Optimization for Data Science

F. Rinaldi[1]

[1]  Università degli Studi di Padova | Dipartimento Matematica

Padova
2020

Outline

Optimization for Data Science

Algorithm Analysis

Core of this course

Analyzing algorithms for solving optimization problems in data science.

- Definitions of "problem" and "algorithm" might be formally analyzed by means of the *Touring Machine* concept.
- Development of a rigorous theory for complexity analysis is out of the scope of this course.

Here we will give just an idea of what are the *complexity issues* when dealing with optimization algorithms.

Algorithm

Definition

An *"algorithm"* is a finite executable description of an effective process, i.e. executable sequence of steps/operations that computes some function.

Complexity theory

Complexity theory is related to the analysis of computer algorithms.

The goal of the theory is twofold:

- Developing criteria for measuring the effectiveness of a given algorithm (and thus, comparing it with other algorithms using the same criteria),
- Assessing the hardness of various problems.

Complexity

- The term complexity refers to the amount of resources required by a computation.
- In our context we mainly focus on computing time.
- In complexity theory, however, one is not interested in the execution time of a program implemented in a particular programming language, running on a particular computer over a particular input.
- Instead, one wishes to associate to an algorithm more specific measures.

Complexity II

Roughly speaking, to do so we need to define:

- a notion of input size,
- a set of basic operations,
- a cost for each basic operation.

The last two allow one to associate a cost of a computation:

- x is any input
- $C(x)$ is the cost of the computation with input x (sum of the costs of all the basic operations performed during this computation);

Basic Definitions

We call

- $\mathcal{A}$ an algorithm that we want to analyze;
- $\mathcal{I}_n$ the set of all inputs having same size n;
- The *worst-case cost function* of $\mathcal{A}$ for an input size n is a function defined as follows:
$$U_w^{\mathcal{A}}(n) = \sup_{x \in \mathcal{I}_n} C(x);$$
- If some probability is involved in generation of $\mathcal{I}_n$ we might want to use the so-called *average-case cost function* of $\mathcal{A}$ for an input size n is a function defined as follows:
$$U_a^{\mathcal{A}}(n) = E_n(C(x)),$$
with E_n expectation over the set $\mathcal{I}_n$.

When handling expectations we need to properly choose probabilities to assign.

Set of Basic Operations

We focus now on the description of the three elements listed before.

Basic operations

Selecting a set of basic operations is usually an easy task. The classic choice is represented by the set of basic arithmetic and logical operations. In our case we will see what to use later on.

Getting a notion of *input size* and *cost per operation* depends on the way data are represented.

Input Size and Cost per Operation

Two possible ways to represent data:

- *Fixed-precision floating-point numbers* (which are stored in a fixed amount of memory - usually 64 bits): size of an element is usually considered to be 1 and consequently to have unit size per number.
- *Integer data*: we need a number of bits approximately equal to the logarithm of their absolute value (which is the bit size of the integer). Similar ideas can be used when dealing with rational numbers.

Example of bit size for integer number 1024

$$\log_2 1024 = 10.$$

Input Size and Cost per Operation

Let D be some data and $x = (x_1 \dots x_n)^\top \in D^n$.

If data in D are represented with

- unit-size numbers, we have that the size will be n;
- bit-size numbers, the size will be the sum of bit size for each x_i.

Simple operations (e.g., +,-,$\cdot$,/) on

- two unit-size numbers have cost equal to 1 (this is the *unit cost*).
- two bit-size numbers have a cost equal to the product of their bit-sizes (for multiplications and divisions) or their maximum (for additions, subtractions and comparisons).

Computational Model

When comparing algorithms, we need to specify the computational model we use to analyze complexity.

The two options that can be usually considered are:

- *Real number arithmetic model*. idealized reals with unit size and unit cost are used;
- *Turing model of computation*. integer or rational data with their associated bit size are used.

Remark

We use real number arithmetic model in here!!!

Big $\mathcal{O}$ Notation

Big $\mathcal{O}$ Notation

Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We write

$$f(x) = \mathcal{O}(g(x))$$

for $x \to \infty$ if and only if there exist values x_0 and $C > 0$ such that

$$|f(x)| \leq C|g(x)|, \quad \forall x > x_0.$$

Example

If we have

$$f(n) = \log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3,$$

then

$$f(n) = \mathcal{O}(n^3).$$

Polynomial Time

Polynomial time algorithm

An algorithm $\mathcal{A}$ is said to be a *polynomial time* algorithm if $U_w^{\mathcal{A}}(n)$ is bounded above by a polynomial, that is

$$U_w^{\mathcal{A}}(n) \leq \mathcal{O}(n^k), \qquad \textit{with } k > 1.$$

- A problem can be solved in polynomial time if there is a polynomial time algorithm solving the problem.
- The notion of average polynomial time is defined similarly, replacing worst-case with average-case cost function.
- Polynomial time means *efficient* in complexity theory.

IMPORTANT: if an algorithm solves a class of problems in polynomial time, then the algorithm can be considered efficient.

Other Orders of Complexity

Other orders of complexity are reported below:

$\mathcal{O}(1)$	constant time
$\mathcal{O}(\log n)$	logarithmic time
$\mathcal{O}(n)$	linear time
$\mathcal{O}(n \log n)$	log-linear time
$\mathcal{O}(n^p),\ p > 1$	polynomial time
$\mathcal{O}(c^n),\ c > 1$	exponential time
$\mathcal{O}(n!)$	factorial time

Table: Orders of complexity

How to Choose the Algorithm

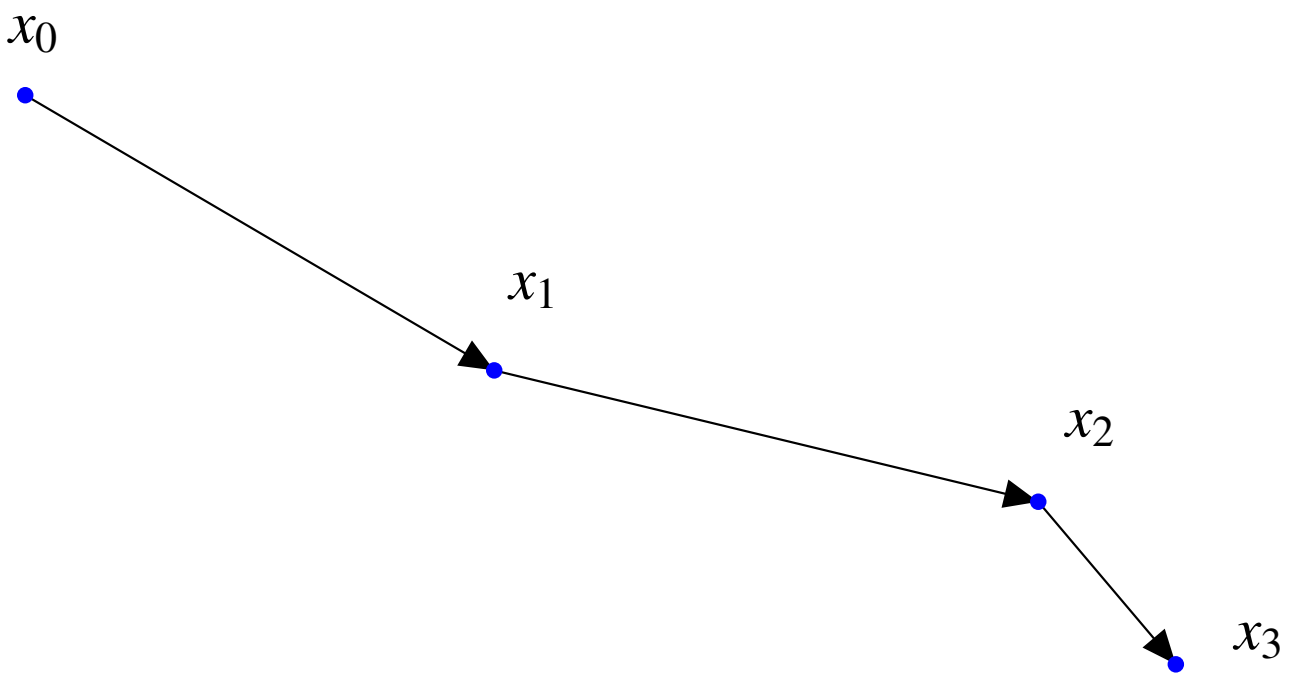
Definition

Iterative algorithm uses an initial guess to generate a sequence of improving approximate solutions (*iterates*) for a class of problems, in which the k-th approximation is derived from the previous ones.

Keep in mind that

- given a class of convex optimization problems, our goal is to pick (or eventually develop) an efficient iterative algorithm for the class, i.e., an algorithm with a "good" efficiency estimate on the class;
- during the course, we will present and analyze a number of efficient algorithms for standard classes of convex optimization problems.

Iterative Algorithm



Our Algorithm Framework

Details

- All we know in advance here is that the problem belongs to a given class (e.g., convex, convex of a given degree of smoothness, etc).
- Besides this a priori qualitative piece of information, we have the possibility to query an “oracle”

Oracle

Definition

An *"oracle"* gives us local information on the objective $f(x)$ and the constraints $g(x)$ (their values and derivatives at a point) for a problem of the form:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g(x) \leq 0, \end{aligned}$$

i.e., feasible set $C = \{x \in \mathbb{R}^n : \ g(x) \leq 0\}$.

Informational Complexity Approach

In order to evaluate the performance of a given algorithm

- We then use the so-called *informational complexity approach*;
- What we do in practice is estimating the number of function evaluations/gradients needed to find an exact or approximate solution for a given problem

$$\min_{x \in C} \quad f(x),$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function and C is some convex set.

Definition

Complexity of the method is the number of calls to the oracle needed to get a solution.

Remark

Computational effort of querying the oracle/processing oracle outputs not included!

Oracle-based Analysis

In our framework:

- No information on the instance (apart from the class it belongs to);
- Information accumulated by calls to the oracle.

Anyway, oracles simplify theoretical analysis of a given algorithm.

Remark

Oracles can make our complexity analysis free of input size!!!

Oracle-based Analysis

We distinguish two different classes of algorithms:

- dimension-free complexity algorithms (e.g., gradient based methods): Described by means of an oracle; algorithm possesses no global information.
- dimension-based complexity algorithms (e.g., interior point methods): Data specifying the problem form the input of the algorithm; algorithm possesses global information!

Remark

Features of dimension-free complexity algorithms make them attractive for optimization in very high dimension.

How to calculate complexity?

- *complexity of the method* is the number of iterations (oracle calls) required to get a solution within a given error $\epsilon > 0$ (or, in some cases, the number of iterations required to get the optimal solution).
- We need a theoretical tool to calculate the complexity.
- We use the *convergence rate* of the algorithm to get complexity bounds.

Error Measure

We indicate with X^* the *set of optimal points* for the problem and with x^* a point belonging to X^*.

Error function

We define a measure of error, or *error function* $\mathcal{E}(x)$ for a given iterate x as follows:

$$\mathcal{E}(x) = \inf_{x^* \in X^*} \|x - x^*\|,$$

or

$$\mathcal{E}(x) = f(x) - f(x^*).$$

Convergence rate

For each given method $\mathcal{A}$, we want to identify an upper bound on the error at iteration k (i.e., $\mathcal{E}(x_k)$). This is the *convergence rate* related to $\mathcal{A}$.

Convergence Rate

In particular, we consider the following inequality

$$\lim_{k\to\infty} \frac{\mathcal{E}(x_{k+1})}{\mathcal{E}(x_k)^p} \leq q$$

and classify the different rates as follows:

- **Linear rate:** we have $p = 1$ and $q \in (0, 1)$. Equivalently, we can write, for all k the following inequality

$$\mathcal{E}(x_k) \leq cq^k, \quad \text{with } c > 0.$$

- **Sublinear rate:** we have $p = 1$ and $q = 1$. Equivalently, we can write, for all k the following inequality

$$\mathcal{E}(x_k) \leq \frac{c}{k^\beta}, \quad \text{with } c, \beta > 0.$$

A few More Details...

Classic asymptotic analysis

We study

$$\lim_{k\to\infty} \frac{\mathcal{E}(x_{k+1})}{\mathcal{E}(x_k)^p} \leq q$$

This is used to analyze what happens at convergence

Modern non-asymptotic analysis

We study the bounds on the reduction obtained after k iterations, e.g.:

$$\mathcal{E}(x_k) \leq c \cdot (e^{-\alpha})^k, \quad c, \alpha > 0$$

That is

$$\mathcal{E}(x_k) = \mathcal{O}(e^{-\alpha k}).$$

This is used to analyze what happens after a few iterations.

What to Use in Data Science?

Remark

In data science applications algorithms stopped long before convergence!!!

We are interested in how many iterations needed to get a good solution

Answer

Better to use Non-Asymptotic Analysis then!

Example

Consider two algorithms having different behaviour, e.g. :

$$\mathcal{E}_{\mathcal{A}_1}(x_k) = \mathcal{O}(1/k^2)$$

and

$$\mathcal{E}_{\mathcal{A}_2}(x_k) = \mathcal{O}(1/k)$$

Asymptotically are both sublinear but $\mathcal{A}_1$ better than $\mathcal{A}_2$!!

Complexity Calculation

Example 1

If, e.g., we have a linear rate $\mathcal{E}(x_k) = \mathcal{O}(e^{-\alpha k})$, with $\alpha > 0$ and we want to get a solution within ϵ, then we consider inequality

$$\mathcal{E}(x_k) \leq ce^{-\alpha k} \leq \epsilon, \tag{1}$$

with $c > 0$, and, by simple calculations, we can write $\log\left(\frac{1}{e^{-\alpha k}}\right) \geq \log\left(\frac{c}{\epsilon}\right)$. So, we have

$$k \geq \frac{1}{\alpha} \log\left(\frac{c}{\epsilon}\right),$$

and the minimum number of iterations needed to satisfy equation (1) is

$$\bar{k} = \frac{1}{\alpha} \log\left(\frac{c}{\epsilon}\right),$$

that is a number of iterations of the order $\mathcal{O}\left(\log \frac{1}{\epsilon}\right)$.

Complexity Calculation

Example 2

Similarly, if we have a sublinear rate, $\mathcal{E}(x_k) = \mathcal{O}(1/k^\beta)$, with $\beta > 0$ and we want to get a solution within ϵ, then we consider inequality

$$\mathcal{E}(x_k) \leq \frac{c}{k^\beta} \leq \epsilon,$$

with $c > 0$, and, by simple calculations, we can write

$$k \geq \left(\frac{c}{\epsilon}\right)^{1/\beta}, \quad \text{and} \quad \bar{k} = \left(\frac{c}{\epsilon}\right)^{1/\beta}$$

that is a number of iterations of the order $\mathcal{O}\left(\frac{1}{\epsilon^{1/\beta}}\right)$.

Remark

It is easy to see that an algorithm with linear rate is much faster (thus getting much better complexity) than one with sublinear rate.

Linear is Better than Sublinear

Remark

It is easy to see that an algorithm with linear rate is much faster (thus getting much better complexity) than one with sublinear rate.

Example $\epsilon = 10^{-6}$

Linear rate

$$\log(10^6) = 6\log(10) \approx 14.$$

Sublinear rate ($\beta = 1$)

$$\cdots \approx 10^6$$

Huge difference!!!!