

Analysis of Algorithms, I

CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

The dynamic programming principle; segmented least squares

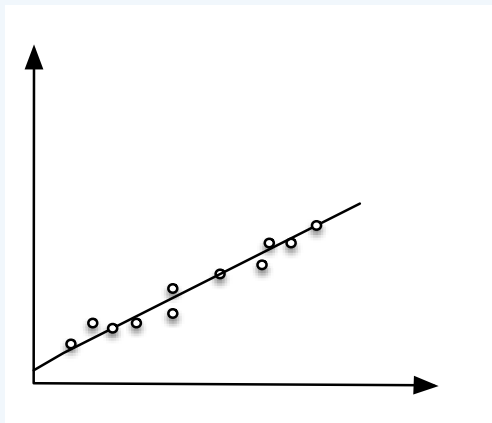
- 1 Segmented least squares
 - An exponential recursive algorithm
- 2 A Dynamic Programming (DP) solution
 - A quadratic iterative algorithm
 - Applying the DP principle

Today

- 1 Segmented least squares
 - An exponential recursive algorithm
- 2 A Dynamic Programming (DP) solution
 - A quadratic iterative algorithm
 - Applying the DP principle

Linear least squares fitting

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



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 error

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

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

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

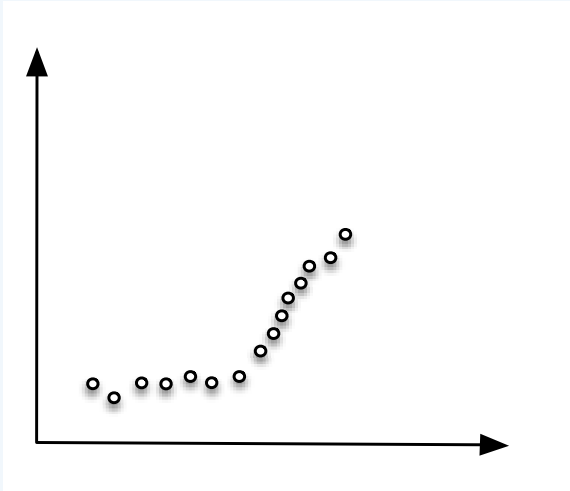
satisfies

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

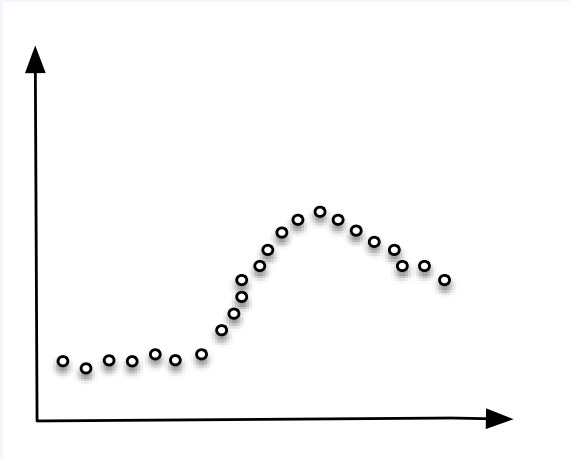
$$b = \frac{\sum_i y_i - a \sum_i x_i}{n} \quad (4)$$

How fast can we compute a, b ?

What if the data changes direction?



What if the data changes direction more than once?



How to detect change in the data

- ▶ Any single line would have large error.
- ▶ **Idea 1:** hardcode number of lines to 2 (or some *fixed* m).
 - ▶ Fails for the dataset on the last slide.
- ▶ **Idea 2:** pass an *arbitrary set* of lines through the points and seek the set of lines that minimizes the error.
 - ▶ Trivial solution: have a different line pass through each pair of consecutive points in P .
- ▶ **Idea 3:** fit the points well, using as few lines as possible.
 - ▶ Trade-off between complexity and error of the model

Formalizing the problem

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 the input.
- ▶ 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

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

Segmented least squares

This problem is an instance of change detection in data mining and statistics.

Input: A set P of n data points $p_i = (x_i, y_i)$ as before.

Output: A segmentation $\mathcal{A}^* = \{S_1, S_2, \dots, S_{m_{\mathcal{A}^*}}\}$ of P whose **cost**

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

is minimum.

A brute force approach

We can find the optimal partition (that is, the one incurring the minimum cost) by exhaustive search.

- ▶ Enumerate every possible partition (segmentation) and compute its cost.
- ▶ Output the one that incurs the minimum cost.

△ $\Omega(2^n)$ partitions

A crucial observation regarding the last data point

Consider the last point p_n in the data set.

- ▶ p_n belongs to a single segment in the **optimal** partition.
- ▶ That segment starts at an earlier point p_i , for some $1 \leq i \leq n$.

This suggests a **recursive** solution: **if** we knew where the last segment starts, then we could remove it and recursively solve the problem on the remaining points $\{p_1, \dots, p_{i-1}\}$.

A recursive approach

- ▶ Let $OPT(j)$ = minimum cost of a partition of the points p_1, \dots, p_j .
- ▶ Then, if the last segment of the optimal partition is $\{p_i, \dots, p_n\}$, the cost of the optimal solution is

$$OPT(n) = err(L, \{p_i, \dots, p_n\}) + C + OPT(i - 1).$$

- ▶ But we don't know where the last segment starts! *How do we find the point p_i ?*
- ▶ Set

$$OPT(n) = \min_{1 \leq i \leq n} \left\{ err(L, \{p_i, \dots, p_n\}) + C + OPT(i - 1) \right\}.$$

A recurrence for the optimal solution

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

Then

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

If we apply the above expression recursively to remove the last segment, we obtain the recurrence

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

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!).

Exponential-time recursion

Notation: $T(n)$ = time to compute $OPT(n)$, that is, the cost of the optimal partition for n points.

Then

$$T(n) \geq T(n-1) + T(n-2).$$

- ▶ Can show that $T(n) \geq F_n$, the n -th Fibonacci number (by strong induction on n).
 - ▶ From optional problem 6a in Homework 1, $F_n = \Omega(2^{n/2})$.
 - ▶ Hence $T(n) = \Omega(2^{n/2})$.
- \Rightarrow The recursive algorithm requires $\Omega(2^{n/2})$ time.

Today

- 1 Segmented least squares
 - An exponential recursive algorithm
- 2 A Dynamic Programming (DP) solution
 - A quadratic iterative algorithm
 - Applying the DP principle

Are we really that far from an efficient solution?

Recall Fibonacci problem from HW1: exponential recursive algorithm, **polynomial** iterative solution

How?

1. **Overlapping subproblems:** spectacular redundancy in computations of recursion tree
2. **Easy-to-compute recurrence** for combining the smaller subproblems: $F_n = F_{n-1} + F_{n-2}$
3. **Iterative, bottom-up computations:** we computed and stored the subproblems from smallest (F_0, F_1) to largest (F_n) , iteratively.
4. **Small number of subproblems:** only solved $n - 1$ subproblems.

Elements of DP in segmented least squares

Our problem exhibits similar properties.

1. **Overlapping subproblems**
2. **Easy-to-compute recurrence** for combining optimal solutions to smaller subproblems into the optimal solution of a larger subproblem (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, with the subproblem of size 0 being the smallest and the subproblem of size n the largest.
- ⇒ There are $n + 1$ subproblems in total. Solving the j -th subproblem requires $\Theta(j) = O(n)$ time.
- ⇒ The overall running time is $O(n^2)$.
- ▶ Boundary conditions: $OPT(0) = 0$.
- ▶ Segment p_k, \dots, p_j appears in the optimal solution only if the minimum in the expression above is achieved for $i = k$.

An iterative algorithm for segmented least squares

Let M be an array of 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 $1 \leq i \leq j$ 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.