Introduction to Algorithms for Data Mining and Machine Learning

Xin-She Yang
Middlesex University
School of Science and Technology
London, United Kingdom

ELSEVIER

ACADEMIC PRESS
An imprint of Elsevier

Academic Press is an imprint of Elsevier
125 London Wall, London EC2Y 5AS, United Kingdom
525 B Street, Suite 1650, San Diego, CA 92101, United States
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, United Kingdom

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-817216-2

Publisher: Candice Janco
Acquisition Editor: J. Scott Bentley
Editorial Project Manager: Michael Lutz
Production Project Manager: Nilesh Kumar Shah
Designer: Miles Hitchen

Typeset by VTeX

Contents

Optimization algorithms

3

Contents

Optimization algorithms are diverse with many specialized techniques and a few general techniques. The algorithms for optimization include gradient-based algorithms, gradient-free algorithms, evolutionary algorithms and nature-inspired metaheuristics. We will mainly focus on the introduction of the gradient-based techniques due to their importance in data mining and machine learning. We will also briefly outline some gradient-free methods and metaheuristic algorithms. For a more detailed introduction, we refer the readers to the more advanced literature [22,5,7,159,161].

3.1 Gradient-based methods

Gradient-based methods are iterative methods that extensively use the gradient information of the objective function during iterations. Let us start with the simplest Newton method.

3.1.1 Newton's method

For minimization and maximization of a univariate function $f(x)$, it is equivalent to finding the roots of its gradient $g(x) = f'(x) = 0$. From Newton's root-finding

Introduction to Algorithms for Data Mining and Machine Learning. https://doi.org/10.1016/B978-0-12-817216-2.00010-7

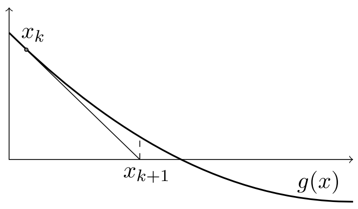

Figure 3.1 Newton's method and iterations.

algorithm (1.6) we have

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)} = x_k - \frac{f'(x_k)}{f''(x_k)}, \tag{3.1}$$

where we have used $g'(x) = f''(x)$, implicitly assuming that these derivatives exist. The main idea of Newton's method is shown in Fig. 3.1.

Example 14

For a simple function $f(x) = (x-1)^2 = x^2 - 2x + 1$ in the real domain $x \in \mathbb{R}$, we know that its global minimum is $f_{\min} = 0$ at $x_* = 1$. Let us use Newton's formula (3.1) to find this solution starting from any value $x_0 = a > 0$. We have $f'(x) = 2x - 2$ and $f''(x) = 2$. From

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{3.2}$$

we have

$$x_1 = x_0 - \frac{2x_0 - 2}{2} = a - \frac{2a-2}{2} = 1, \tag{3.3}$$

which gives the optimal solution in a single step. This shows that this algorithm is very efficient.

It is worth pointing out that this function is a special case because $f(x) = (x-1)^2$ is a convex function, and thus Newton's method can find the solution in a single step. In general, $f(x)$ is not convex with possible multiple solutions, and some care should be taken.

Let us revisit an earlier example (Example 3) where $f(x) = x^2 \exp(-x^2)$ has two maxima at $x_* = \pm 1$ and one minimum at $x_* = 0$. Then $f'(x) = 2x(1-x^2)\mathrm{e}^{-x^2}$ and $f''(x) = 2\mathrm{e}^{-x^2}(1 - 5x^2 + 2x^4)$. Thus, Newton's iterative formula becomes

$$\begin{aligned} x_{k+1} &= x_k - \frac{f'(x_k)}{f''(x_k)} = x_k - \frac{2x_k(1-x_k^2)\mathrm{e}^{-x_k^2}}{2\mathrm{e}^{-x_k^2}(1 - 5x_k^2 + 2x_k^4)} \\ &= x_k - \frac{x_k(1-x_k^2)}{1 - 5x_k^2 + 2x_k^4}, \end{aligned} \tag{3.4}$$

where we have used $\exp(-x_k^2) \neq 0$.

If we start with $x_0 = 0.8$, then we have

$$x_1 = 0.8 - \frac{0.8 \times (1 - 0.8^2)}{1 - 5 \times 0.8^2 + 2 \times 0.8^4} \approx 1.0085747 \tag{3.5}$$

and

$$x_2 = 0.999961, \quad x_3 = 0.9999999, \tag{3.6}$$

which are very close to $x_* = 1.0$.

If we use $x_0 = 0.5$, then we have

$$x_1 = 3.5, \quad x_2 \approx 3.66414799, \quad x_3 \approx 3.818812, \tag{3.7}$$

which gradually moves toward infinity. In this case, the iterative sequence becomes divergent. We can never reach the optimal solution $x_* = 1.0$. Similarly, if we use $x_0 = 2.0$, then it leads a similar divergent sequence.

If we start with $x_0 = 0.2$, then we will get $x_* = 0$ in a few steps. However, if we start with $x_0 = 0.4$, then we have

$$x_1 = -0.93757962, \quad x_2 \approx -0.9988808587, \quad x_3 \approx -0.9999993, \tag{3.8}$$

which rapidly converges toward $x_* = -1.0$.

This highlights an important issue here. Different starting points x_0 can lead to completely different final solutions. The solution sequences will largely depend on the initial solution x_0. It seems that we cannot predict easily which final solutions the iterative procedure may produce. Through detailed mathematical analysis of the iterative formula, it may be possible to figure out critical points for bifurcation and different branches. However, a high nonlinearity of the iteration formula makes it difficult to see which branch a particular initial point may lead to. Even if it is possible, it may not worth the effort because it cannot be generalized.

In fact, this issue of the dependence of the final solutions on the initial point is almost universal for many optimization algorithms, especially for those based on gradient information. The only exception is linear programming and convex optimization. Various efforts have been dedicated to solve this issue so as to design optimization algorithms that are less dependent on (ideally, independent of) initial configuration. We will come back to this issue again in later chapters and will provide, whenever possible, various remedies to this key issue.

3.1.2 Newton's method for multivariate functions

Newton's method works for univariate functions. Now let us extend it to solve optimization problems for multivariate functions. For the minimization of a function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, x_2, ..., x_n)$, the essence of this method is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha g(\nabla f, \boldsymbol{x}^{(k)}), \tag{3.9}$$

where α is the step size, which can vary during iterations, and $g(\nabla f, \boldsymbol{x}^{(k)})$ is a function of the gradient ∇f and the current location $\boldsymbol{x}^{(k)}$. Different methods use different forms of $g(\nabla f, \boldsymbol{x}^{(k)})$.

We know that Newton's method is a popular iterative method for finding the zeros of a nonlinear univariate function of $f(x)$ on the interval $[a, b]$. It can be modified for solving optimization problems because it is equivalent to finding the zeros of the first derivative $f'(\boldsymbol{x})$ once the objective function $f(\boldsymbol{x})$ is given.

For a given continuously differentiable function $f(\boldsymbol{x})$, we have the Taylor expansion about a known point $\boldsymbol{x} = \boldsymbol{x}_k$ (with $\Delta\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_k$)

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_k) + (\nabla f(\boldsymbol{x}_k))^T \Delta\boldsymbol{x} + \frac{1}{2}\Delta\boldsymbol{x}^T \nabla^2 f(\boldsymbol{x}_k)\Delta\boldsymbol{x} + \cdots,$$

which is minimized near a critical point when $\Delta\boldsymbol{x}$ is the solution of the linear equation

$$\nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\Delta\boldsymbol{x} = 0, \quad \text{or} \quad \boldsymbol{x} = \boldsymbol{x}_k - \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x}_k), \tag{3.10}$$

where $\boldsymbol{H} = \nabla^2 f(\boldsymbol{x}_k)$ is the Hessian matrix. If the iteration procedure starts from the initial vector $\boldsymbol{x}^{(0)}$ (usually taken to be a guessed point in the domain), then Newton's iteration formula for the kth iteration is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \boldsymbol{H}^{-1}(\boldsymbol{x}^{(k)})\nabla f(\boldsymbol{x}^{(k)}). \tag{3.11}$$

It is worth pointing out that if $f(\boldsymbol{x})$ is quadratic, then the solution can be found exactly in a single step. However, this method may become tricky for nonquadratic functions, especially when we have to calculate a large Hessian matrix.

It can usually be time-consuming to calculate the Hessian matrix for second derivatives. A good alternative is to use an identity matrix to approximate the Hessian by using $\boldsymbol{H}^{-1} = \boldsymbol{I}$, and we have the quasi-Newton method

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha\boldsymbol{I}\,\nabla f(\boldsymbol{x}^{(k)}), \tag{3.12}$$

where $\alpha \in (0, 1)$ is a step size. In this case, the method is essentially the steepest descent method.

Though gradient-based methods can be very efficient, the final solution tends to depend on the starting point. If the starting point is very far away from the optimal solution, the algorithm can either reach a completely different solution for multimodal problems or simply fail in some cases. Therefore there is no guarantee that the global optimal solution can be found.

It is worth pointing out that there are many variations of the steepest descent methods. If such optimization aims is to find the maximum, then this method becomes a *hill-climbing* method because the aim is to climb up the hill to the highest peak.

3.1.3 Line search

In the steepest descent method, there are two important parts, the descent direction and the step size (or how far to descend). The calculations of the exact step size may be very time consuming. In reality, we intend to find the right descent direction. Then a reasonable amount of descent, not necessarily the exact amount, during each iteration will usually be sufficient. For this, we essentially use a line search method.

To find the local minimum of the objective function $f(\boldsymbol{x})$, we try to search along a descent direction $\boldsymbol{s}_k$ with an adjustable step size α_k so that

$$\psi(\alpha_k) = f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k) \tag{3.13}$$

decreases as much as possible, depending on the value of α_k. Loosely speaking, a reasonably right step size should satisfy the Wolfe conditions

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k) \le f(\boldsymbol{x}_k) + \gamma_1 \alpha_k \boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k) \tag{3.14}$$

and

$$\boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k) \ge \gamma_2 \boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k), \tag{3.15}$$

where $0 < \gamma_1 < \gamma_2 < 1$ are algorithm-dependent parameters. The first condition is a sufficient decrease condition for α_k, often called the Armijo condition or rule, whereas the second inequality is often referred to as the curvature condition. For most functions, we can use $\gamma_1 = 10^{-4}$ to 10^{-2} and $\gamma_2 = 0.1$ to 0.9. These conditions are usually sufficient to ensure the algorithm converge in most cases; however, stronger conditions may be needed for some tough functions.

The basic steps of the line search method can be summarized in Algorithm 2.

Algorithm 2 Line search method.

```
1: Initial guess $\boldsymbol{x}_0$ at $k = 0$
2: while $\|\nabla f(\boldsymbol{x}_k)\| >$ accuracy do
3:     Find the search direction $\boldsymbol{s}_k = -\nabla f(\boldsymbol{x}_k)$
4:     Solve for $\alpha_k$ by decreasing $f(\boldsymbol{x}_k + \alpha \boldsymbol{s}_k)$ significantly
5:         satisfying the Wolfe conditions
6:     Update the result $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$
7:     $k \leftarrow k + 1$
8: end while
```

3.2 Variants of gradient-based methods

Over many decades, a class of gradient-based methods have been developed. They all use some forms of gradient information, though their algorithmic procedures can be very different. Here, we introduce a few commonly used variants. In the context of machine learning, there are some comprehensive reviews. For example, Ruder [125] provided an overview of gradient descent optimization algorithms.

3.2.1 Stochastic gradient descent

In many optimization problems, especially in deep learning, the objective function or risk function to be minimized can be written in the following form:

$$E(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{x}_i, \boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\big[u_i(\boldsymbol{x}_i, \boldsymbol{w}) - \bar{y}_i\big]^2, \tag{3.16}$$

where

$$f_i(\boldsymbol{x}_i, \boldsymbol{w}) = \big[u_i(\boldsymbol{x}_i, \boldsymbol{w}) - \bar{y}_i\big]^2. \tag{3.17}$$

Here, $\boldsymbol{w} = (w_1, w_2, \dots, w_K)^T$ is a parameter vector such as the weights in a neural network. In addition, $\bar{y}_i$ $(i = 1, 2, \dots, m)$ are the target or real data (data points or data sets), whereas $u_i(\boldsymbol{x}_i)$ are the predicted values based on the inputs $\boldsymbol{x}_i$ by a model such as the models based on trained neural networks.

The standard gradient descent for finding new weight parameters in terms of iterative formula can be written as

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \frac{\eta}{m}\sum_{i=1}^{m}\nabla f_i, \tag{3.18}$$

where $0 < \eta \le 1$ is the learning rate or step size. Here, the gradient ∇f_i is with respect to $\boldsymbol{w}$. This requires the calculations of m gradients. When m is large and the number of iteration t is large, this can be very expensive.

To save computation, the true gradient can be approximated by the gradient at a single value at f_i instead of all m values, that is,

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \nabla f_i, \tag{3.19}$$

where η_t is the learning rate at iteration t, which may vary with iterations. Though this is a crude estimate at a randomly selected point i at iteration t to the true gradient, the computation costs have dramatically reduced by a factor of $1/m$. Due to the random nature of the selection of a sample i (which can be very different at each iteration), this way of calculating gradient is called stochastic gradient. The method based on such crude estimation of gradients is called stochastic gradient descent (SGD) for minimization or stochastic gradient ascent (SGA) for maximization.

The learning rate η_t should be reduced gradually. For example, a commonly used reduction of learning rates is

$$\eta_t = \frac{1}{1+\beta t}, \quad t = 1, 2, \dots, \tag{3.20}$$

where $\beta > 0$ is a hyper-parameter (see Fig. 3.2).

Bottou showed that SGD almost surely converges if

$$\sum_t \eta_t = \infty, \quad \sum_t \eta_t^2 < \infty. \tag{3.21}$$

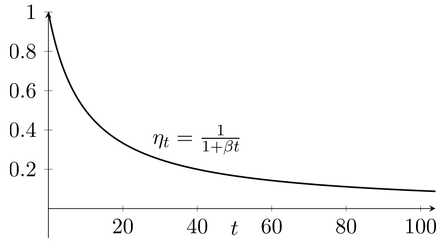

Figure 3.2 Monotonic decrease of the learning rate.

The best convergence rate is $\eta_t \sim 1/t$ with the averaged residual error decreasing as $E \sim 1/t$ [18].

It is worth pointing out the stochastic gradient descent is not the direct descent in the true gradient sense, but the descent is in terms of average or expectation. Thus, the paths can still be zig-zag, sometimes, it may be up the gradient, not necessarily all the way down the gradient directions, but the overall computation efficiency is usually much higher than the true gradient descent for large-scale problems. Therefore, it is widely used for deep learning problems and large-scale problems.

3.2.2 Subgradient method

All the gradient-based methods mentioned assume implicitly that the functions are differentiable. In the case of nondifferentiable functions, we have to use the subgradient method for non-differential convex functions or more generally gradient-free methods for nonlinear functions to be introduced later in this chapter.

For nondifferentiable convex functions, the subgradient vectors $\boldsymbol{v}_k$ can be defined by

$$f(\boldsymbol{x}) - f(\boldsymbol{x}_k) \geq \boldsymbol{v}_k^T(\boldsymbol{x} - \boldsymbol{x}_k), \tag{3.22}$$

and Newton's iteration formula can be replaced by

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \boldsymbol{v}_k, \tag{3.23}$$

where α_k is the step size at iteration k. As the iteration formula involves the subgradient $\boldsymbol{v}_k = \partial f(\boldsymbol{x}_k)$ calculated at iteration k, the method is called the subgradient method.

It is worth pointing out that since there are many arbitrary subgradients (see Fig. 3.3), the subgradient calculated at $\boldsymbol{x}_k$ may not be in the desirable direction. Some choices such as choosing greater values of the norm $\boldsymbol{v}$ can be expected.

In addition, though a constant step size $\alpha_k = \alpha$ where $0 < \alpha < 1$ can work well in many cases, it is desirable that the step size α_k should vary and be scaled when appropriate. For example, a commonly used scheme for varying step sizes is $\alpha_k \geq 0$, $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$, and $\lim_{k\to\infty} \alpha_k = 0$.

The subgradient method is still used in practice, and it can be very effective in combination with the stochastic gradient method, which leads to a class of so-called

Xin-She Yang

Introduction to Algorithms for Data Mining and Machine Learning

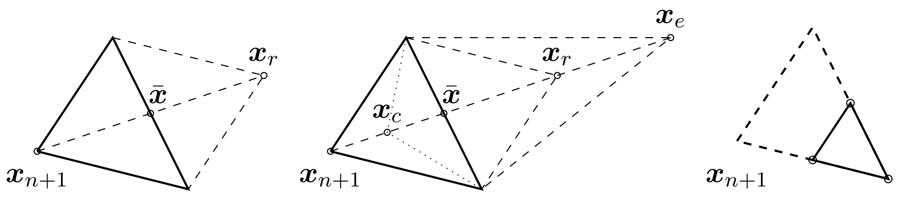

Figure 3.4 Simplex transformation: (a) reflection (left), (b) expansion and contraction (middle), (c) reduction (right).

3.4 Gradient-free methods

Though gradient-based methods are very efficient, they need to calculate derivatives during iterations. For some problems, the computation of derivatives can be expensive. For problems with discontinuous objectives, it is not possible to calculate such derivatives. In this case, methods that do not require derivatives are preferred. Such derivative-free or gradient-free methods can also be very effective.

The Nelder–Mead method is a downhill simplex algorithm for unconstrained optimization without using derivatives, and it was first developed in 1965 by Nelder and Mead [109]. This is one of widely used traditional methods since its computational effort is relatively small and is something to get a quick grasp of the optimization problem. The basic idea of this method is to use the flexibility of the constructed simplex via amoeba-style manipulations by reflection, expansion, contraction, and reduction (see Fig. 3.4). In some books, such as the best-known *Numerical Recipes*, it is also called the "Amoeba algorithm" [115]. It is worth pointing out that this downhill simplex method has nothing to do with the simplex method for linear programming.

In the n-dimensional space, a simplex, which is a generalization of a triangle on a plane, is a convex hull with $n+1$ distinct points. For simplicity, a simplex in the n-dimensional space is referred to as an n-simplex. Therefore a 1-simplex is a line segment, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and so on.

There are a few variants of the algorithm that use slightly different ways of constructing initial simplex and convergence criteria. However, the fundamental procedure is the same (see Algorithm 3).

The first step is constructing an initial n-simplex with $n+1$ vertices and evaluating the objective function at the vertices. Then, by ranking the objective values and reordering the vertices, we have an ordered set, so that

$$f(\boldsymbol{x}_1) \le f(\boldsymbol{x}_2) \le \cdots \le f(\boldsymbol{x}_{n+1}) \tag{3.48}$$

at $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{n+1}$, respectively. As the downhill simplex method is for minimization, we use the convention that $\boldsymbol{x}_{n+1}$ is the worse point (solution) and $\boldsymbol{x}_1$ is the best solution. Then, at each iteration, similar ranking manipulations are carried out.

Algorithm 3 Nelder–Mead (downhill simplex) method.

```
1:  Initialize a simplex with $n+1$ vertices in $n$ dimension.
2:  while (stop criterion is not true) do
3:      Reorder the points so that $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \cdots \leq f(\boldsymbol{x}_{n+1})$
          with $\boldsymbol{x}_1$ being the best and $\boldsymbol{x}_{n+1}$ being the worse (highest value)
4:      Find the centroid $\bar{\boldsymbol{x}}$ using $\bar{\boldsymbol{x}} = \sum_{i=1}^{n} \boldsymbol{x}_i/n$ excluding $\boldsymbol{x}_{n+1}$.
5:      Generate a trial point via the reflection of the worse vertex
6:      $\boldsymbol{x}_r = \bar{\boldsymbol{x}} + \alpha(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$ where $\alpha > 0$ (typically $\alpha = 1$)
7:      if $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_r) < f(\boldsymbol{x}_n)$ then
8:          $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$;
9:      end if
10:     if $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$ then
11:         Expand in the direction of reflection $\boldsymbol{x}_e = \boldsymbol{x}_r + \beta(\boldsymbol{x}_r - \bar{\boldsymbol{x}})$
12:         if $(f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r))$ $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_e$ else $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$; end
13:     end if
14:     if $f(\boldsymbol{x}_r) > f(\boldsymbol{x}_n)$ then
15:         Contract by $\boldsymbol{x}_c = \boldsymbol{x}_{n+1} + \gamma(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$;
16:         if $f(\boldsymbol{x}_c) < f(\boldsymbol{x}_{n+1})$ then $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_c$;
17:         else Reduction $\boldsymbol{x}_i = \boldsymbol{x}_1 + \delta(\boldsymbol{x}_i - \boldsymbol{x}_1)$, $(i = 2, \ldots, n+1)$; end
18:     end if
19: end while
```

Then, we have to calculate the centroid $\boldsymbol{x}$ of the current simplex excluding the worst vertex $\boldsymbol{x}_{n+1}$:

$$\bar{\boldsymbol{x}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i. \tag{3.49}$$

Using the centroid as the basis point, we try to find the reflection of the worse point $\boldsymbol{x}_{n+1}$ by

$$\boldsymbol{x}_r = \bar{\boldsymbol{x}} + \alpha(\bar{x} - \boldsymbol{x}_{n+1}) \qquad (\alpha > 0), \tag{3.50}$$

though the typical value of $\alpha = 1$ is often used.

Whether the new trial solution is accepted or not and how to update the new vertex depends on the objective function at $\boldsymbol{x}_r$. There are three possibilities:

- If $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_r) < f(\boldsymbol{x}_n)$, then replace the worst vertex $\boldsymbol{x}_{n+1}$ by $\boldsymbol{x}_r$, that is, $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$.
- If $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$, which means the objective improves, then we seek a more bold move to see if we can improve the objective even further by moving or expanding the vertex further along the line of reflection to a new trial solution

$$\boldsymbol{x}_e = \boldsymbol{x}_r + \beta(\boldsymbol{x}_r - \bar{\boldsymbol{x}}), \tag{3.51}$$

where $\beta = 2$. Now we have to test if $f(\boldsymbol{x}_e)$ improves even better. If $f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r)$, then we accept it and update $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_e$; otherwise, we can use the result of the reflection, that is, $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$.

- If there is no improvement or $f(\boldsymbol{x}_r) > f(\boldsymbol{x}_n)$, then we have to reduce the size of the simplex while maintaining the best sides. This is the contraction

$$\boldsymbol{x}_c = \boldsymbol{x}_{n+1} + \gamma(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1}), \tag{3.52}$$

where $0 < \gamma < 1$, though $\gamma = 1/2$ is usually used. If $f(\boldsymbol{x}_c) < f(\boldsymbol{x}_{n+1})$, then we update $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_c$.

If all these steps fail, then we have to reduce the size of the simplex toward the best vertex $\boldsymbol{x}_1$. This is the reduction step

$$\boldsymbol{x}_i = \boldsymbol{x}_1 + \delta(\boldsymbol{x}_i - \boldsymbol{x}_1) \qquad (i = 2, 3, \ldots, n+1). \tag{3.53}$$

Then, we go to the first step of the iteration process and start over again.

There are many other gradient-free optimization algorithms, and swarm intelligence-based algorithms are mostly gradient-free.

3.5 Evolutionary algorithms and swarm intelligence

The literature on evolutionary algorithms and nature-inspired algorithms is expanding rapidly. Most of these algorithms have drawn inspiration from evolutionary characteristics of biological or natural systems, and these algorithms form the majority of the evolutionary algorithms. In addition, recent algorithms use multiple agents to mimic the collective or social characteristics of swarming systems, and these algorithms somehow can simulate certain aspects of swarm intelligence (SI). In general, a vast majority of these algorithms are nature-inspired algorithms.

There are a large number of different algorithms, including evolutionary strategy, simulated annealing (SA), colony optimization (ACO), bees algorithms, genetic algorithm, bat algorithm, cuckoo search, differential evolution, firefly algorithm, particle swarm optimization, flower pollination algorithm, harmony search, memetic algorithm, and others. To introduce these algorithms systematically, an entire book [159] is required; therefore, we will only briefly introduce some of the most recent and widely used nature-inspired optimization algorithms [159,158].

Though many algorithms such as ACO, SA, and others belong to this category, we will not introduce them since they are not widely used yet in machine learning and data mining. We start with genetic algorithms.

3.5.1 Genetic algorithm

The genetic algorithm (GA), developed by John Holland and his collaborators in the 1960s and 1970s, is a model or abstraction of biological evolution based on Charles

Darwin's theory of natural selection. The genetic algorithm (GA) is an evolutionary algorithm and probably the most widely used. It is becoming a conventional and classic method. However, it does have fundamental genetic operators that have inspired many later algorithms, so we will introduce it in detail. There are many variants of the genetic algorithm, and they now form a class of genetic algorithms [75,57]. The essence of genetic algorithms involves the encoding of an objective function as arrays of bits or character strings to represent the chromosomes, the manipulation operations of strings by genetic operators, and the selection according to their fitness with the aim of finding a solution to the problem concerned. This is often done by the following procedure: 1) encoding of solutions into strings; 2) defining a fitness function and selection criterion; 3) creating a population of individuals and evaluating their fitness; 4) evolving the population by generating new solutions using crossover, mutation, fitness-proportionate reproduction; 5) selecting new solutions according to their fitness and replacing the old population by better individuals; and 6) decoding the results to obtain the solution(s) to the problem.

An important issue is the formulation or choice of an appropriate fitness function that determines the selection criterion in a particular problem. For the minimization of $f(\boldsymbol{x})$ using genetic algorithms, one simple way of constructing a fitness function is to use the simplest form $F(\boldsymbol{x}) = A - f(\boldsymbol{x})$ with A being a large constant (though $A = 0$ will do), and thus the objective is maximizing the fitness function. However, there are many different ways of defining a fitness function. For example, we can use the individual fitness assignment relative to the whole population

$$F(\boldsymbol{x}_i) = \frac{f(\boldsymbol{x}_i))}{\sum_{i=1}^{N} f(\boldsymbol{x}_i)}, \tag{3.54}$$

where N is the population size. The appropriate form of the fitness function will ensure that the solutions with higher fitness will be selected efficiently. Poorly defined fitness functions may result in incorrect or meaningless solutions.

Another important issue is the choice of various parameters. The crossover probability p_c is usually very high, typically in the range 0.7–0.99. On the other hand, the mutation probability p_m is usually small (usually 0.001–0.05). If p_c is too small, then the crossover occurs sparsely, which is not efficient for evolution. If the mutation probability is too high, then the diversity of the population may be too high, which makes it harder for the system to converge.

The selection criterion is also important so as to select the current population so that the best individuals with higher fitness are preserved and passed on to the next generation, which is often carried out in association with a certain elitism. The basic elitism is to select the most fit individual (in each generation), which will be carried over to the new generation without being modified by genetic operators. This ensures that the best solution is achieved more quickly.

3.5.2 Differential evolution

Differential evolution (DE) was developed in 1997 by Storn and Price [137]. It is a vector-based algorithm, which has some similarity to pattern search and genetic algorithms due to its use of crossover and mutation. DE is a stochastic search algorithm with self-organizing tendency and does not use the information of derivatives. Thus it is a population-based derivative-free method. In addition, DE uses real numbers as solution strings, and thus no encoding and decoding are needed.

For a D-dimensional optimization problem with D parameters, a population of n solution vectors is initially generated, and we have $\boldsymbol{x}_i$ for $i = 1, 2, \ldots, n$. For each solution $\boldsymbol{x}_i$ at any generation t, we use the conventional notation

$$\boldsymbol{x}_i^t = (x_{1,i}^t, x_{2,i}^t, \ldots, x_{D,i}^t), \tag{3.55}$$

which consists of D components in the D-dimensional space. This vector can be considered as the chromosomes or genomes.

Differential evolution consists of three main steps: mutation, crossover, and selection.

Mutation is carried out by the mutation scheme. For each vector $\boldsymbol{x}_i$ at any time or generation t, we first randomly choose three distinct vectors $\boldsymbol{x}_p$, $\boldsymbol{x}_q$, and $\boldsymbol{x}_r$ at t and then generate a so-called donor vector by the mutation scheme

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{x}_p^t + F(\boldsymbol{x}_q^t - \boldsymbol{x}_r^t), \tag{3.56}$$

where $F \in [0, 2]$ is a parameter, often referred to as the differential weight. This requires that the minimum number of population size is $n \geq 4$. In principle, $F \in [0, 2]$, but in practice, a scheme with $F \in [0, 1]$ is more efficient and stable. In fact, almost all the studies in the literature use $F \in (0, 1)$.

Crossover is controlled by a crossover parameter $C_r \in [0, 1]$, controlling the rate or probability for crossover. The actual crossover can be carried out in two ways, binomial and exponential. The binomial scheme performs crossover on each of the D components or variables/parameters. By generating a uniformly distributed random number $r_i \in [0, 1]$ the jth component of $\boldsymbol{v}_i$ is manipulated as

$$\boldsymbol{u}_{j,i}^{t+1} = \begin{cases} \boldsymbol{v}_{j,i} & \text{if } r_i \leq C_r, \\ \boldsymbol{x}_{j,i}^t & \text{otherwise}, \end{cases} \qquad j = 1, 2, \ldots, D. \tag{3.57}$$

This way, each component can be decided randomly whether or not to exchange with the counterpart of the donor vector.

In the exponential scheme, a segment of the donor vector is selected, and this segment starts with random integer k and random length L, which can include many components. Mathematically, choosing $k \in [0, D-1]$ and $L \in [1, D]$ randomly, we have

$$\boldsymbol{u}_{j,i}^{t+1} = \begin{cases} \boldsymbol{v}_{j,i}^t & \text{for } j = k, \ldots, k - L + 1 \in [1, D], \\ \boldsymbol{x}_{j,i}^t & \text{otherwise}. \end{cases} \tag{3.58}$$

The binomial scheme is simpler to implement.

Selection is essentially the same as that used in genetic algorithms. It is selecting the most fittest, that is, the minimum objective value for a minimization problem. Therefore we have

$$\boldsymbol{x}_i^{t+1} = \begin{cases} \boldsymbol{u}_i^{t+1} & \text{if } f(\boldsymbol{u}_i^{t+1}) \le f(\boldsymbol{x}_i^t), \\ \boldsymbol{x}_i^t & \text{otherwise.} \end{cases} \tag{3.59}$$

It is worth pointing out here that the use of $\boldsymbol{v}_i^{t+1} \neq \boldsymbol{x}_i^t$ may increase the evolutionary or exploratory efficiency. The overall search efficiency is controlled by two parameters, the differential weight F and the crossover probability C_r.

3.5.3 Particle swarm optimization

Many swarms in nature such as fish and birds can have higher-level behavior, but they all obey simple rules. For example, a swarm of birds such as starlings simply follow three basic rules: each bird flies according to the flight velocities of their neighbor birds (usually about seven adjacent birds) while keeping a certain separation distance; birds on the edge of the swarm tend to fly into the center of the swarm (so as to avoid being eaten by potential predators such as eagles); and, in addition, birds tend to fly to search for food or shelters, and thus a short memory is used. Based on such swarming characteristics, in 1995 particle swarm optimization (PSO) was developed by Kennedy and Eberhart [87], which uses equations to simulate the swarming characteristics of birds and fish.

For the ease of discussions, let us use $\boldsymbol{x}_i$ and $\boldsymbol{v}_i$ to denote the position (solution) and velocity, respectively, of a particle or agent i. In PSO, there are n particles as a population, and thus $i = 1, 2, \ldots, n$. There are two equations for updating positions and velocities of particles, and they can be written as follows:

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \alpha \boldsymbol{\epsilon}_1 [\boldsymbol{g}^* - \boldsymbol{x}_i^t] + \beta \boldsymbol{\epsilon}_2 [\boldsymbol{x}_i^* - \boldsymbol{x}_i^t], \tag{3.60}$$

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1} \Delta t, \tag{3.61}$$

where $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ are two uniformly distributed random numbers in [0,1]. The learning parameters α and β are usually in the range of [0,2]. In Eq. (3.60), $\boldsymbol{g}^*$ is the best solution found so far by all the particles in the population, and each particle has an individual best solution $\boldsymbol{x}_i^*$ by itself during the entire past iteration history.

It is worth pointing out that $\Delta t = 1$ should be used because iterations in algorithms are discrete with a step counter $t \leftarrow t + 1$. Thus, there is no need to consider units and Δt in all algorithms discussed in this book.

3.5.4 Bat algorithm

Bat algorithm (BA), developed by Xin-She Yang [150,152] in 2010, uses some characteristics of frequency-tuning and echolocation of microbats. It also uses the variations of pulse emission rate r and loudness A to control exploration and exploitation. In the bat algorithm, main algorithmic equations for position $\boldsymbol{x}_i$ and velocity $\boldsymbol{v}_i$ for bat i are

$$f_i = f_{\min} + (f_{\max} - f_{\min})\beta, \tag{3.62}$$

$$\boldsymbol{v}_i^t = \boldsymbol{v}_i^{t-1} + (\boldsymbol{x}_i^{t-1} - \boldsymbol{x}_*) f_i, \tag{3.63}$$

$$\boldsymbol{x}_i^t = \boldsymbol{x}_i^{t-1} + \boldsymbol{v}_i^t \Delta t, \tag{3.64}$$

where $\beta \in [0, 1]$ is a random vector drawn from a uniform distribution so that the frequency can vary from $f_{\min}$ to $f_{\max}$. Here $\boldsymbol{x}_*$ is the current best solution found so far by all virtual bats. As pointed out earlier, $\Delta t = 1$ is used for iterative discrete algorithms.

From these equations we can see that both equations are linear in terms of $\boldsymbol{x}_i$ and $\boldsymbol{v}_i$. But the control of exploration and exploitation is carried out by the variations of loudness $A(t)$ from a high value to a lower value and the emission rate r from a lower value to a higher value, that is,

$$A_i^{t+1} = \alpha A_i^t, \quad r_i^{t+1} = r_i^0 (1 - e^{-\gamma t}), \tag{3.65}$$

where $0 < \alpha < 1$ and $\gamma > 0$ are two parameters. As a result, the actual algorithm can have a weak nonlinearity. Consequently, BA can have a faster convergence rate in comparison with PSO.

3.5.5 Firefly algorithm

Based on the flashing characteristics of tropical firefly species, in 2008 Xin-She Yang [148,149] developed the firefly algorithm (FA). FA uses a nonlinear system by combining the exponential decay of light absorption and inverse-square law of light variation with distance. In the FA, the main algorithmic equation for the position $\boldsymbol{x}_i$ (as a solution vector to a problem) is

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \beta_0 e^{-\gamma r_{ij}^2} (\boldsymbol{x}_j^t - \boldsymbol{x}_i^t) + \alpha\, \boldsymbol{\epsilon}_i^t, \tag{3.66}$$

where α is a scaling factor controlling the step sizes of the random walks, whereas γ is a scale-dependent parameter controlling the visibility of the fireflies (and thus search modes). In addition, β_0 is the attractiveness constant when the distance between two fireflies is zero (i.e., $r_{ij} = 0$). This system is a nonlinear system, which may lead to rich characteristics in terms of algorithmic behavior.

Since the brightness of a firefly is associated with the objective landscape with its position as the indicator, the attractiveness of a firefly seen by others, depending on their relative positions and relative brightness. Thus the beauty is in the eye of the beholder. Consequently, a pair comparison is needed for comparing all fireflies.

3.5.6 Cuckoo search

In the natural world, among 141 cuckoo species, 59 species engage the so-called obligate brood parasitism. These cuckoo species do not build their own nests, and they lay eggs in the nests of host birds such as warblers. In fact, there is an arms race between cuckoo species and host species, forming an interesting cuckoo-host species coevolution system.

Based the above characteristics, Xin-She Yang and Suash Deb [154–156] developed in 2009 the cuckoo search (CS) algorithm. CS uses a combination of both local and global search capabilities, controlled by a discovery probability p_a. There are two algorithmic equations in CS, and one equation is

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \alpha s \otimes H(p_a - \epsilon) \otimes (\boldsymbol{x}_j^t - \boldsymbol{x}_k^t), \tag{3.67}$$

where $\boldsymbol{x}_j^t$ and $\boldsymbol{x}_k^t$ are two different solutions selected randomly by random permutation, $H(u)$ is the Heaviside function, ϵ is a random number drawn from a uniform distribution, and s is the step size. This step is primarily local, though it can become global search if s is large enough. However, the main global search mechanism is realized by the other equation with Lévy flights:

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \alpha L(s, \lambda), \tag{3.68}$$

where the Lévy flights are simulated (or drawn random numbers) by drawing random numbers from the Lévy distribution

$$L(s, \lambda) \sim \frac{\lambda \Gamma(\lambda) \sin(\pi \lambda / 2)}{\pi} \frac{1}{s^{1+\lambda}} \quad (s \gg 0). \tag{3.69}$$

Here $\alpha > 0$ is the step size scaling factor. It is worth pointing out that we use "$\sim$" here to highlight the fact that the steps are drawn from the distribution on the right-hand side as a sampling technique.

3.5.7 Flower pollination algorithm

Flower pollination algorithm (FPA) is a population-based algorithm, developed by Xin-She Yang and his collaborators, based on the inspiration from the pollination characteristics of flowering plants [153,157,159]. FPA intends to mimic some key characteristics of biotic and abiotic pollination as well as coevolutionary flower constancy between certain flower species and some pollinator species such as insects and animals.

In essence, there are two main equations for this algorithm, and the global search is carried out by

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \gamma L(\lambda)(\boldsymbol{g}_* - \boldsymbol{x}_i^t), \tag{3.70}$$

where γ is a scaling parameter, $L(\lambda)$ is the random number vector drawn from a Lévy distribution governed by the exponent λ in the same form given in (3.69). Here $\boldsymbol{g}_*$ is the best solution found so far, which acts as a selection mechanism. The current solution $\boldsymbol{x}_i^t$ is modified by varying step sizes because Lévy flights can have a fraction of large step sizes in addition to many small steps.

The local search is carried out by

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + U(\boldsymbol{x}_j^t - \boldsymbol{x}_k^t), \tag{3.71}$$

which mimics local pollination and flower constancy. Here U is a uniformly distributed random number. Furthermore, $\boldsymbol{x}_j^t$ and $\boldsymbol{x}_k^t$ are solutions representing pollen from different flower patches.

As we mentioned earlier, the literature is expanding, and more nature-inspired algorithms are being developed by researchers, but we will not introduce more algorithms here. We refer the interested readers to more specialized literature such as Yang's book [159].

3.6 Notes on software

As we mentioned earlier, many software packages and programming languages have implemented some optimization capabilities, whereas commercial software packages tend to have well-tested toolboxes. It is not our intention to provide a comprehensive list of toolboxes and functionalities; we only intend to provide some flavor and diversity of a few software packages or programming languages.

- Matlab: The optimization toolboxes of Matlab include linear programming `linprog`, integer programming `intlinprog`, nonlinear programming such as `fminsearch` and `fmincon`, quadratic programming `quadprog`, and multiobjective optimization by genetic algorithm `gamultiobj`.
- Octave has many functionalities similar to Matlab, and it is an open-source package. Its optimization toolbox `optim` has implemented linear programming, quadratic programming, nonlinear programming, and linear least squares.
- R has a relatively general purpose optimization solver `optimr` with `optim()` using conjugate gradient, Nelder–Mead method, Broyden–Fletcher–Goldfarb–Shanno (BFGS) method, and simulated annealing. It also has a quadratic programming `solve.QP()` and least-squares solver `solve.qr()` as well as metaheuristic optimization such as the firefly algorithm.
- Python does have good optimization capabilities via `scipy.optimize()`, which includes the BFGS method, conjugate gradient, Newton's method, trust-region method, and least-square minimization.
- Mathematica is a commercial symbolic computation package. It has powerful functionalities for optimization, including nonlinear constrained global optimization `NMiminize` or `NMaximize`, linear programming and integer programming `LinearProgramming`, Knapsack solver `KnapsackSolve`, traveling salesman problem `FindShortesTour`, and others.
- Maple is mainly a symbolic and numerical computing tool with some functions for optimization such as `Minimize`, linear programming `LPSolve`, and nonlinear programming `NLPSolve`.
- Microsoft Excel Solver can do linear programming, integer programming, generalized reduced gradient, evolutionary algorithms (via a variant of the genetic

algorithm). On the other hand, the OpenSolver is free and has no limit on the number of variables. Its core algorithmic engine is COIN-OR linear and integer programming optimizers, and thus OpenSolver is very powerful and efficient.

Other powerful optimization tools include the computational infrastructure for Operations Research (COIN-OR), also known as common optimization interface for OR. Many software packages use it as a core optimization engine.

There are some Matlab demo codes for most of the nature-inspired algorithms discussed in this book. They are available from Matlab file exchanges,[2] including

- accelerated particle swarm optimization,[3]
- firefly algorithm,[4]
- cuckoo search,[5]
- flower pollination algorithm.[6]

It is worth pointing out that these codes are demo and incomplete codes. The reason is that such demo codes focus on the essential steps of the algorithms without any messy implementation of handling constraints. However, the performance of such concentrated demo codes may be reduced as the proper constraint-handling is an important part of practical applications. These codes should still work reasonably well for solving function optimization problems. This gives the readers an opportunity to understand the basic algorithms and potentially improve them.

[2] https://uk.mathworks.com/matlabcentral/profile/authors/2652824-xin-she-yang.
[3] https://uk.mathworks.com/matlabcentral/fileexchange/29725-accelerated-particle-swarm-optimization.
[4] https://uk.mathworks.com/matlabcentral/fileexchange/29693-firefly-algorithm.
[5] https://uk.mathworks.com/matlabcentral/fileexchange/29809-cuckoo-search-cs-algorithm.
[6] https://uk.mathworks.com/matlabcentral/fileexchange/45112-flower-pollination-algorithm.