1 Machine learning algorithms

This chapter covers

- Types of ML algorithms
- The importance of learning algorithms from scratch
- An introduction to Bayesian inference and deep learning
- Software implementation of machine learning algorithms from scratch

An *algorithm* is a sequence of steps required to achieve a particular task. An algorithm takes an input, performs a sequence of operations, and produces a desired output. The simplest example of an algorithm is *sorting*, where given a list of integers, we perform a sequence of operations to produce a sorted list. A sorted list enables us to organize information better and find answers in our data.

Two popular questions to ask about an algorithm are how fast it runs (run-time complexity) and how much memory it takes (memory or space complexity) for an input of size n. For example, a comparison-based sort, as we'll see later, has $O(n \log n)$ run-time complexity and requires $O(n)$ memory storage.

There are many approaches to sorting, and in each case, in the classic algorithmic paradigm, the algorithm designer creates a set of instructions. Imagine a world where you can *learn* the instructions based on a sequence of input and output examples available to you. This is a setting of the ML algorithmic paradigm. Like a human brain learns when one plays a connect-the-dots game or sketches a natural landscape, we compare the desired output with what we have at each step and fill in the gaps. This, in broad strokes, is what (supervised) machine learning (ML) algorithms do. During training, ML algorithms learn the rules (e.g., classification boundaries) based on training examples by optimizing an objective function. During testing, ML algorithms apply previously learned rules to new input data points and provide a prediction, as shown in figure 1.1.

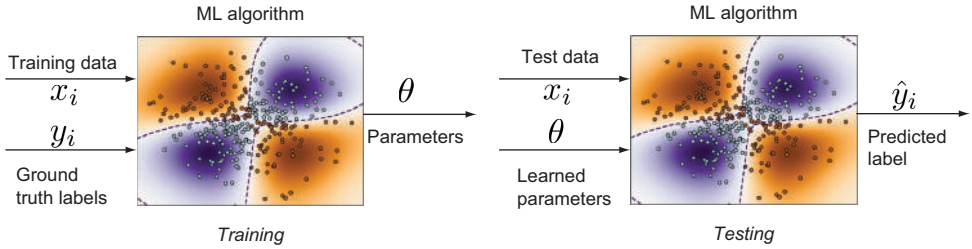

Figure 1.1 Supervised learning: training (left) and testing (right)

1.1 Types of ML algorithms

Let's unpack the previous paragraph a little bit and introduce some notation. This book focuses on ML algorithms that can be grouped together into the following categories: supervised learning, unsupervised learning, and deep learning. In *supervised learning*, the task is to learn a mapping f from inputs x to outputs y given a training dataset $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n input–output pairs. In other words, we are given n examples of what the output should look like given the input. The output y is also often referred to as the *label*, and it is the supervisory signal that tells our algorithm what the correct answer is.

Supervised learning can be subdivided into *classification* and *regression*, depending on the quantity we are trying to predict. If our output y is a discrete quantity (e.g., K distinct classes), we have a classification problem. On the other hand, if our output y is a continuous quantity (e.g., a real number, such as stock price) we have a regression problem.

Thus, the nature of the problem changes based on the quantity y we are trying to predict. We want to get as close as possible to the ground truth value of y.

A common way to measure performance or closeness to ground truth is via the *loss function*. The loss function computes a distance between the prediction and the true label. Let $y = f(x; \theta)$ be our ML algorithm that maps input examples x to output labels y, parameterized by θ, where θ captures all the learnable parameters of our ML algorithm.

Then, we can define the classification loss function as the number of misclassified samples, as represented by equation 1.1.

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} 1[y_i \neq f(x_i;\theta)] \tag{1.1}$$

Here, 1[] is an indicator function that is equal to 1 when the argument inside is true and 0 otherwise. The expression in equation 1.1 denotes that we are adding up all the instances in which our prediction $f(x_i;\ \theta)$ did not match the ground truth label y_i and dividing by the total number of examples n. In other words, we are computing an average misclassification rate. Our goal is to minimize the loss function (i.e., find a set of parameters θ that make the misclassification rate as close to zero as possible). Note that there are many alternative loss functions for classification, such as cross-entropy, which we will examine in later chapters.

For continuous labels or response variables, a common loss function is the *mean squared error* (MSE). The MSE measures how far away our estimate is from the ground truth, as illustrated in equation 1.2.

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} [y_i - f(x_i;\theta)]^2 \tag{1.2}$$

As we can see from the equation, we subtract our prediction from the ground truth label, square it, and average the result over the data points. By taking the square we eliminate the negative sign and penalize large deviations from the ground truth.

One of the central goals of ML is to be able to generalize to unseen examples. We want to achieve high accuracy (low loss) on not just the training data (which is already labeled) but on new, unseen, test data examples. This generalization ability is what makes ML so attractive: if we can design ML algorithms that can see outside their training box, we'll be one step closer to artificial general intelligence (AGI).

In *unsupervised learning*, we are not given the label y, nor are we learning the mapping between input and output examples; instead, we are interested in making sense of the data itself. Usually, that implies discovering patterns in data. It's often easier to discover patterns if we project high-dimensional data into a lower-dimensional space as shown in figure 1.2. Therefore, in the case of unsupervised learning, our training dataset consists of $D = \{x_1,\ldots,x_n\}$ of n input examples, without any corresponding labels y. The simplest example of unsupervised learning is finding clusters within data. Intuitively, we know that data points that belong to the same cluster have similar characteristics. In fact, data points within a cluster can be represented by the cluster center as an exemplar and used as part of a data compression algorithm. Alternatively, we can look at the distances between clusters in a projected lower-dimensional space to understand the interrelation between different groups. Additionally, a point that's far away from all the existing clusters can be considered an anomaly, leading to an anomaly

detection algorithm. As you can see, there's an infinite number of interesting use cases that arise from unsupervised learning, and throughout this book, we'll be learning some of the most intriguing algorithms in that space from scratch.

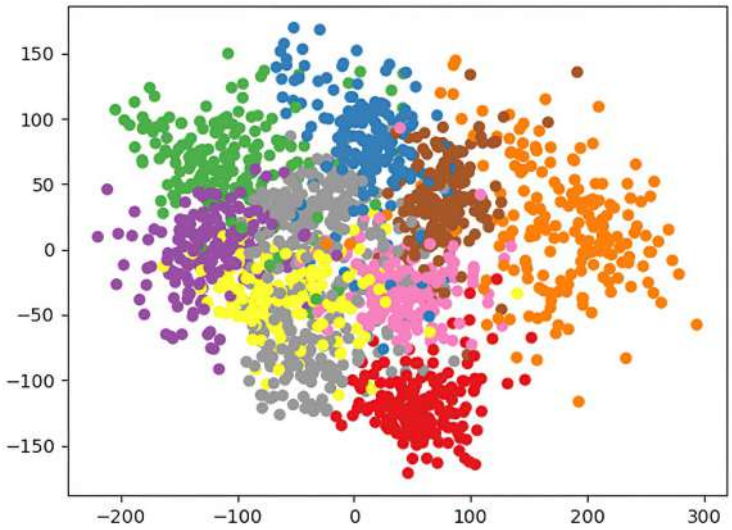

Figure 1.2 Unsupervised learning: clusters of data points projected onto 2-dimensional space

Another very important area of modern machine algorithms is *deep learning*. The name comes from a stack of computational layers forming a computational graph together. The depth of this graph refers to sequential computation and the breadth to parallel computation.

As we'll see, deep learning models gradually refine their parameters through a back-propagation algorithm until they meet the objective function. Deep learning models have permeated the industry due to their ability to solve complex problems with high accuracy. For example, figure 1.3 shows a deep learning architecture for sentiment analysis. We'll learn more about what individual blocks represent in future chapters.

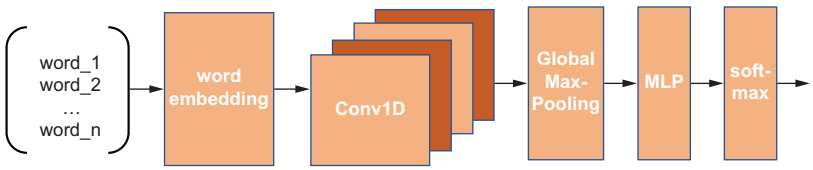

Figure 1.3 Deep neural network (DNN) architecture for sentiment analysis

Deep learning is a very active research area, and we'll be focusing on modern deep learning algorithms throughout this book. For example, in self-supervised learning, used in transformer models, we are using the context and structure of the natural

language as a supervisory signal, thereby extracting the labels from the data itself. In addition to classic applications of deep learning in natural language processing (NLP) and computer vision (CV), we'll discuss generative models, learning how to predict time-series data and journey into relational graph data.

1.2 Why learn algorithms from scratch?

Understanding ML algorithms from scratch has several valuable outcomes for the reader. First, we will be able to choose the right algorithm for the task. By knowing the inner workings of an algorithm, we can understand its shortcomings, assumptions made in the derivation of the algorithm, as well as advantages in different data scenarios. This enables us to exercise judgment when selecting the right solution to a problem and save time by eliminating approaches that don't work.

Second, we will be able to explain the results of a given algorithm to stakeholders. Being able to interpret and present the results to the audience in industrial or academic settings is an important trait of an ML algorithm designer.

Third, we will be able to use intuition developed by reading this book to troubleshoot advanced ML problems. Breaking down a complex problem into smaller pieces and understanding where things went wrong often requires a strong sense of fundamentals and algorithmic intuition. This book will allow the reader to construct minimum working examples and build upon existing algorithms to develop and be able to debug more complex models.

Fourth, we will be able to extend an algorithm when a new situation arises in the real world—particularly, where the textbook algorithm or a library cannot be used as is. The in-depth understanding of ML algorithms that you will acquire in this book will help you modify existing algorithms to meet your needs.

Finally, we are often interested in improving the performance of existing models. The principles discussed in this book will enable the reader to accomplish that. In conclusion, understanding ML algorithms from scratch will help you choose the right algorithm for the task, explain the results, troubleshoot advanced problems, extend an algorithm to a new scenario, and improve the performance of existing algorithms.

1.3 Mathematical background

To learn ML algorithms from scratch, it's a good idea to review concepts from applied probability, calculus, and linear algebra. For a review of probability, the reader is encouraged to consult Dimitri Bertsekas and John Tsitsiklis's *Introduction to Probability* (Athena Scientific, 2002). The reader is expected to be familiar with continuous and discrete random variables, conditional and marginal distributions, the Bayes rule, Markov chains, and limit theorems.

For a review of calculus, I recommend James Stewart's *Calculus* (Thomson Brooks/Cole, 2007). The reader is expected to be familiar with the rules of differentiation and integration, sequences and series, vectors and the geometry of space, partial derivatives, and multidimensional integrals.

Finally, *Introduction to Linear Algebra* by Gilbert Strang (Wellesley-Cambridge Press, 2016) serves as a great introduction to linear algebra. The reader is expected to be familiar with vector spaces, matrices, eigenvalues and eigenvectors, matrix norms, matrix factorizations, positive definite and semidefinite matrices, and matrix calculus. In addition to the aforementioned texts, please see appendix A for recommended texts that can enrich your understanding of the algorithms presented in this book.

1.4 Bayesian inference and deep learning

Bayesian inference allows us to update our beliefs about the world given observed data. Our minds hold a variety of mental models explaining different aspects of the world, and by observing new data points, we can update our latent representation and improve our understanding of reality. Any *probabilistic model* is described by a set of parameters θ modeled as random variables, which control the behavior of the model, and associated data x.

The goal of Bayesian inference is to find the posterior distribution $p(\theta|x)$ (distribution over the parameters given the data) to capture a particular aspect of reality well. The posterior distribution is proportional to the product of the likelihood $p(x|\theta)$ (distribution over the data given the parameters) and the prior $p(\theta)$ (initial distribution over the parameters), which follows from the Bayes rule in equation 1.3.

$$\underbrace{p(\theta|x)}_{\text{Posterior}} = \frac{\overbrace{p(x|\theta)}^{\text{Likelihood}}\overbrace{p(\theta)}^{\text{Prior}}}{\underbrace{p(x)}_{\text{Evidence}}} = \frac{p(x|\theta)p(\theta)}{\underbrace{\int p(x|\theta)p(\theta)d\theta}_{\text{Partition function Z}}} \sim p(x|\theta)p(\theta) \tag{1.3}$$

The prior $p(\theta)$ is our initial belief and can be either noninformative (e.g., uniform over all possible states) or informative (e.g., based on experience in a particular domain). Moreover, our inference results depend on the prior we choose—not only the value of prior parameters but also on the functional form of the prior. We can imagine a chain of updates in which the prior becomes a posterior as more data is obtained in the form of a Bayes engine, as shown in figure 1.4. We can see how our prior is updated to a posterior via the Bayes rule as we observe more data.

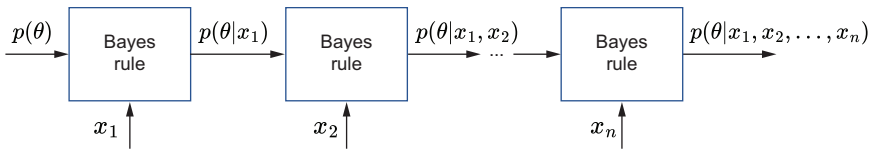

Figure 1.4 A Bayes engine showing the transformation of a prior to a posterior as more data is observed

Priors that have the same form as the posterior are known as *conjugate priors*, which are generally preferred, since they simplify computation by having closed-form updates. The denominator $Z = p(x) = \int p(x|\theta)\, p(\theta)\, d\theta$ is known as the *normalizing constant* or the *partition function* and is often intractable to compute, due to integration in high dimensional parameter space. In this book, we'll examine several techniques for estimating Z.

We can model relationships between different random variables in our model as a graph as shown in figure 1.5, giving rise to *probabilistic graphical models*. Each node in the graph represents a random variable (RV), and each edge represents a conditional dependency. The model parameters are represented by clear nodes, while shaded nodes represent the observed data, and rectangular plates denote a copy or a repetition of the random variable. The topology of the graph itself changes depending on the application you are interested in modeling. However, the goals of Bayesian inference remain the same: to find the posterior distribution over model parameters, given observed data.

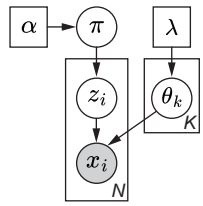

Figure 1.5 A probabilistic graphical model (PGM) for a Gaussian mixture model

In contrast to PGMs, where the connections are specified by domain experts, *deep learning models* learn representations of the world automatically through the backpropagation algorithm that minimizes an objective function. *Deep neural networks* (DNNs) consist of multiple layers parameterized by weight matrices and bias parameters. Mathematically, DNNs can be expressed as a composition of individual layer functions, as in equation 1.4.

$$\mathrm{DNN}(x;\theta) = f_L(f_{L-1}(\cdots(f_1(x;\theta_1))\cdots);\theta_L) \tag{1.4}$$

Here, $f_l(x) = f(x;\theta_l)$ is the function at layer l. The compositional form of the DNNs reminds us of the chain rule when it comes to differentiating parameters as part of stochastic gradient descent. Throughout this book, we'll look at several different kinds of DNNs, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), as well as transformers and graph neural networks (GNNs), with applications ranging from computer vision to finance.

1.4.1 Two main camps of Bayesian inference: MCMC and VI

Markov chain Monte Carlo (MCMC) is a methodology of sampling from high-dimensional parameter spaces to approximate the posterior distribution $p(\theta|x)$. In statistics, *sampling* refers to generating a random sample of values from a given probability distribution. There are many approaches to sampling in high-dimensional parameter spaces. As we'll see in later chapters, MCMC is based on constructing a Markov chain whose stationary distribution is the target density of interest (i.e., posterior distribution). By performing a random walk over the state space, the fraction of time we spend in each state θ will be proportional to $p(\theta|x)$. As a result, we can use Monte Carlo integration to derive the quantities of interest associated with our posterior distribution.

Before we discuss high-dimensional parameter spaces, let's examine how we can sample from low-dimensional spaces. The most popular method for sampling from univariate distributions is known as the *inverse cumulative density function* (CDF) *method*; it is defined as $\text{CDF}_X(x) = P(X \leq x)$ (see figure 1.6).

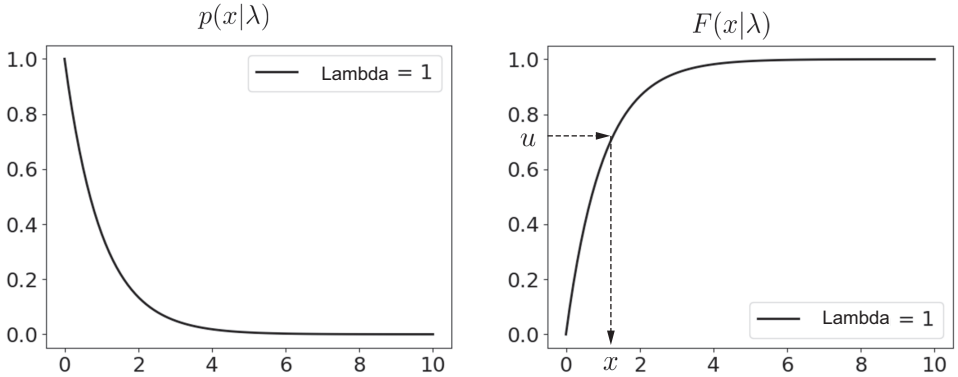

Figure 1.6 Exponential RV probability density function (left) and cumulative density function (right)

For example, let's examine an exponential RV with a probability density function (PDF) and CDF given by equation 1.5.

$$p(x|\lambda) = \lambda e^{-\lambda x}, x \geq 0 \qquad F(x|\lambda) = \int_0^x p(x|\lambda)dx = 1 - e^{-\lambda x}, x \geq 0 \quad (1.5)$$

The inverse CDF can be found as shown in equation 1.6.

$$F^{-1}(u) = -\frac{\ln(1-u)}{\lambda} \quad (1.6)$$

Thus, to generate a sample from an exponential RV, we first need to generate a sample from uniform random variable $u \sim \text{Unif}(0, 1)$ and apply the transformation $-\ln(1-u)/\lambda$. By generating enough samples, we can achieve an arbitrary level of accuracy. One challenge of MCMC is determining how to *efficiently* generate samples from high-dimensional distributions. We'll look at two ways of doing that in this book: Gibbs sampling and Metropolis-Hastings (MH) sampling.

Variational inference (VI) is an optimization-based approach to approximating the posterior distribution $p(x)$. We simplify the notation here by assigning a generic distribution p(x) the meaning of a posterior distribution. The basic idea behind VI is to choose an approximate distribution $q(x)$ from a family of tractable distributions and then make this approximation as close as possible to the true posterior distribution $p(x)$. A *tractable distribution* simply refers to one that is easy to compute. As we will see in the mean-field section of the book, the approximate $q(x)$ can take on a fully

factored representation of the joint posterior distribution. This factorization significantly speeds up computation.

Next, we introduce the Kullback-Leibler (KL) divergence and use it to measure the proximity of our approximate distribution to the true posterior. Figure 1.7 shows the two versions of KL divergence.

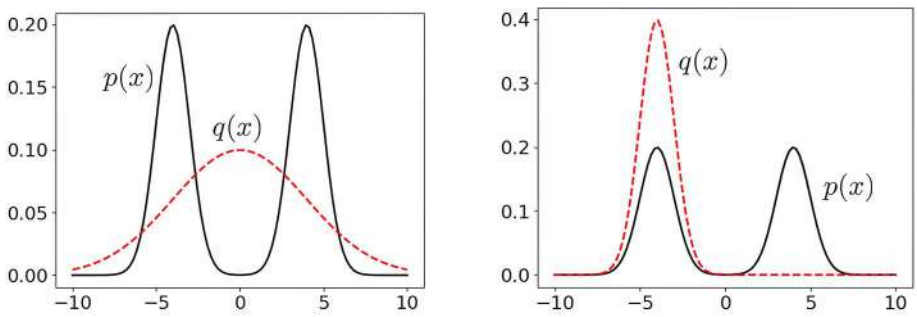

Figure 1.7 Forward KL (left) and reverse KL (right) approximate distribution q(x) fit to Gaussian mixture $p(x)$

The original distribution $p(x)$ is a bimodal Gaussian distribution (aka Gaussian mixture with two components), while the approximating distribution $q(x)$ is a unimodal Gaussian. As we can see from figure 1.7, we can approximate the distribution with two peaks either at the center with $q(x)$ that has high variance, to capture the support of the bimodal distribution, or at one of its modes, as shown on the right. This is a result of forward and reverse KL divergence definitions in equation 1.7. As we'll see in later chapters, by minimizing KL divergence, we effectively convert VI into an optimization problem.

$$KL(p\|q) = \sum p(x) \log \frac{p(x)}{q(x)} \qquad KL(q\|p) = \sum q(x) \log \frac{q(x)}{p(x)} \tag{1.7}$$

1.4.2 Modern deep learning algorithms

Over the years, deep learning architecture has evolved from the basic building blocks of the LeNet CNN to visual transformer architectures. Certain architectural design themes, such as residual connections in the ResNet model, which became the standard architectural choice of modern neural networks of arbitrary depth, began to emerge. In the later chapters of this book, we'll take a look at modern deep learning algorithms, including self-attention based transformers; generative models, such as variational autoencoders; and graph neural networks.

We will also discuss amortized variational inference, which is an interesting research area, as it combines the expressiveness and representation learning of deep neural networks with domain knowledge of probabilistic graphical models. We will see one such

application of mixture density networks, where we'll use a neural network to map from observation space to the parameters of the approximate posterior distribution.

Most deep learning models fall in the category of narrow AI showing high performance on a specific dataset. While it is a useful skill to be able to do well on a narrow set of tasks, we would like to generalize away from narrow AI and towards artificial general intelligence (AGI).

1.5 Implementing algorithms

A key part of learning algorithms from scratch is software implementation. It's important to write good code that is efficient both in its use of data structures and its low algorithmic complexity. Throughout this chapter, we'll be grouping the functional aspects of the code into classes and implementing different computational methods from scratch. Thus, you'll be exposed to a lot of object-oriented programming (OOP), which is common practice with popular ML libraries, such as scikit-learn. While the intention of this book is to write all code from scratch (without reliance on third-party libraries), we can still use ML libraries (e.g., scikit-learn, https://scikit-learn.org/stable/) to check the results of our implementation, if available. We'll be using the Python language throughout this book.

1.5.1 Data structures

A *data structure* is a way of storing and organizing data. Each data structure offers different performance tradeoffs, and some are more suitable for the task than others. We'll be using *linear data structures*, such as fixed-size arrays, primarily in our implementation of ML algorithms, since the time to access an element in the array is constant $O(1)$. We'll also frequently use dynamically resizable arrays (e.g., lists in Python) to keep track of data over multiple iterations.

Throughout the book, we'll be using *nonlinear data structures*, such as maps (dictionaries) and sets. We'll use these data structures because ordered dictionaries and ordered sets are built upon self-balanced binary search trees (BSTs) that guarantee $O(n\log n)$ insertion, search, and deletion operations. Finally, a hash table or unordered map is another commonly used, efficient data structure with $O(1)$ access time, assuming no collisions.

1.5.2 Problem-solving paradigms

Most of the ML algorithms in this book can be grouped into one of four problem-solving paradigms: complete search, greedy, divide and conquer, and dynamic programming. *Complete search* is a method for solving a problem by traversing the entire search space, looking for a solution. A machine learning example in which a complete search takes place is an exact inference by complete enumeration. During exact inference, we must completely specify a number of probability tables to carry out our calculations.

A *greedy algorithm* takes a locally optimum choice at each step, with the hope of eventually reaching a globally optimum solution. Greedy algorithms often rely on a

greedy heuristic. A machine learning example of a greedy algorithm consists of sensor placement. For example, given a room and several temperature sensors, we would like to place the sensors in a way that maximizes room coverage.

Divide and conquer is a technique that divides the problem into smaller, independent sub-problems and then combines each of their solutions. A machine learning example that uses the divide and conquer paradigm can be found in the CART decision tree algorithm. As we'll see in a future chapter, in the CART algorithm, an optimum threshold for splitting a decision tree is found by optimizing a classification objective (e.g., Gini index). The same procedure is applied to a tree of depth one greater, resulting in a recursive algorithm.

Finally, *dynamic programming* (DP) is a technique that divides a problem into smaller, overlapping subproblems, computes a solution for each, and stores the solution in a DP table. A machine learning example that uses dynamic programming occurs in reinforcement learning (RL) when finding a solution to Bellman equations. For a small number of states, we can compute the Q-function in a tabular way using dynamic programming.

Summary

- An algorithm is a sequence of steps required to achieve a particular task. Machine learning algorithms can be grouped into one of the following categories: supervised learning, unsupervised learning, and deep learning.
- Understanding algorithms from scratch will help you choose the right algorithm for the task, explain the results, troubleshoot advanced problems, and improve the performance of existing models.
- Bayesian inference allows us to update our beliefs about the world given observed data, while deep learning models learn representations of the world through the algorithm of backpropagation that minimizes an objective function. There are two camps of Bayesian inference: Markov chain Monte Carlo (MCMC) and variational inference (VI). These camps focus on sampling and approximating the posterior distribution, respectively.
- It is important to write good code that is efficient both in its use of data structures and its low algorithmic complexity. Most of the ML algorithms in this book can be grouped into one of four problem-solving paradigms: complete search, greedy, divide and conquer, and dynamic programming.