CSCE 629 Analysis of Algorithms Project 1

Main idea :-

Line Fitting basics :-

Set P :- n points $\Rightarrow (x_1, y_1) \cdots (x_n, y_n)$

Suppose $x_1 < x_2 < \cdots \cdots < x_n$

Line L :- $y = ax + b$

Error of line L with respect to P :-

$\Rightarrow$ Sum of squared distances to the point P

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

$\Downarrow$

Gives better approximation of how well line L fits P points.

Less error $\Rightarrow$ better fit

If $n \geq 2$,

Given set P points

The line with minimal error Error(L, P)

is $y = ax + b$

$$a = \frac{n \sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$b = \frac{\sum_{i=1}^{n} y_i - a \sum_{i=1}^{n} x_i}{n}$$

Multi line fitting problem :-

Here, we fit points using multiple lines.

→ not simple to know the number of lines that are best choice beforehand.

solve it as a modelling problem :-

- model a set of "observed points" using a simple & accurate model

Here :-

model ⇒ set of lines each with different x interval

Input :-

① Set of n-points in a 2-D plane.

$$P = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$$

$$x_1 \leq x_2 < \cdots < x_n$$

$$P_i = (x_i, y_i) \quad \text{for } i = 1 \text{ to } n$$

② Real number $C > 0$

Output :-

Partition of points P into k segments :-

$$P_1 = \{P_1, P_2 \cdots, P_{m_1}\}$$

$$P_2 = \{P_{m_1+1}, P_{m_1+2} \cdots, P_{m_2}\}$$

$$P_3 = \{P_{m_2+1}, P_{m_2+2}, \cdots, P_{m_3}\}$$

⋮

$P_k = \{ P_{m_{k-1}+1}, \ldots \ldots P_{m_k} = P_n \}$

↓

these minimize the cost function

$$\text{cost}(P, C) = \underbrace{E(P_1) + E(P_2) + \cdots + E(P_k)}_{\text{total error using } k \text{ lines to fit } n\text{-points}} + \underbrace{C_k}_{\text{penalty for using } k \text{ lines}}$$

The value of k & turning points $(P_{m_1}, P_{m_2} \cdots P_{m_{k-1}})$ are needed to be decided by the algorithm

Idea behind my algorithm:-

I have solved this problem using dynamic programming.

I took the inspiration from the rod cutting problem.

In the given problem, we need to segregate the points in such a way that, the number of lines chosen and the cost has to be optimal.

In the rod cutting problem we go for a solution that gives maximum price. But, in this problem we should go for a solution that gives minimal cost.

the optimal cost is given by

d[i] => cost till ith point

$$dp[i] = \min_{2 \le k \le i} \begin{cases} \text{if } dp[i] > 0 \\ \quad dp[i-k] + cost[i-k+1][i] \\ \text{else} \\ \quad cost[0][i] \end{cases}$$

because we cannot abandon the last point, we must always include it

Here k is the number of steps we are going back from the current dp point to get the cost.

In order to get the k list, we keep track of break points every time we find a better cost dp from the point that we are standing and add it to the breakpoints of the current dp's corresponding index.

K values are stored such that

$$K[i] = [i] + \begin{cases} \text{if :-} \\ (dp[i-k] + cost[i-k+1][i]) < \min \\ \quad K \Rightarrow K[i-k] \\ \text{else} \\ \quad K \Rightarrow [\,] \end{cases}$$

We just have to keep track of the optimal dp's break points and append them to the current dp along with the end point till the current dp

Here $dp[i] \Rightarrow$ minimum cost till i^{th} point
$0 \leq i <$ num points

$K[i] \Rightarrow$ Break points till i^{th} dp point
$0 \leq i <$ numpoints

In this algorithm I calculate cost between two points using an error matrix, which use $O(n^2)$ to get the entire error matrix

$cost[i][j] \Rightarrow$ minimum cost from point i to j.

Pseudo Code :-

Main function to pass the dataset :-

piecewise_dict(dataset)

For i from (0 to len of dataset) :-

```
// read the data set to first record
cur_dataset = dataset[i]
cur_dataset_result = piecewise_linear_segmentation(cur_dataset)
Final_result.append(cur_dataset_result)
return Final_result
```

Calculate result for each record :-

```
Piecewise_linear_segmentation(record)
    load n_list, x_list, y_list, c_list from record
    // error matrix
    errors = linear_regression_error_matrix(n_list, x_list, y_list, c_list)
    // errors[i][j] = cost for segment from ith to jth point

    dp = [0, 0 - - -  0]   // size n_list - 1
    dp[i] = errors[0][i]
    K = [[][] - - - []]   // size n_list - 1
    K[i].append(i)
    For i from 2 to (n_list - 1)
        min_val = MAX_INT
        K[i].append(i)
        temp_list = []
        For k from 2 to i.   // steps to go back
            if (i - k) > 0:
```

```
temp_val = dp[i-k] + errs[i-k+1][i]
if (temp_val < min_val):
    min_val = temp_val
    temp_list = k[i-k]

Else:
    temp_val = errs[0][i]
    if (temp_val < min_val):
        min_val = temp_val
        temp_list = []

dp[i] = min_val          // update dp
k[i] = k[i] + temp_list  // update k
```

```
result['k_list'] = len(k[n_list - 1])
result['last_points_list'] = k[n_list - 1]
result['OPT_list'] = dp[n_list - 1]

Return result.
```

✓ Calculate error matrix :-

```
linear_regressor_error_matrix(n_list, x_list, y_list, c_list)
    error_matrix = np.zeros((n_list, n_list))
// get the cummulative sum arrays // cummulative sum til point
x_sum, y_sum, x_power2_sum, y_power2_sum, xy_dot
    = preprocess_arrays(n_list, x_list, y_list)
For i from 0 to n_list:
    For j from i+1 to n_list:
        n = j - i + 1
        if (j - 1) == 1):
            error_matrix[i][j] = c_list
```

elif(i==0):

X_sum_ij = X_sum[j]
Y_sum_ij = Y_sum[j]

X_pow_2_sum_ij = X_pow_2_sum[j]
Y_pow_2_sum_ij = Y_pow_2_sum[j]
XY_dot_ij = XY_dot[j]

else:-
X_sum_ij = X_sum[j] - X_sum[i-1]
Y_sum_ij = Y_sum[j] - Y_sum[i-1]
X_pow_2_sum_ij = X_pow_2_sum[j] - X_pow_2_sum[i-1]
Y_pow_2_sum_ij = Y_pow_2_sum[j] - Y_pow_2_sum[i-1]
XY_dot_ij = XY_dot[j] - XY_dot[i-1]

$$a = \frac{((n * XY\_dot\_ij) - (X\_sum\_ij * Y\_sum\_ij))}{((n * X\_pow\_2\_sum\_ij) - (X\_sum\_ij) * (X\_sum\_ij))}$$

$$b = (Y\_sum\_ij - a * X\_sum\_ij)/(n)$$

error_matrix[i][j] = absolute value of

$$(Y\_pow\_2\_sum\_ij) + (a^2 * X\_pow\_2\_sum\_ij) + (n * b^2 *) + (2ab * X\_sum\_ij) - (2 * a * XY\_dot\_ij) - (2 * b * Y\_sum\_ij)) + c\_list$$

code to find cummulative sum of variables —

preprocess_arrays (n_list, x_list, y_list):

x_sum[0] = x_list[0], y_sum[0] = y_list[0],
x_pow2_sum[0] = x_list[0] * x_list[0],
y_pow_2_sum[0] = y_list[0] * y_list[0]
xy_dot[0] = x_list[0] * y_list[0]

// Calculate cummulative sum for other points

for i in range(0, n_list): —

x_sum[i] = x_sum[i-1] + x_list[i]
y_sum[i] = y_sum[i-1] + y_list[i]
x_pow2_sum[i] = x_pow2_sum[i-1] + (x_list[i] * x_list[i])
y_pow_2_sum[i] = y_pow2_sum[i-1] + (y_list[i] * y_list[i])
xy_dot[i] = xy_dot[i-1] + (x_list[i] * y_list[i])

return x_sum, y_sum, x_pow2_sum, y_pow2_sum, xy_dot

Proof of correctness

My algorithm is correct because it covers various cases in which no single point is abandoned & always considers two or more points in a segment.

In my algorithm, I collect and save the cummulative cost till a point 'i' in an array called dp.

dp[0] => cost till 0th point but this considered because single point can't be abandoned.

dp[i] => cost [0 -> i]

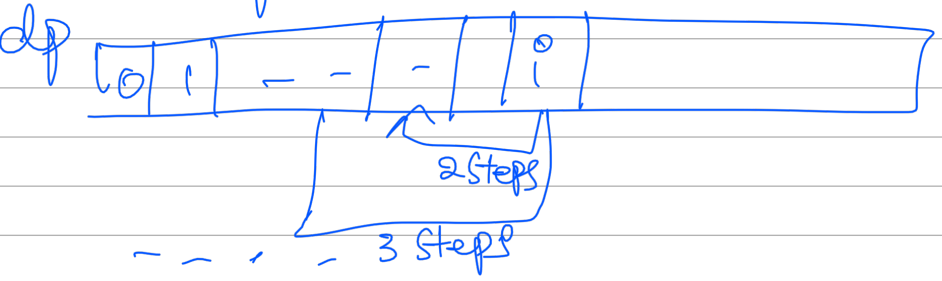
whenever we encounter dp[0] or dp[1]

↓

we always calculate the cost as error(0 → i)

as we cannot abandon the 0th point.

When we are at point 'i'



we always take 2 or more steps back to find the optimal cost.

If we take just one step:-

1 step

it will be dp[i-1] + cost(i)

↓

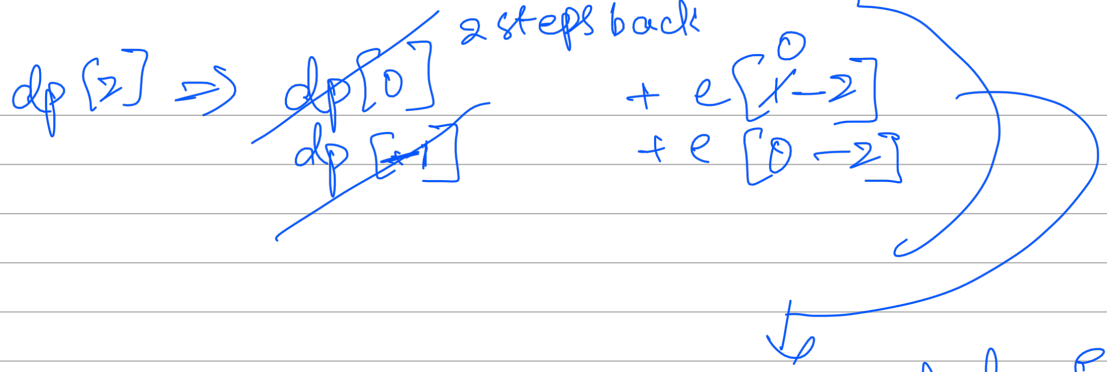
single point

So we do not consider the case with single step.

For every iteration we check for all possible cases & find the least

dp[0] ⇒ NA

dp[1] ⇒ e[0→1]



as we encounted 0 & -1 in dp index we only cosider $e[0 \to i]$ whenever we encounter dp[0] & dp[-1]

$$dp[3] = \min \begin{cases} dp[1] + e[2 \to 3] & \text{2 step} \\ dp[0] + e[x \to 3] & \text{3 steps} \\ dp[-1] + e[0 \to 3] & \text{4 steps} \end{cases}$$

$$dp[4] = \min \begin{cases} dp[2] + e[3 \to 4] & \text{2 steps} \\ dp[1] + e[2 \to 4] & \text{3 steps} \\ dp[0] + e[x \to 4] & \text{4 steps} \\ dp[-1] + e[0 \to 4] & \text{5 steps} \end{cases}$$

Every time we update the minimum, we update 'x' value corresponding to the considered dp that gives the minimum

As my algorithm covers all possible cases when solving a problem such as multiline fitting I can say that this is an optimal approach with the best time complexity $O(n^2)$ and this approach also involves dynamic programming.

Time Complexity analysis

The time complexity for my algorithm is $O(n^2)$ for a dataset record (single set of points)

The following is the flow for my algorithm :-

Calculate the result for a given data set record

```
In [5]: def piecewise_linear_segmentation(record):
            n_list = record[0]
            x_list = record[1]
            y_list = record[2]
            c_list = record[3]

            #calculate and store errors for the possible ranges
            errors = linear_regressor_error_matrix(n_list,x_list,y_list,c_list)

            # Lets Calculate (k_list,last_points_list, OPT_list)
            # k_list = number of segments
            # last_points_list = list of last points in each segment
            # OPT_list is the cost function value

            # Lets create a list dp to store the cost till a point (this stores the OPT_list va
            # dp[i] means cost till ith point

            dp = [0 for i in range(n_list)]
            dp[1] = errors[0][1]

            # Lets track K value for every i value in DP (this stores the last_points_list in K
            K = [[] for i in range(n_list) ]
            K[1].append(1)

            # run for ith point cost in dp
            for i in range(2,n_list):

                #store min for this dp point
                min_val = sys.maxsize
                K[i].append(i)
                temp_list = []
                #number of steps we are going back to check the cost
                for k in range(2,i+1):
                    if ((i-k) > 0):
                        temp_val = dp[i-k] + errors[i-k+1][i]
                        # now lets check if temp_val is less than min_val; if yes lets save it'
                        if(temp_val < min_val):
                            min_val = temp_val
                            temp_list = K[i-k]
            #             min_val = min(temp_val,min_val)
                    else:
                        temp_val = errors[0][i]
                        if(temp_val < min_val):
                            min_val = temp_val
                            temp_list = []
            #             min_val = min(temp_val,min_val)
            #             temp_list = []

                dp[i] = min_val
                K[i] = K[i]+temp_list

            #     print("K list is")
            #     print(K[n_list-1])
            #     print("Optimum value is")
            #     print(dp[n_list-1])
            #     print("Segments count is")
            #     print(len(K[n_list-1]))
            result = {}
            result['k_list'] = len(K[n_list-1])
            result['last_points_list'] = K[n_list-1]
            result['OPT_list'] = dp[n_list-1]
            return result
```

Total $O(n^2)$

Calculate Error Matrix

In [4]:
```
def linear_regressor_error_matrix(n_list,x_list,y_list,c_list):

    #create an error matrix for the given points

    #error_matrix[i][j] => error from point i to j segment
    #i from 0 to n_list-1 and j from 0 to n_list-1

    error_matrix = np.zeros((n_list, n_list))

    # let's the cummulative sum parameters

    X_sum,Y_sum,X_pow_2_sum,Y_pow_2_sum,XY_dot = preprocess_arrays(n_list,x_list,y_list)

    for i in range(0,n_list):
        for j in range(i+1,n_list):

            #if i and j are adjacent point, no need to calculate error, as it is going to be 0, just add penal
            if(j-i == 1):
                error_matrix[i][j] = c_list

            #let's write the if condition where it checks if 0th point is involved
            #here we can directly use cummulative sum till jth point
            elif(i==0):
                #total points n
                n = j+1
                #calc individual params
                X_sum_ij = X_sum[j]
                Y_sum_ij = Y_sum[j]
                X_pow_2_sum_ij = X_pow_2_sum[j]
                Y_pow_2_sum_ij = Y_pow_2_sum[j]
                XY_dot_ij = XY_dot[j]
                a=((n*XY_dot_ij)-(X_sum_ij*Y_sum_ij))/((n*X_pow_2_sum_ij)-((X_sum_ij)*(X_sum_ij)))
                b=(Y_sum_ij-a*X_sum_ij)/(n)
                error_matrix[i][j]=abs((Y_pow_2_sum_ij)+((a*a)*X_pow_2_sum_ij)+(n*(b*b))+(2*a*b*X_sum_ij)-(2*a
            else:
                n = j-i+1
                #calc individual params
                X_sum_ij = X_sum[j] - X_sum[i-1]
                Y_sum_ij = Y_sum[j] - Y_sum[i-1]
                X_pow_2_sum_ij = X_pow_2_sum[j] - X_pow_2_sum[i-1]
                Y_pow_2_sum_ij = Y_pow_2_sum[j] - Y_pow_2_sum[i-1]
                XY_dot_ij = XY_dot[j] - XY_dot[i-1]
                a=((n*XY_dot_ij)-(X_sum_ij*Y_sum_ij))/((n*X_pow_2_sum_ij)-((X_sum_ij)*(X_sum_ij)))
                b=(Y_sum_ij-a*X_sum_ij)/(n)
                error_matrix[i][j]=abs((Y_pow_2_sum_ij)+((a*a)*X_pow_2_sum_ij)+(n*(b*b))+(2*a*b*X_sum_ij)-(2*a

    return error_matrix
```

Calculate Error Matrix variables

In [3]:
```
def preprocess_arrays(n_list,x_list,y_list):
    #let's find cummulative sum for all the points error calculation variables

    #consider points from 0 to n_list-1 , sum[0] means sum till the point 0


    #create an X sum array with 0's
    X_sum = np.zeros(n_list)
    #create a Y sum array with 0's
    Y_sum = np.zeros(n_list)
    #create an array for sigma X**2 with 0's
    X_pow_2_sum = np.zeros(n_list)
    #create an array for sigma Y**2 with 0's
    Y_pow_2_sum = np.zeros(n_list)
    #XY dot product _ sum
    XY_dot = np.zeros(n_list)

    #let's calculate the cummulative sum for all these variables in a single for loop

    # let's calculate it for 0
    X_sum[0] = x_list[0]
    Y_sum[0] = y_list[0]
    X_pow_2_sum[0] = x_list[0]*x_list[0]
    Y_pow_2_sum[0] = y_list[0]*y_list[0]
    XY_dot[0] = x_list[0]*y_list[0]

    #calc cummulative sum for remaining points
    for i in range(0,n_list):
        X_sum[i] = X_sum[i-1]+x_list[i]
        Y_sum[i] = Y_sum[i-1]+y_list[i]
        X_pow_2_sum[i] = X_pow_2_sum[i-1] + (x_list[i]*x_list[i])
        Y_pow_2_sum[i] = Y_pow_2_sum[i-1] + (y_list[i]*y_list[i])
        XY_dot[i] = XY_dot[i-1] + (x_list[i]*y_list[i])

    return X_sum,Y_sum,X_pow_2_sum,Y_pow_2_sum,XY_dot
```

∴ The total time complexity for my algorithm is $O(n^2)$ for a single dataset record