leq j \leq n} Y_j^L \delta + \sum_{(i,j) \in L} \alpha_{x_i y_j}$$



# Examples

## Example 1.

Let  $L_1$  be the alignment shown below.

$x_1$	$x_2$		$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$
o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$

# Examples

## Example 1.

Let  $L_1$  be the alignment shown below.

$x_1$	$x_2$		$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$
o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$

$$L_1 = \{(1, 1), (2, 2), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10)\}$$

$$\text{cost}(L_1) = \delta + \alpha_{ae} \quad (\text{This is } Y_3^{L_1} + \alpha_{x_6 y_7}.)$$

# Examples

## Example 2.

Let  $L_2$  be the alignment shown below.

$x_1$		$x_2$	$x_3$	$x_4$	$x_5$		$x_6$	$x_7$	$x_8$	$x_9$
o	-	c	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$		$y_8$	$y_9$	$y_{10}$

# Examples

## Example 2.

Let  $L_2$  be the alignment shown below.

$x_1$		$x_2$	$x_3$	$x_4$	$x_5$		$x_6$	$x_7$	$x_8$	$x_9$
o	-	c	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$		$y_8$	$y_9$	$y_{10}$

$$L_1 = \{(1, 1), (2, 3), (3, 4), (4, 5), (5, 6), (7, 8), (8, 9), (9, 10)\}$$

$$\text{cost}(L_2) = 3\delta \quad (\text{This is } X_6^{L_2} + Y_2^{L_2} + Y_7^{L_2}.)$$

# Examples

## Example 3.

Let  $L_3, L_4$  be the alignments shown below.

$x_1$	$x_2$	$x_3$	$x_4$
G	C	A	T
C	A	T	G
$y_1$	$y_2$	$y_3$	$y_4$

$x_1$	$x_2$	$x_3$	$x_4$	
G	C	A	T	-
-	C	A	T	G
	$y_1$	$y_2$	$y_3$	$y_4$

# Examples

## Example 3.

Let  $L_3, L_4$  be the alignments shown below.

$x_1$	$x_2$	$x_3$	$x_4$
G	C	A	T
C	A	T	G
$y_1$	$y_2$	$y_3$	$y_4$

$$L_3 = \{(1, 1), (2, 2), (3, 3), (4, 4)\}$$

$$\text{cost}(L_3) = \alpha_{GC} + \alpha_{CA} + \alpha_{AT} + \alpha_{TG}$$

$x_1$	$x_2$	$x_3$	$x_4$	
G	C	A	T	-
-	C	A	T	G
	$y_1$	$y_2$	$y_3$	$y_4$

$$L_4 = \{(2, 1), (3, 2), (4, 3)\}$$

$$\text{cost}(L_4) = 2\delta$$

# The sequence alignment problem

## Input:

- ▶ **two** strings  $X, Y$  consisting of  $m, n$  symbols respectively; each symbol is from some alphabet  $\Sigma$
- ▶ the gap penalty  $\delta$
- ▶ the mismatch costs  $\{\alpha_{pq}\}$  for every pair  $(p, q) \in \Sigma^2$

**Output:** the **minimum** cost to align  $X$  and  $Y$ , and an optimal alignment.

# Towards a recursive solution

## Claim 1.

*Let  $L$  be the optimal alignment. Then either*

- 1. the last two symbols  $x_m, y_n$  of  $X, Y$  are matched in  $L$ , hence the pair  $(m, n) \in L$ ; or*
- 2.  $x_m, y_n$  are not matched in  $L$ , hence  $(m, n) \notin L$ .  
In this case, at least one of  $x_m, y_n$  is not matched in  $L$ , hence at least one of  $m, n$  does not appear in  $L$ .*



# Proof of Claim 1

By contradiction.

Suppose  $(m, n) \notin L$  but  $x_m$  and  $y_n$  are **both** matched in  $L$ .  
That is,

1.  $x_m$  is matched with  $y_j$  for some  $j < n$ , hence  $(m, j) \in L$ ;
2.  $y_n$  is matched with  $x_i$  for some  $i < m$ , hence  $(i, n) \in L$ .

Since pairs  $(i, n)$  and  $(m, j)$  cross,  $L$  is not an alignment.

# Rewriting Claim 1

The following equivalent way of stating Claim 1 will allow us to easily derive a recurrence.

## Fact 4.

*In an optimal alignment  $L$ , at least one of the following is true*

1.  $(m, n) \in L$ ; or
2.  $x_m$  is not matched; or
3.  $y_n$  is not matched.

# The subproblems for sequence alignment

Let

$OPT(i, j)$  = **minimum cost** of an alignment between  $x_1 \dots x_i, y_1 \dots y_j$

We want  $OPT(m, n)$ . From Fact 4,

1. If  $(m, n) \in L$ , we pay  $\alpha_{x_m y_n} + OPT(m - 1, n - 1)$ .
2. If  $x_m$  is not matched, we pay  $\delta + OPT(m - 1, n)$ .
3. If  $y_n$  is not matched, we pay  $\delta + OPT(m, n - 1)$ .

*How do we decide which of the three to use for  $OPT(m, n)$ ?*

# The recurrence for the sequence alignment problem

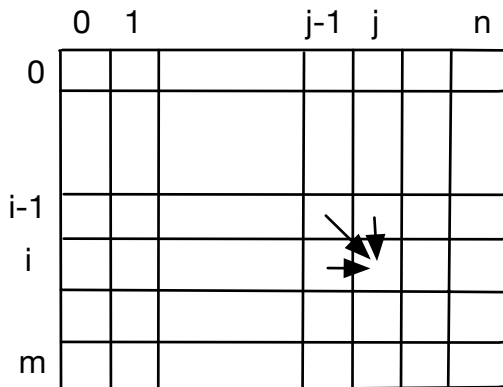
$$OPT(i, j) = \begin{cases} j\delta & , \text{ if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & , \text{ if } i, j \geq 1 \\ i\delta & , \text{ if } j = 0 \end{cases}$$

## Remarks

- ▶ Boundary cases:  $OPT(0, j) = j\delta$  and  $OPT(i, 0) = i\delta$ .
- ▶ Pair  $(i, j)$  appears in the optimal alignment for subproblem  $x_1 \dots x_i, y_1 \dots y_j$  if and only if the minimum is achieved by the first of the three values inside the min computation.

# Computing the cost of the optimal alignment

- ▶  $M$  is an  $(m + 1) \times (n + 1)$  dynamic programming table.
- ▶ Fill in  $M$  so that all subproblems needed for entry  $M[i, j]$  have already been computed when we compute  $M[i, j]$  (e.g., column-by-column).



# Pseudocode

SequenceAlignment( $X, Y$ )

Initialize  $M[i, 0]$  to  $i\delta$

Initialize  $M[0, j]$  to  $j\delta$

**for**  $j = 1$  to  $n$  **do****for**  $i = 1$  to  $m$  **do**
$$M[i, j] = \min \left\{ \alpha_{x_i y_j} + M[i-1, j-1], \right. \\ \left. \delta + M[i-1, j], \delta + M[i, j-1] \right\}$$

end for

end for

```

return  $M[m, n]$ 

```

## Running time?

# Reconstructing the optimal alignment

Given  $M$ , we can reconstruct the optimal alignment as follows.

**TraceAlignment**( $i, j$ )

**if**  $i == 0$  or  $j == 0$  **then** return

**else**

**if**  $M[i, j] == \alpha_{x_i y_j} + M[i - 1, j - 1]$  **then**

**TraceAlignment**( $i - 1, j - 1$ )

            Output ( $i, j$ ),

**else**

**if**  $M[i, j] == \delta + M[i - 1, j]$  **then** **TraceAlignment**( $i - 1, j$ )

**else** **TraceAlignment**( $i, j - 1$ )

**end if**

**end if**

**end if**

Initial call: **TraceAlignment**( $m, n$ )

Running time?

# Resources used by dynamic programming algorithm

- ▶ Time:  $O(mn)$
- ▶ Space:  $O(mn)$ 
  - ▶ English words:  $m, n \leq 10$
  - ▶ Computational biology:  $m = n = 100000$ 
    - ▶ Time: 10 billion ops
    - ▶ Space: 10GB table!
- ▶ *Can we avoid using quadratic space while maintaining quadratic running time?*



## Using only $O(m + n)$ space

1. First, suppose we are only interested in the **cost** of the optimal alignment.

Easy: keep a table  $M$  with 2 columns, hence  $2(m + 1)$  entries.

2. *What if we want the optimal alignment too?*
  - No longer possible in  $O(n + m)$  time.