Objectives

Dynamic Programming

- Segmented Least Squares
- Subset Sums/Knapsack

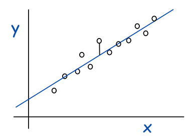
Least Squares

Foundational problem in statistic and numerical analysis

Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Find a line $y = ax + b$ that minimizes the sum of the squared error

- "line of best fit"



Sum of squared error

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

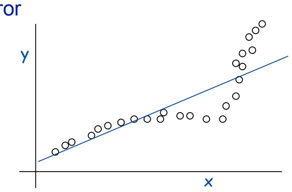
Closed form solution. Calculus $\Rightarrow$ min error is achieved when

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

Least Squares

What happens to the error if we try to fit one line to these points?

- Large error



Pattern: More like 3 lines

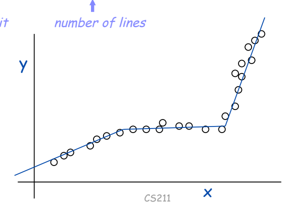
Segmented Least Squares

Points lie roughly on a **sequence** of line segments

Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes f(x)

Q. What's a reasonable choice for $f(x)$ to balance *accuracy* and *parsimony*?

goodness of fit — number of lines



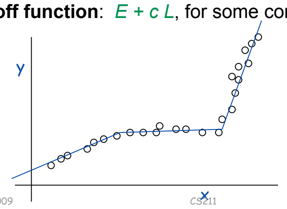
Segmented Least Squares

Points lie roughly on a **sequence** of several line segments.

Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:

- the sum of the sums of the squared errors E in each segment
- the number of lines L

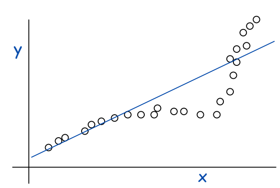
Tradeoff function: $E + c\,L$, for some constant $c > 0$.



How should we define an optimal solution?

Segmented Least Squares

What made it seem like the points were in 3 lines? What happened?



Looking for *change* in linear approximation

- Where to partition points into line segments

Recall: Properties of Problems for DP

Polynomial number of subproblems

Solution to original problem can be easily computed from solutions to subproblems

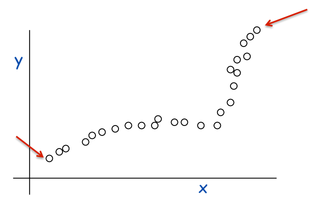
Natural ordering of subproblems, easy to compute recurrence

We need to:
- Figure out how to break the problem into subproblems
- Figure out how to compute solution from subproblems
- Define the recurrence relation between the problems

Toward a Solution

Consider just the first or last point

- What do we know about those points/their segments/cost of segments?

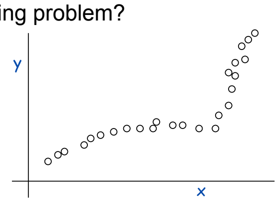


Toward a Solution

p_n can only belong to one segment

- Segment: $p_i, \ldots, p_n$
- Cost: c (cost for segment) + error of segment

What is the remaining problem?



Toward a Solution

p_n can only belong to one segment

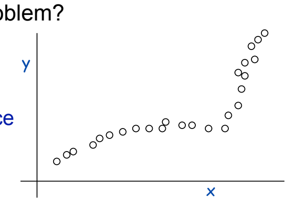
- Segment: $p_i, \ldots, p_n$
- Cost: c (cost for segment) + error of segment

What is the remaining problem?

- Solve for $p_1, \ldots, p_{i-1}$

Goal:

- Formulate as a recurrence



Dynamic Programming: Multiway Choice

Notation.

- **OPT(j)** = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- **e(i, j)** = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

How do we compute OPT(j)?

- Last problem: binary decision (include job or not)
- This time: multiway decision
  - Which option do we choose?

Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- e(i, j) = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute OPT(j):

- Last segment contains points $p_i, p_{i+1}, \ldots, p_j$ for some i
- Cost = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares()
   M[0] = 0
   e[0][0] = 0
   for j = 1 to n
      for i = 1 to j
         e[i][j] = least square error for the
                   segment pi, …, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (e[i][j] + c + M[i-1])

   return M[n]
```

Costs?

Segmented Least Squares: Algorithm Analysis

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares()
   M[0] = 0
   e[0][0] = 0
   for j = 1 to n
      for i = 1 to j
         e[i][j] = least square error for the
                   segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (e[i][j] + c + M[i-1])

   return M[n]
```

$O(n^3)$

can be improved to $O(n^2)$ by pre-computing various statistics

$O(n^2)$

Bottleneck: computing e(i, j) for $O(n^2)$ pairs, O(n) per pair using previous formula

KNAPSACK PROBLEM

The Price is Right

Or, shopping with someone else's money

Goal: Spend as much money as possible without going over $100

- CD $18
- Jeans $40
- DVD $35
- Dinner $15
- Book $8
- Ice cream $5
- Shoes $61
- Pizza $7

Possible solutions?

Knapsack Problem

Given *n* objects and a "knapsack"

Item *i* weighs $w_i > 0$ kilograms and has value $v_i > 0$

- Could be jobs that require w_i time

Knapsack has capacity of *W* kilograms

- *W* is time interval that resource is available

Goal: fill knapsack so as to maximize total value

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Towards a Recurrence…

What do we know about the knapsack with respect to item *i*?

Towards a Recurrence…

What do we know about the knapsack with respect to item *i*?

- Either select item *i* or not
- If don't select
  - Pick optimum solution of remaining items
- Otherwise
  - What happens?
  - How does problem change?

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i

- Case 1: OPT does not select item i
  - OPT selects best of { 1, 2, …, i-1 }
- Case 2: OPT selects item i
  - Accepting item *i* does not immediately imply that we will have to reject other items
    - No known conflicts
  - Without knowing what other items were selected before *i*, we don't even know if we have enough room for *i*

➡Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w

- Case 1: OPT does not select item *i*
  - OPT selects best of { 1, 2, …, i-1 } using weight limit *w*
- Case 2: OPT selects item *i*
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Fill up an n-by-W array

```
Input: N, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to N  # for all items
   for w = 1 to W  # for possible weights
      if w_i > w  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], v_i + M[i-1, w-w_i] }

return M[n, W]
```


Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0											
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

N = 1

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

W + 1

N=2

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

W + 1

N=3

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

W + 1

N=4

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0											

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

W + 1

N=5

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT:
Value=

W = 11

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

OPT: { 4, 3 }
Value = 22 + 18 = 40

W = 11

Analyzing Solution

Costs?

```
Input: N, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to N  # for all items
   for w = 1 to W  # for possible weights
      if w_i > w  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], v_i + M[i-1, w-w_i] }

return M[n, W]
```

Analyzing Solution

Costs?

```
Input: N, w1,…,wN, v1,…,vN

for w = 0 to W                                O(W)
   M[0, w] = 0

for i = 1 to N  # for all items                  O(N W)
   for w = 1 to W  # for possible weights
      if wi > w  # item's weight is more than available
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max{ M[i-1, w], vi + M[i-1, w-wi] }

return M[n, W]
```


Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$

- **Not** polynomial in input size!
- "Pseudo-polynomial"
  - Reasonably efficient when W is reasonably small
- Decision version of Knapsack is NP-complete [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]