CSCE 629 Analysis of Algorithms
Project 1

I. Some Basics on Line Fitting

Suppose that there is a set P of n points in the two-dimensional plane, denoted by

$$(x_1, y_1),\ (x_2, y_2),\ \cdots,\ (x_n, y_n),$$

and suppose $x_1 < x_2 < \cdots < x_n$. Given a line L defined by the equation

$$y = ax + b,$$

we say that the error of L with respect to P is the sum of its squared "distances" to the points in P:

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

The above error shows how well the line L approximates the points in P: the smaller the error is, the better it fits.

We show an example in Fig. 1. In this example, there are $n = 7$ points, and a line that optimally fits them is shown.

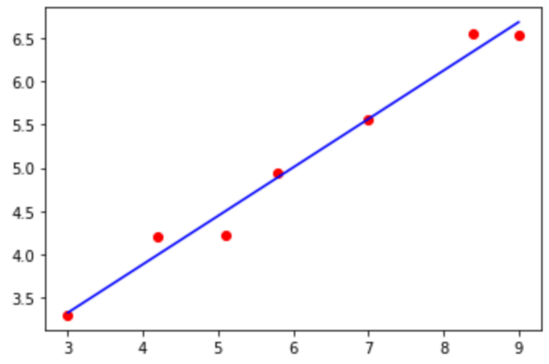

Fig. 1. A line that optimally fits 7 points.

It is known that when $n \geq 2$, given the set of points P, the line L that minimizes the error $\text{Error}(L, P)$ is $y = ax + b$ with

$$a = \frac{n \sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \quad \text{and} \quad b = \frac{\sum_{i=1}^{n} y_i - a \sum_{i=1}^{n} x_i}{n}.$$

If $n = 1$, then any line passing through the only point trivially fits the point perfectly well, and the error would be $\text{Error}(L, P) = 0$.

In the following, given a set of $n \geq 1$ points P, let's use

$$\varepsilon(P)$$

to denote the minimum error that a line fitting them can achieve. (Clearly, we can achieve the minimum error $\varepsilon(P)$ by using the solution above.)

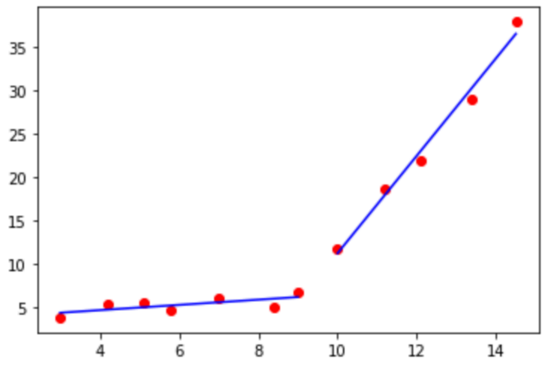

Fig. 2. Two lines that fit 12 points.

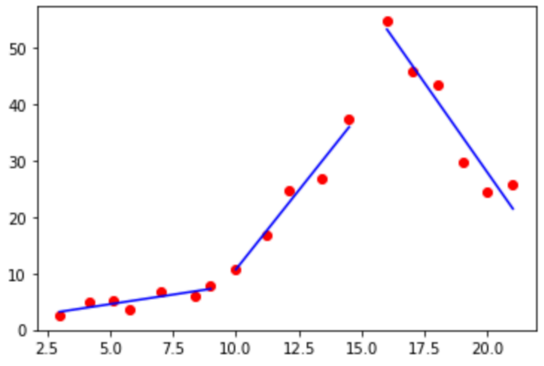

Fig. 3. Three lines that fit 18 points.

II. Multi-Line Fitting Problem

Sometimes it is hard to fit points well using just one line. Instead, it is more natural to fit the points with multiple lines. For example, for the points in Fig. 2, it is more natural to fit them using two lines. And for the points in Fig. 3, it is more natural to fit them using three lines.

It is not always simple, however, to know how many lines are the best choice beforehand. On the other hand, if there is no restriction on how many lines can be used, it would become trivial to fit the points perfectly: just have a different line pass through each pair of consecutive points in P. But that would make the problem meaningless. The problem is essentially a "modeling" problem: we want to model a set of "observed points" using a simple yet accurate model, and in this case, the model is a set of lines, each for a different interval of x. If too many lines are used, the model would not be "simple" any more. Thus, intuitively, we would like a problem formulation that requires us to fit the points well, using as few lines as possible.

Let's formulate our problem, which we shall call the "Multi-Line Fitting Problem":

Input: We have the following inputs:

1) A set of n points in a 2-dimensional plane

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

with $x_1 < x_2 < \cdots < x_n$. We shall use p_i to denote the point (x_i, y_i), for $i = 1, 2, \cdots, n$.

2) A real number $C > 0$.

Output: A partition of the points P into k segments:

$$P_1 = \{p_1, p_2, \cdots, p_{m_1}\}$$

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

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

$$\vdots$$

$$P_k = \{p_{m_{k-1}+1}, p_{m_{k-1}+2}, \cdots, p_{m_k} = p_n\}$$

that minimizes the cost function

$$\text{cost}(P, C) = \varepsilon(P_1) + \varepsilon(P_2) + \cdots + \varepsilon(P_k) + Ck.$$

In the cost function $\text{cost}(P, C)$, the part $\varepsilon(P_1) + \varepsilon(P_2) + \cdots + \varepsilon(P_k)$ is the total error of using k lines to fit the n points, and Ck is the penalty for using k lines (namely, for partitioning the n points into k segments). The greater k is, the greater the penalty term Ck is. Note that the value of

$$k,$$

as well as the turning points

$$p_{m_1}, p_{m_2}, \cdots, p_{m_{k-1}},$$

are what an algorithm solving the problem needs to decide on.

III. What to Submit

In this project, you need to submit three items: (1) an algorithm report, (2) a program that implements your algorithm, (3) output of your program on test instances.

A. Submission Item One: An Algorithm Report

First, you are to design an efficient algorithm for the "Multi-Line Fitting Problem", and submit it as a report. The report is just like what we usually do for algorithm design in a homework, and should have the four elements as usual:

1) The main idea of your algorithm.
2) The pseudo code of your algorithm.
3) The proof of correctness for your algorithm.
4) the analysis of time complexity for your algorithm.

B. Submission Item Two: A Program That Implements Your Algorithm

You need to implement your algorithm using a commonly used programming language, such as Python, C++, Java, etc. We encourage you to use Python if possible, but other languages are fine, too.

Your program will take a list of instances of the "Multi-Line Fitting Problem" as input, and return a list of solutions (corresponding to those instances) as its output.

To test your program, we will provide you with input instances as a "dictionary" in Python (after you have submitted the program), and ask you to submit the output solutions as a "dictionary" in Python. (Details on the two files, including their formats, will be explained in the next subsection for "Submission Item Three".)

When you submit your program, you need to explain clearly how to run your code. If your programming language is not Python, then you also need to include an explanation on how you turned the provided "dictionary" of "input instances" in Python to your programming language, and how you turned your "output solutions" in your programming language to a "dictionary" in Python (following the formats specified in the next subsection).

C. Submission Item Three: Output of Your Program on Test Instances

To test your algorithm/program, we will provide you with three sets of "input instances", all in the Python language:

1) A set of instances of relatively small sizes.
2) A set of instances of medium sizes.
3) A set of instances of relatively large sizes.

Of course, if your algorithm/program is correct, then you should get the correct solutions for all three sets of input instances. However, an algorithm is not only about correctness, but also about efficiency. If your algorithm's time complexity is low, then computing should be easy. (When we tested the algorithm in Google Colab using Python, the set of small instances took less than 1 minute 20 seconds, the set of medium-sized instances took about 14 minutes 30 seconds, and the set of large instances took about 13 minutes 30 seconds.) However, if your algorithm's time complexity is high, you may experience much longer running times, or may not be able to get the solutions to the large instances.

To give you an idea what the three sets of input instances are like, we provide three "example" files (downloadable from our course webpage): "examples_of_small_instances", "examples_of_medium_instances", "examples_of_large_instances". They are all Python dictionaries, whose format will be explained below. You can download them by clicking on the links in our course webpage. After downloading each file, you can open it using pickle.load(open(filePath, 'rb')), where "filePath" is the path of your downloaded file. (Of course, these three files are different from the three files we will send you for testing your program. But they are similar.)

After you run your program on the three sets of "input instances", save your results as three Python "dictionaries" in three files:

1) File "small_solutions" corresponding to the input instances of small sizes.
2) File "medium_solutions" corresponding to the input instances of medium sizes.
3) File "large_solutions" corresponding to the input instances of large sizes.

Now let's explain the formats of the "input instances" and "output solutions".

The "input instances" is a "dictionary" (in Python) that contains 4 key-value pairs (in the following, let's use m to denote the number of provided instances of the "Multi-Line Fitting Problem"):

1) "key" is "n_list", "value" is a list of m numbers. For $i = 0, 1, \cdots, m-1$, $n\_list[i]$ is the number of points in the i-th instance (that are to be fitted by lines). (Note that here and in the following, the points will be indexed by 0, 1, 2, $\cdots$ instead of by 1, 2, 3, $\cdots$.)
2) "key" is " x_list, "value" is a list of m items. For $i = 0, 1, \cdots, m-1$, $x\_list[i]$ is a list of $n\_list[i]$ numbers. For $i = 0, 1, \cdots, m-1$ and $j = 0, 1, \cdots, n\_list[i]-1$, $x[i][j]$ is the x-coordinate of the j-th point in the i-th instance. Here $x[i][0] < x[i][1] < x[i][2] < \cdots < x[i][n\_list[i]-1]$.
3) "key" is " y_list, "value" is a list of m items. For $i = 0, 1, \cdots, m-1$, $y\_list[i]$ is a list of $n\_list[i]$ numbers. For $i = 0, 1, \cdots, m-1$ and $j = 0, 1, \cdots, n\_list[i]-1$, $y[i][j]$ is the y-coordinate of the j-th point in the i-th instance.
4) "key" is C_list, "value" is a list of m numbers. For $i = 0, 1, \cdots, m-1$, $C\_list[i]$ is a positive real number, which is the input parameter C specified in the definition of the "Multi-Line Fitting Problem" for the i-th instance. (That is, for the i-th instance, $C\_list[i]$ is the extra cost for every extra line we use to fit points.)

We can see that in the above dictionary, for $i = 0, 1, \cdots, m-1$, the four items – n_list[i], x_list[i], y_list[i], C_list[i] – describe all the information we need to know about the i-th instance.

The "output solutions" is a "dictionary" (in Python) that contains 3 key-value pairs (as above, let's use m to denote the number of provided instances):

1) "key" is k_list, "value" is a list of m numbers. For $i = 0, 1, \cdots, m-1$, $k\_list[i]$ is the number of lines that the solution to the i-th instance uses to fit the points.
2) "key" is $last\_points\_list$, "value" is a list of m items. For $i = 0, 1, \cdots, m-1$, $last\_points\_list[i]$ is a list of $k\_list[i]$ integers. For $i = 0, 1, \cdots, m-1$ and $j = 0, 1, \cdots, k\_list[i]-1$, $last\_points\_list[i][j]$ is the index (which is between 0 and $n\_list[i]-1$) of the last point (namely, the right-most point) of those points that are fitted by the j-th line in the i-th instance.
3) "key" is OPT_list, "value" is a list of m numbers. For $i = 0, 1, \cdots, m-1$, $OPT\_list[i]$ is the cost of the solution to the i-th instance. (The "cost" is as defined in the definition of the "Multi-Line Fitting Problem", which we try to minimize.)

We give two small "example" files for the "input instances" and "output solutions". Their filenames are "examples_of_instances" and "examples_of_solutions", and you can download them by clicking the links in the course webpage. (As before, after you have downloaded a file, you can use pickle.load(open(filePath, 'rb')) to open it.) The file "examples_of_instances" contains $m = 10$ sample instances, and the file 'examples_of_solutions" contains 10 corresponding solutions. You can run your program on the 10 sample instances, to see if your solutions match the 10 solutions given here. (However, note that optimal solutions may not be unique.) Here is an example:

- Among the 10 instances in "examples_of_instances", Instance 0 has the following details:
  1) n_list[0]=94. It means there are 94 points.
  2) x_list[0] = [0.6134631912663004, 1.0211871365118395, 2.0729039402348004, $\cdots$, 99.28951511207407, 99.65907060262492]. They are the x-coordinates of the 94 points, in a monotonically increasing order.

3) y_list[0] = [-4.710514174087215, 7.686070430601503, 12.710443366535106, $\cdots$, -190.0743023201769, -193.40089091558218]. They are the y-coordinates of the 94 points.
4) C_list[0]= 750. It means that for every extra line we use to fit the points, the extra cost is 750. (Of course, using more lines also decreases the fitting errors.)

Among the 10 solutions in "examples_of_solutions", Solution 0 has the following details:

1) k_list[0]=3. It means the solution uses 3 lines to fit the 94 points in Instance 0.
2) last_points_list[0] = $[31, 44, 93]$. It means the first line fits the points $p_0, p_1, \cdots, p_{31}$, the second line fits the points $p_{32}, p_{33}, \cdots, p_{44}$, the third line fits the points $p_{45}, p_{46}, \cdots, p_{93}$.
3) OPT_list[0] = 9955.201272375221. That is the cost of Solution 0.

The 94 points in Instance 0 and the 3 lines that fit them in Solution 0 are illustrated in Fig. 4.

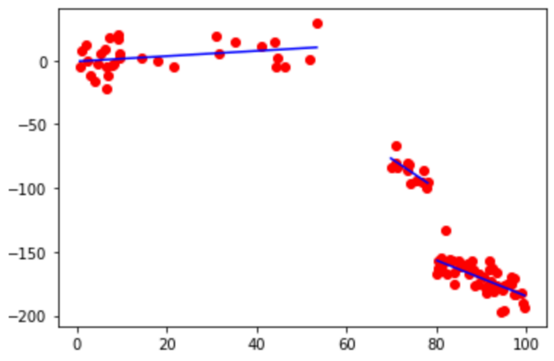

Fig. 4. Instance 0 in "examples_of_instances" and Solution 0 in "examples_of_solutions".

IV. How to Grade Project

The project has a total of 100 points. If ANY of the three items below is not submitted on time, the whole project will receive no point.

Assuming that all the three items are submitted on time, the project will be graded as follows.

A. Submission Item One: An Algorithm Report

The algorithms report has 40 points:

- 20 points for the "main idea" and the "correctness proof". They should be rigorous and clear.
- 5 points for the "pseudo code". It should be correct and clear.
- 15 points for the "time-complexity analysis". It should not only be correct and clear, but should also show that your algorithm achieves low time-complexity. In other words, if your algorithm is correct but does not achieve the lowest time complexity that we require, some or most of these 15 points can be deducted.

B. Submission Item Two: A Program That Implements Your Algorithm

The program has 6 points.

If your program has bugs or does not have a clear interface – which means we have to revise your code or spend time trying to understand how to run your code – some points will be deducted.

C. Submission Item Three: Output of Your Program on Test Instances

After you have submitted your "algorithm report" (submission item one) and "program" (submission item two), we will post three files: test_set_small_instances, test_set_medium_instances, test_set_large_instances. They contain instances of small, medium and large sizes, respectively. You need to run your submitted program (the exact program that you have submitted as Submission Item Two) on the three sets of instances, save your results in three files "small_solutions", "medium_solutions", and "large_solutions" and submit them. Those three files are your "Submission Item Three", and they have 54 points:

1) If all the solutions in "small_solutions" are correct, you receive 18 points. If any of those solutions is incorrect, you receive 0 point.

2) If all the solutions in “medium_solutions” are correct, you receive 18 points. If any of those solutions is incorrect, you receive 0 point.
3) If all the solutions in “large_solutions” are correct, you receive 18 points. If any of those solutions is incorrect, you receive 0 point.

Note that for an algorithm to be correct, it needs to be correct for all instances. So we need to be very rigorous with our designed algorithm and its implementation. That is why in the above, if any solution is incorrect in a file, all the 18 points will be lost. (So there is no partial credit here.)

Please also note that the program you use to generate the solutions here should be exactly the same program that you have submitted as “Submission Item Two”. If any modification is detected (e.g., if we find that some solution submitted here is not the same as what the already-submitted program would produce), the project will receive 0 point.

V. Hint on the Algorithm

Here is a hint on how to solve the problem: Consider dynamic programming. Do computation for points in smaller intervals before computing those things for points in larger intervals.