Chapter 2

Supervised Learning

Supervised learning is the machine learning task of inferring a function that maps an input to an output based on example input–output pairs. Depending on whether the output is a continuous variable or a categorical variable, the supervised learning can be further divided into two types:

- Regression;
- Classification.

In Section 2.1, we first focus on regression problems. A general framework of regression includes the model, loss function, optimization, prediction and validation. Each component of the regression framework is discussed in detail in the following sections. In Section 2.2, we explain how to go from regression to classification. Lastly, we discuss how to use an ensemble of multiple models to enhance the performance of supervised learning.

2.1 Framework of Regression

Let us introduce the standard setup of the regression problem. For concreteness, we consider the case of a scalar output. Suppose that we have the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where (x_i, y_i) denotes the i^{th} input–output pair (also called the i^{th} sample). Each sample input x_i is a d-dimensional vector, i.e., $x_i := (x_i^{(1)}, \ldots, x_i^{(d)}) \in \mathbb{R}^d$. Assume that there exists $f : \mathbb{R}^d \to \mathbb{R}$, such that

$$y_i = f(x_i) + \varepsilon_i, \tag{2.1}$$

where $y_i \in \mathbb{R}$ and ε_i are independent and identically distributed (iid) random variables with $\mathbb{E}[\varepsilon_i | x_i] = 0$. For the regularity assumption of f, assume that f is a continuous function. For ease of notation, we also adopt the matrix form for $\mathcal{D} = (X, Y)$, where

$$X = \begin{pmatrix} x_1^{(1)}, x_1^{(2)}, \ldots, x_1^{(d)} \\ \vdots \quad \vdots \qquad\quad \vdots \\ x_N^{(1)}, x_N^{(2)}, \ldots, x_N^{(d)} \end{pmatrix} \text{ and } Y = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}, \tag{2.2}$$

where X is an $N \times d$ matrix and Y is an $N \times 1$ vector.

The first question we ask is how to estimate the corresponding output for any given new input x_*. In the context of the regression problem, a rigorous mathematical formulation of this question is to estimate $\mathbb{E}[y|x = x^*]$, i.e., $f(x^*)$ for any given new input data x^*, which is equivalent to estimating f. Thus f is also called the *mean* function of the regression problem.

The next important question is how to choose the best estimator for f among different possible estimators, which boils down to what "best" means and how to quantify the performance of each estimator.

In the following, we explain how to approach the above two questions and summarize this as a general framework for regression. Recall that the goal of the regression problem is to learn the fixed but unknown mean function f from the labeled dataset $\mathcal{D}$ such that Equation (2.1) holds. A natural step is to postulate the model f_θ to describe the unknown mean function f, where θ are the model parameters that fully characterize the model f_θ. In this way, the problem of finding f is translated into finding the best parameters θ to fit the data.

To find the best parameters θ, we need to quantify what we mean by "the best parameters." Motivated by this, we propose the *loss function* to quantify the discrepancy between the model estimated output $f_\theta(x)$ and the actual output y. Once choosing the loss function $L(\theta|\mathcal{D})$, the optimal parameter set θ^* is defined to be the one that minimizes the loss function. In most cases, there is no closed formula for the optimal parameter set θ^*, and we need to use the numerical optimization method. No matter how we obtain the estimator of the optimal parameters θ^*, either by closed formula or numerical methods, once we have θ^*, we are ready to make prediction. More specifically, for any new input x_*, the estimator of the conditional expectation of the output $\mathbb{E}[y_*|x_*]$ is given by $f_{\theta^*}(x_*)$. Lastly, we need to quantify the goodness of the fit by specifying the metrics, e.g., mean squared error (MSE), R-squared (R^2). Those metrics may not be the same as the one used in the loss function.

Table 2.1. The framework of regression.

Dataset:	$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$
Model:	$f_\theta(x) \approx \mathbb{E}[y\|x] = f(x),\ \forall x \in \mathbb{R}^d$
Empirical Loss:	$L(\theta\|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^N d(f_\theta(x_i), y_i)$ →Minimize
Optimization:	$\theta^* = \arg\min_\theta(L(\theta\|\mathcal{D}))$
Prediction:	$\hat{y}_* = f_{\theta^*}(x_*)$
Validation:	Compute the indicators for the goodness of fit

Table 2.1 summarizes the entire process that we described above. Dataset, model, empirical loss, optimization, prediction and validation are the key elements of supervised learning. We follow this general framework to introduce several supervised learning algorithms in the following chapters and summarize each algorithm in the framework box.

In the rest of the chapter, we discuss each component of the framework, including model, loss function, optimization and prediction/prediction in details.

2.1.1 Model

In this subsection, we introduce various types of models, ranging from linear models to non-linear models and explain the main idea behind most non-linear models—so-called basis expansion. In regression, the proposed model is a family of parametric functions, say f_θ, where θ denotes the parameter set, which fully characterizes the model. For simplicity, we focus on the one-dimensional output case.

Let us start with the simplest model—the linear model—where we assume that $f_\theta\colon \mathbb{R}^d \to \mathbb{R}$ is a linear function, i.e., $\forall x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)}) \in \mathbb{R}^d$,

$$f_\theta(x) = \theta^T x = \sum_{j=1}^{d} \theta^{(j)} x^{(j)},$$

where $\theta = (\theta^{(1)}, \ldots, \theta^{(d)}) \in \mathbb{R}^d$ is the parameter set. This is the model adopted by linear regression methods (Chapter 3).

However, linear models might not be rich enough to describe the complex functional relationship between the input and the output. Motivated by this, there are various types of non-linear models. We list some popular non-linear models as follows, but the list is not exhaustive.

◇ Polynomial model, e.g.,

$$f_\theta(x) = x\mu + x\Sigma x^T,$$

where $\theta = (\mu, \Sigma)$, and $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^d \times \mathbb{R}^d$.

◇ Spline model, e.g.,

$$f_\theta(x) = \sum_{i=1}^{M} C_i(x - l_i)^+,$$

where $\theta = (l_i, C_i)_{i=1}^M$ are model parameters.

◇ Regression tree model (Chapter 4):

$$f(x) = \sum_{m=1}^{M} c_m \mathbb{I}(x \in R_m),$$

where $\{R_1, R_2, \ldots, R_M\}$ is a partition of the input space with M disjoint regions. The tree model allows the partition of the input space by splitting variables and points, which agrees with the topology that a tree should have (e.g., Figure 2.1).

◇ Neural network model (Chapter 5).
Neural network models are based on a collection of connected neurons (nodes). There are various types, which are illustrated in Figure 2.2. We elaborate the main types of neural network models in Chapter 5.

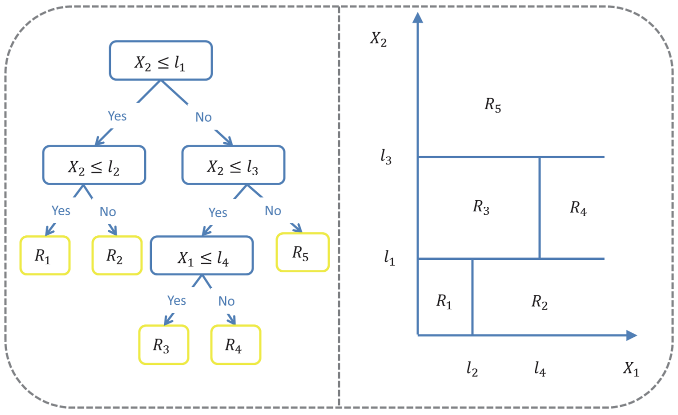


Figure 2.1. An example of a tree model.

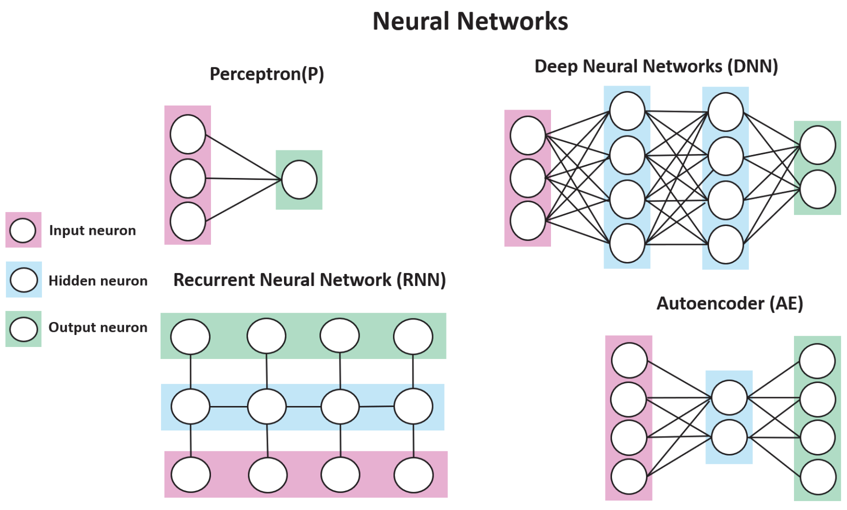


Figure 2.2. Examples of the main types of neural network architectures.

2.1.2 Loss function

In statistics, the loss function (also called cost function) is proposed to quantify the difference between estimated and actual values for output data. It serves as a utility function for parameter estimation. The loss function is a measure for parameters. The smaller the value of the loss function, which indicates that the estimated output is closer to its actual output, the better the parameter is.

The concept of the loss function represents the price paid for inaccuracy of predictions in learning problems. One of the most commonly used loss functions in regression is the quadratic loss function, which is defined as the squared error between the model estimated output and the actual output (see Definition 2.1).

Definition 2.1 (Quadratic Loss Function). Let f_θ denote the model fully characterized by parameters θ. The quadratic loss function is defined to be that $\forall(x, y) \in E \times \mathbb{R}$,

$$Q_\theta(x, y) = (y - f_\theta(x))^2.$$

We can evaluate the loss function for each sample. Averaging the loss function of all samples leads to the empirical risk, which denotes the average loss on the whole data set.

Definition 2.2 (Empirical Risk). Let Q_θ denote a loss function where θ is a model parameter set. Then the empirical risk denoted by L is defined as follows:

$$L(\theta|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i),$$

where $\mathcal{D} = (x_i, y_i)_{i=1}^N$.

In the following we often call the empirical risk the loss function.

2.1.3 Optimization

After specifying the loss function $L(\theta|\mathcal{D})$, the next step is to find the optimal parameter set θ^* to minimize the loss function. In general, unlike for standard linear regression (Ordinary Least Squares, OLS for short), there is no closed formula for the optimal parameters $\hat{\theta}$. It is important to design an effective numerical algorithm to find the optimal parameters. There are various numerical methods for optimization methods, including

- gradient descent based methods;
- gradient boosting method;
- expectation–maximization method.

In this section, we focus on the gradient descent based methods. The gradient boosting method is left for discussion in Chapter 4 and the expectation–maximization method (EM) is covered in Chapter 6.

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function, which can be applied to tackle numerical optimization. We start with the gradient descent (GD) method and explain the main idea and intuition behind it. Batch gradient descent (BGD) is an algorithm for employing GD to estimate the optimal parameters to minimize the loss function. Then we discuss the variants of GD to accommodate the computational issues caused by large scale datasets by introducing randomness to the GD, i.e., stochastic gradient descent (SGD) and mini-batch gradient descent (mini-batch GD). Those methods are particularly widely used for the neural network models discussed in Chapter 5.

2.1.3.1 *Gradient descent method*

Gradient descent (GD) is a general first-order iterative algorithm to solve optimization problem numerically, which can find the local optimal $\hat{\theta}$ such that $\hat{\theta}$ achieves the local minimum of a given differentiable function $f : \mathbb{R}^p \to \mathbb{R}$. The main idea is to find a local minimum of a function using GD by taking steps that are proportional to the negative of the gradient of the function at the current point.

Intuitively, imagine that you are lost in the mountains in a dense fog, and you only feel the slope of the ground below your feet. A reasonable strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. Mathematically what we aim to do is to construct a convergent sequence of $(\theta_n)_{n=0}^{\infty}$ such that

$$\theta^* = \lim_{n\to\infty} \theta_n, \tag{2.3}$$

where θ^* is a local minimum. A sufficient condition for such $(\theta_n)_{n=0}^{\infty}$ is that

(a) there exists an integer N_0 large enough such that $(f(\theta_n))_{n\geq N_0}$ is a non-increasing sequence w.r.t. n.
(b) when $\lim_{n\to\infty} \theta_n = \theta^*$,

$$\lim_{n\to\infty} \triangledown f(\theta_n) = 0, \tag{2.4}$$

where $\triangledown f(\theta)$ is the derivative of f at θ, i.e., $\triangledown L(\theta) = (\partial_{\theta_1} L(\theta), \cdots, \partial_{\theta_p} L(\theta))$.
(c) the derivative of f is continuous.

When $\triangledown L(\theta)$ is continuous, then condition (b) implies that

$$\triangledown f(\theta^*) = \triangledown f(\lim_{n\to\infty} \theta_n) = 0, \tag{2.5}$$

i.e., $f(\theta^*)$ is the local minimum of f.

In the GD algorithm, at the $(n+1)^{th}$ iteration, for given θ_n, we update the $(n+1)^{th}$ estimator θ_{n+1} by

$$\theta_{n+1} = \theta_n - \eta \triangledown L(\theta_n),$$

where $\eta > 0$ is a constant, which is also called the learning rate and will be discussed in detail below. Next, let us explain why the above update can fulfill the sufficiency condition.

(a) When η is small enough, by Taylor's expansion,

$$L(\theta_{n+1}) - L(\theta_n) \approx \triangledown L(\theta_n) \underbrace{(\theta_{n+1} - \theta_n)}_{-\eta \triangledown L(\theta_n)} = -\eta \left(\triangledown L(\theta_n)\right)^2 \leq 0.$$

It follows that for some integer $N_0 > 0$, $(L(\theta_n))_{n\geq N_0}$ is a non-increasing sequence as the above equation holds when the first order Taylor expansion holds.

(b) Suppose that $\{\theta_n\}$ is a convergent series, then

$$\begin{aligned}\lim_{n\to\infty}\theta_{n+1} &= \lim_{n\to\infty}\theta_n - \eta\lim_{n\to\infty}\nabla L(\theta_n)\\ \Downarrow \quad & \quad\ \ \Downarrow\\ \theta^* &= \theta^* - \eta\lim_{n\to\infty}\nabla L(\theta_n).\end{aligned}$$

It follows that $\lim_{n\to\infty}\nabla L(\theta_n) = 0$.

The GD algorithm is often called the *steepest gradient descent.* Let us explain to you the reason behind this name. By Taylor expansion, we have that

$$L(\theta) \approx L(\theta_0) + \nabla L(\theta_0)(\theta - \theta_0).$$

In the above Taylor expansion approximation, $L(\theta)$ decreases fastest on the optimal direction, which is equivalent to the minimization of $\nabla L(\theta_0)(\theta - \theta_0)$. We can show that the gradient direction $\nabla L(\theta_0)$ is the optimal direction, given the constraint that the distance between θ_0 and θ is a positive constant η. Mathematically, it is equivalent to show that if θ^* is the solution to the following constraint optimization problem,

$$\hat{L}(\theta) := \nabla L(\theta_0)(\theta - \theta_0) \to \min, \tag{2.6}$$

$$\text{subject to } ||\theta - \theta_0||_2 = \eta, \tag{2.7}$$

then there exists $\lambda_* \in \mathbb{R}$ such that

$$\theta^* = \theta_0 - \lambda_*\nabla L(\theta_0),$$

where $\lambda_* = \frac{\eta}{||\nabla L(\theta_0)||_2}$.

Proof. This constraint optimization problem can be rewritten as an unconstrained problem using the Lagrange multiplier:

$$\tilde{L}(\theta, \lambda) = \nabla L(\theta_0)(\theta - \theta_0) - \lambda(||\theta - \theta_0||_2^2 - \eta^2) \to \min, \tag{2.8}$$

where $\lambda \in \mathbb{R}$.

Then the optimal (θ^*, λ^*) satisfies that

$$\nabla\tilde{L}(\theta^*, \lambda^*) = 0.$$

Thus we have that

$$\nabla L(\theta_0) - 2\lambda^*(\theta^* - \theta_0) = 0. \tag{2.9}$$

By rearranging Equation 2.9, we have the formula for θ^* as follows:

$$\theta^* = \theta_0 + \frac{1}{2\lambda^*}\nabla L(\theta_0).$$

It is noted that as λ^* is a scalar, the optimal direction θ^* from θ_0 is along the gradient of $\nabla L(\theta_0)$.

March 15, 2021 14:50 An Introduction to Machine Learning... 9in x 6in b4055-ch02 **FA22** page 23

Table 2.2. Summary of the gradient descent (GD) method.

Goal:	Find the local optimum θ^* to minimize a continuously differentiable function L.
Algorithm:	Initialize θ_0. For $n = 1 : N_e$, $\theta_{n+1} = \theta_n - \eta\nabla L(\theta_n)$, where N_e is the maximum number of iterations and η is the learning rate.
Idea:	We construct a sequence of $\{\theta_n\}_{n\geq 0}$ such that • For some N, $(L(\theta_n))_{n\geq N}$ is a decreasing sequence, i.e., $L(\theta_N) \geq L(\theta_{N+1}) \geq \cdots$; • $\lim_n \nabla L(\theta_n) = 0$. This implies that $\{\theta_n\}_{n\geq 0}$ converges to the local minimum θ^*.

The only remaining part is to find the scalar λ^*. Equation 2.7 ensures that

$$||\theta^* - \theta_0||_2 = \frac{1}{2|\lambda^*|}||\nabla L(\theta_0)||_2 = \eta. \tag{2.10}$$

Thus it implies that $2|\lambda|^* = \frac{1}{\eta}||\nabla L(\theta_0)||_2$. Then we have that $\lambda^* = \pm\frac{1}{2\eta}||\nabla L(\theta_0)||_2$. Thus there are only two possibilities for λ^*, which is either $\frac{\eta}{2}\nabla L(\theta_0)$ or $-\frac{\eta}{2}\nabla L(\theta_0)$. It follows that

$$\hat{L}(\theta^*) = \begin{cases} \eta, & \text{if } \lambda^* = \frac{1}{2\eta}\nabla L(\theta_0); \\ -\eta, & \text{if } \lambda^* = -\frac{1}{2\eta}\nabla L(\theta_0). \end{cases} \tag{2.11}$$

Recall that the goal is to find θ^* that minimizes $\hat{L}(\theta)$. Thus it implies that $\lambda^* = -\frac{1}{2\eta}\nabla L(\theta_0)$ and

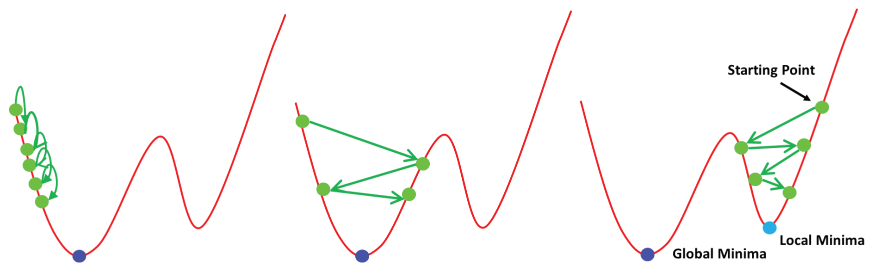
$$\lambda_* = -\frac{1}{2\lambda^*} = \frac{\eta}{||\nabla L(\theta_0)||_2}. \qquad \square$$

The summary of GD is given in Table 2.2.

2.1.3.2 *Discussion on learning rate*

The learning rate η is an important hyperparameter in the GD algorithm. A hyperparameter is a model parameter whose value is set before the learning process begins. By contrast, the parameters of the model can be trained from data, like θ. Most machine learning algorithms require hyperparameters.

Figures 2.3a and 2.3b show that there is a trade-off in the scale of the learning rate: when the learning rate is too small, the convergence of the parameters $(\theta_n)_n$ might be relatively slow; however, if the learning rate is



(a) Too small η (b) Too large η (c) Stuck in a local minimum

Figure 2.3. Effects of learning rate η.

too large, there may be the possibility that $(\theta_n)_n$ is bouncing between two valleys, which may also take a very long time to converge.

It is important to note that the GD algorithm cannot ensure a global minimum in a general setting, which makes the initialization of the parameters and learning rate important. The GD algorithm may be stuck at some local minimum, which is depicted in Figure 2.3c. In this case, a sufficiently large learning rate can help with escaping the local minimum.

2.1.3.3 *Batch gradient descent*

Batch gradient descent (BGD) is an algorithm for applying GD to minimize the empirical loss function; the update rule of BGD requires the computation of the gradient of the empirical loss function evaluated for *all the examples in the training set*. Let us recall the empirical loss function $L(\theta|\mathcal{D})$, which is usually in the additive form,

$$L(\theta|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i).$$

Thus the gradient of $L(\theta|\mathcal{D})$ with respect to θ is simply

$$\nabla_\theta L(\theta|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta Q_\theta(x_i, y_i).$$

A summary of BGD is given in Table 2.3.

Note that the gradient term is computed across all the samples in the dataset. One cycle through an entire training dataset is called an epoch. Therefore, it is often said that BGD performs model updates at the end of

Table 2.3. Summary of BGD.

Goal:	Find optimal θ such as to minimize $L(\theta\|\mathcal{D})$ in the form: $L(\theta\|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i)$.
Algorithm:	Initialize θ_0. For $n = 1 : N_e$, $\theta_{n+1} = \theta_n - \eta \underbrace{\nabla L(\theta_n\|\mathcal{D})}_{\text{Gradient term}} = \theta_n - \eta \underbrace{\frac{1}{N}\sum_{i=1}^{N} \nabla_\theta Q_{\theta_n}(x_i, y_i)}_{\text{Gradient term}}.$
Idea:	Direct application of GD to empirical loss function.

Table 2.4. Pros and cons of BGD.

Pros
• Stable convergence: BGD may require reduced model update frequency because it has a more stable error gradient at each iteration.
• The computation of the gradient term can be implemented in a parallel manner.
Cons
• A too stable error gradient may result in convergence of the model to a local minimum, which is a less optimal set of parameters.
• At the end of the training epoch the updates require the additional complexity of accumulating prediction errors across all training examples.
• BGD is usually implemented in such a way that the entire training dataset is stored in memory and available to the algorithm. Thus BGD is memory-greedy and has very slow model updates for large datasets.

each training epoch. The advantages and disadvantages of BGD are summarized in Table 2.4.[1]

2.1.3.4 ***Stochastic gradient descent***

Stochastic gradient descent, or SGD for short, is a GD-based algorithm that calculates the error and updates the model for *each example* in the training dataset. The main difference between BGD and SGD is the update rule for

[1]`https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/`.

each iteration: In SGD, for each iteration, the update of the model is based on the derivative of $L(\theta|\mathcal{D})$ w.r.t. θ evaluated at a randomly chosen sample in the training set—i.e., at each step $n \geq 1$, given θ_n, we update θ_{n+1} by

$$\theta_{n+1} = \theta_n - \eta_n \underbrace{\nabla_\theta Q_{\theta_n}(x_{i_n}, y_{i_n})}_{\text{Stochastic gradient term}},$$

where the index i_n is randomly selected from $\{1, \ldots, N\}$. The update of the model has the randomness of choosing the training example, which explains 'stochastic' in the name of SGD. SGD is also often called an online machine learning algorithm. Next let us explain the intuition behind SGD without worrying about the technical difficulty of proving its validity. Recall the empirical loss $L(\theta|\mathcal{D})$, which satisfies that

$$L(\theta|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i). \tag{2.12}$$

SGD randomly chooses a sample from the dataset to calculate the gradient. Suppose that i_n are iid with uniform distribution, where $n \in \{1, 2, \ldots, N\}$. Then it follows that

$$\mathbb{E}_{x,y}[\nabla_\theta Q_\theta(x_{i_n}, y_{i_n})] = \nabla L(\theta|\mathcal{D}), \tag{2.13}$$

where (x_{i_n}, y_{i_n}) is sampled from the empirical distribution of $(x_i, y_i)_{i=1}^N$. Or alternatively we can sample (i_n) randomly from $\{1, \ldots, N\}$ without replacement. Equation (2.13) still holds. It implies that although for each iteration the stochastic gradient term is not $\nabla L(\theta|\mathcal{D})$, its expectation coincides with $\nabla L(\theta|\mathcal{D})$. As the number of the maximum iteration N_e tends to infinity, it is reasonable to expect that the limit of θ_n by SGD converges to the local minimum as that of GBD, which shares the spirit of Monte Carlo methods.

However, to make the SGD algorithm work, we need to adjust the learning rate by choosing a suitable decreasing step-size sequence $\{\eta_n\}_n$ instead of the constant learning rate η in BGD. The reason for this is that the limit of the stochastic gradient term cannot converge to zero if there is a gradient evaluated for at least one sample that is non-zero. However, to ensure the convergence of θ_n, we have to make the sequence of η_n converge to zero. This explains why in SGD, the learning rate η_n needs to be reduced gradually. A summary of SGD is given in Table 2.5.

Table 2.5. Summary of SGD.

Goal:	Find optimal θ such as to minimize $L(\theta\|\mathcal{D})$ in the form: $L(\theta\|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i)$.
Algorithm:	Initialize θ_0. For $n = 1 : N_e$, Randomly choose the index i_n from $\{1, \cdots, N\}$, $\theta_{n+1} = \theta_n - \eta_n \underbrace{\nabla_\theta Q_{\theta_n}(x_{i_n}, y_{i_n})}_{\text{Stochastic gradient term}}$, for a suitably chosen decreasing step-size sequence $\{\eta_n\}_n$.
Idea:	$\mathbb{E}_{x,y}[\nabla_\theta Q_\theta(x_{i_n}, y_{i_n})] = \nabla L(\theta\|\mathcal{D})$, where (x_{i_n}, y_{i_n}) is sampled from the empirical distribution of $(x_i, y_i)_{i=1}^N$ or randomly sampled without replacement.

Table 2.6. Pros and cons of SGD.

Pros

- The increased model update frequency may result in faster learning on some problems.
- Noisy gradient updates can avoid the premature convergence of the model to local minima.

Cons

- Updating the model so frequently is more computationally expensive than other configurations of gradient descent. SGD may take significantly longer to train models on large datasets.
- The frequent updates can result in a noisy gradient signal, which may cause the model parameter updates have a higher variance over training epochs and in turn make the model error more oscillatory.
- The unstable estimate of the error gradient can also make it difficult for the algorithm to settle on an error minimum for the model.

Let us summarize the benefits and downsides of SGD in Table 2.6.[2]

To sum up, SGD is very quick to evaluate each iteration. Randomness helps to escape a local minimum, but it makes the settling of the minimum difficult.

[2]`https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/`.

2.1.3.5 *Mini-batch gradient descent*

Mini-batch gradient descent (mini-batch GD) is another variant of the gradient descent algorithm, which splits the training dataset into small batches that are used to calculate model error and update model coefficients. Mini-batch GD can be viewed as a combination of BGD and SGD.

At one iteration, instead of going over all examples, mini-batch GD updates the gradients based on a subset of samples (called mini-batches) for the given batch size b. When $b = 1$, mini-batch GD is SGD; when $b = N$, mini-batch GD is BGD. In mini-batch GD, the typical method of creating mini-batches includes two steps:

(1) Shuffle the dataset to avoid the existing order of samples.
(2) Split the entire training data set into several non-overlapping mini-batches of batch size b; if the sample size is not divisible by the batch size, the remaining samples will be their own batch.

Then we apply batch gradient descent for each mini-batch until all the samples have been processed (this is called one epoch); we repeat this procedure until the number of epochs reaches the maximum epoch number N_e.

Implementations may take average of the gradient, which further reduces the variance of the gradient. Mini-batch gradient descent aims to strike a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. In the field of deep learning (Chapter 5), mini-batch GD is the most common optimization method used to estimate the optimal model parameters.

A summary of mini-batch GD is given in Table 2.7.

Table 2.7. Summary of mini-batch GD.

Goal:	Find optimal θ such as to minimize $L(\theta\|\mathcal{D})$ in the form: $L(\theta\|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} Q_\theta(x_i, y_i)$.
Algorithm:	Initialize θ_0. For $n = 1 : N_e$, Randomly partition the dataset $\mathcal{D}$ into $N_b = \frac{N}{b}$ mini-batches of size b, denoted by $(B_i)_{i=1}^{N_b}$. For $j = 1 : N_b$, $$\theta_{n+1} = \theta_n - \underbrace{\eta_n \frac{1}{b} \sum_{(x,y)\in B_j} \nabla_\theta Q_{\theta_n}(x, y)}_{\text{Stochastic gradient term}},$$ where $\{\eta_n\}_n$ is a suitably chosen decreasing sequence.
Idea:	Combining SGD and BGD.

Table 2.8. Pros and cons of mini-batch GD.

Pros
• The model update frequency is higher than batch gradient descent, which allows for a more robust convergence and avoids local minima.
• The mini-batch updates provide a computationally more efficient process than SGD.
• The mini-batch algorithm allows a balance of both the efficiency of not having all training data in memory and algorithm implementations.
Cons
• Mini-batch requires an additional "mini-batch size" hyperparameter for the learning algorithm. It may increase the computation cost as this hyper-parameter needs to be tuned in practice.
• Error information must be accumulated across mini-batches of training examples, as for batch gradient descent.

The advantages and disadvantages of mini-batch GD are listed in Table 2.8.[3]

2.1.3.6 *Comparison of three types of gradient descent*

In the previous subsections, we have discussed three gradient descent methods, i.e.,

- Batch GD;
- Stochastic GD;
- Mini-batch GD.

They vary in terms of the number of training samples used to calculate empirical loss and to update the model. A summary of the comparison between the above three methods is provided in Table 2.9. We can see that there is a trade-off between the computational efficiency of gradient descent configurations and the accuracy of gradient updates. Figure 2.4 depicts the typical trajectory of parameter sequence for these three methods. More optimization methods can be found at the following website.[4]

[3]`https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/`.
[4]`http://ruder.io/optimizing-gradient-descent/`.

Table 2.9. Comparison of various GD based methods.

	BGD	SGD	Mini-batch GD
Update frequency	Low	High	Medium
Update complexity	High	Low	Medium
Fidelity of error gradient	High	Low	Medium
Stuck in local minimum	Easy	Difficult	Difficult
Easy to converge	Yes	No	No

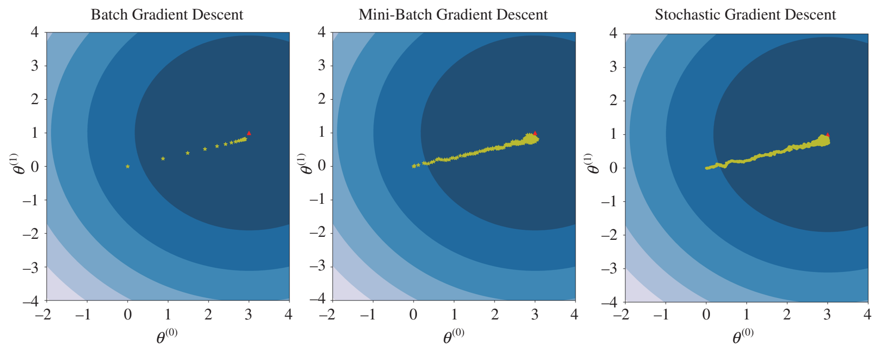


Figure 2.4. Convergence of parameters $(\theta_n)_n$ for BGD, mini-batch GD and SGD.

2.1.4 Prediction and validation

There are various ways of judging goodness of fit, which can be mainly divided into two types:

- statistics-based approach;
- machine learning-based approach.

2.1.4.1 *Statistics-based approach*

For statistics based validation, we usually need to make extra probabilistic assumptions of the residuals. Hypothesis testing is a hypothesis that is testable on the basis of observing a process that is modeled via a set of random variables—e.g., p-value, R^2, R^2_{adj}.

- ***p*-value:** Under the null hypothesis, the probability that the statistical summary is equal to or more extreme than the observed one. A smaller p-value indicates rejecting the null hypothesis. However, it does not measure the probability of making mistakes by rejecting a true null hypothesis (a Type I error).

- $\boldsymbol{R^2}$**:** The proportion of the variance in the output variable that is predictable from input variables, also called the coefficient of determination.

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - x_i^T\hat{\beta})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{2.14}$$

  where $\hat{\beta}$ is the optimal parameter in the linear model.
- **Adjusted** $\boldsymbol{R^2}$**:** A similar concept to R^2 that takes the numbers of model parameters (input dimension) into account. It is defined in the following form, which penalizes larger input dimensions:

$$R_{adj}^2 = 1 - (1 - R^2)\frac{N-1}{N-d-1}, \tag{2.15}$$

  where $\hat{\beta}$ is the optimal parameter in the linear model and d is the input dimension.

2.1.4.2 *Machine learning-based approach*

Machine learning-based validation focuses mainly on predictive power on an unseen new dataset, which is the generalization ability of the fitting model. To achieve this, one usually divides the dataset into a training dataset and a testing dataset. The model is calibrated using the training dataset, and the goodness of fit is computed for both the training set and testing set.

Perfect fitting on the training set is usually not a good thing because typically, the training set contains some random noise; this noise should be filtered out to give good generalization. For example, if you choose an over-complicated model which includes too many parameters, the model might have a perfect performance on the training set, as it mistakenly regards the noise as part of the signal of the model. This model may then not perform well on the testing set as there will be new, different noise, leading to little predictive power. This kind of problem is called the *overfitting* issue as the model overfits the training data. This is a very common but important problem in the training process.

On the other hand, if you choose a simple model, it may be not rich enough to describe the complex relationship between inputs and outputs. This is called the *underfitting* issue. An illustration of fitting issues in the training process is shown in Figure 2.5.

Commonly used indicators for goodness of fit include the mean squared error (MSE), R^2 and R_{adj}^2.

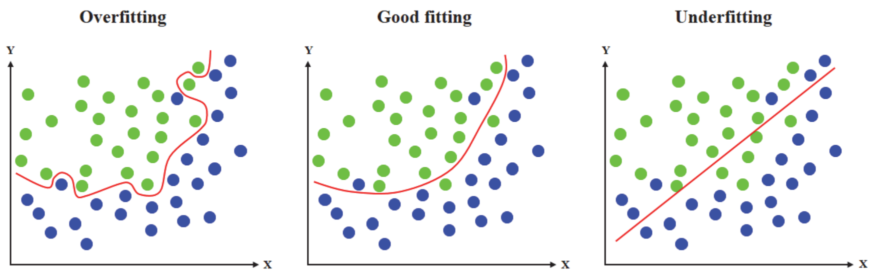


Figure 2.5. Potential fitting issues in the training process.

2.1.4.3 *Cross-validation and parameter tuning*

As we discussed earlier, the ultimate goal of supervised learning is to train a model using labeled data, which can be generalized to an unseen dataset. Thus the evaluation of the predictive power of a model is crucial. This is typically assessed by *cross-validation* in practice.

The main idea of model assessment is to further split the training data into two parts, i.e., a subset of training data for training the model and a validation set to assess the predictive power of the trained model (without touching the test data). We choose the model that achieves the best performance measure in the validation set as the final model and use this model to predict in the testing set. However, this may drastically reduce the size of the training set. A solution to this problem, called k-fold cross-validation (Figure 2.6), is to split the training set into k subsets ("folds") and conduct the following procedures:

(1) Train a model using $(k - 1)$ folds of the training data.
(2) Use the remaining fold for the validation to compute the performance measurement (e.g., MSE) of this model.

For one model with a given set of hyperparameters, we conduct k-fold cross-validation, and the average performance measure of the k folds can be used as a scalar score to measure its performance. This can be combined with *grid search* to select optimal hyperparameters. As its name suggests, grid search is an exhaustive search method of choosing the best hyperparameters. It requires pre-specifying possible values of a hyperparameter (grid) and choosing the best one based on the corresponding cross-validation scores. Lastly, once the optimal hyperparameters are chosen, we refine

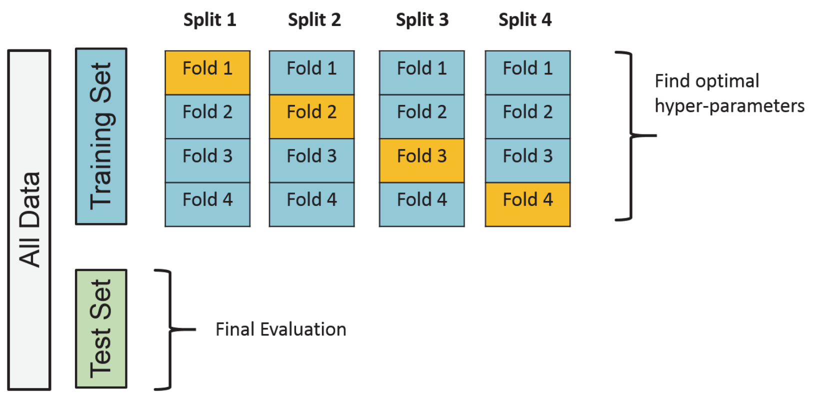


Figure 2.6. 4-fold cross-validation and parameter tuning.

the model using the whole training set and make the prediction in the testing set.[5]

Note that cross-validation and grid search are standard methods for the performance measurement and parameter tuning of both regressors and classifiers. Thus in the next section on validation of classification, we skip the discussion on cross-validation and grid search.

2.2 From Regression to Classification

2.2.1 Categorical output

The setup for classification is very similar to that of regression problems. Given a set of input–output pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, we aim to infer the functional relationship between an input x and an output y. But classification differs from regression mainly because the output variable of the classification is categorical. In other words, there are only finite many possible values of y_i, denoted by $\mathcal{Y}$. W.l.o.g., $\mathcal{Y} = \{1, \cdots, n_o\}$, where n_o denotes the number of possible categories. According to the different numbers of possible categories, classification can be divided into binary classification ($n_o = 2$) and multi-class classification ($n_o > 2$).

Categorical variables represent a qualitative method of scoring data (i.e., they represent categories or group membership). For example, the

[5]Interested readers may refer to `https://scikit-learn.org/stable/modules/cross_validation.html` for more details of cross-validation and its implementation in the Scikit-Learn package.

Table 2.10. Different encoding methods for the blood type example.

Blood type	A	B	AB	O
Integer encoding	1	2	3	4
One-hot vector encoding	0001	0010	0100	1000

blood type of a person may be A, B, AB or O, which is a categorical variable. There are several ways to represent categorical variables numerically, including

- integer encoding (the i^{th} class is represented using an integer i);
- one-hot vector encoding (the i^{th} class is represented using a binary vector of length n_o, which has the unique non-zero element at the i^{th} position).

Let us revisit the example of blood type. The numerical representation of blood types using two above encoding methods are given in Table 2.10.

2.2.2 Model

The objective of classification is to predict the corresponding output for any given new input x_*, just like the regression problem. However, due to the categorical nature of the output, this question has a slightly different mathematical formulation from that of regression problems. In the classification problem, instead of predicting the output y directly, we aim to estimate the probability of the output being y conditional on an input x, which is described by a model $f_\theta : E \to \mathbb{R}^{n_o}$. Intuitively, we have that

$$\langle f_\theta(x), \bar{y} \rangle \approx \mathbb{P}[y|x],$$

where $\bar{y}$ is one-hot encoding of the class y, and $\langle ., . \rangle$ is the inner product of two vectors of length n_o.

Let us first understand why we do not aim to predict the conditional expectation of the output as we do in regression. This is because the conditional expectation of a categorical output does not make sense. For example, if the conditional distribution of the output label is known as a discrete random variable with probability $(0.4, 0.2, 0.4)$, the conditional mean of the output depends on the numeric representation of the output category. More importantly, in classification, one can't infer the best estimator for the output category given the input based on only this conditional mean of output. Therefore we usually estimate the conditional probability of each class label.

2.2.3 Loss function and optimization

In contrast to the quadratic loss function in regression, the cross entropy loss function is commonly used in classification as it provides a way to quantify the difference between the empirical conditional distribution of output y given the input x and the model estimated conditional distribution $f_\theta(x)$. For discrete probability distributions p and q with the same support $\mathcal{Y}$, the cross entropy is defined to be

$$H(p,q) := -\sum_{j\in\mathcal{Y}} p(j)\log(q(j)).$$

For a given distribution p, H is a function of q, and it attains its smallest value when $q = p$. Intuitively, smaller cross entropy $H(p,q)$ means that two distributions are similar. In other words, when minimizing the cross entropy H, the optimal distribution of q is the same as that of p.

Definition 2.3 (Cross Entropy Loss Function). The cross entropy loss function $Q_\theta : E \times \mathcal{Y} \to \mathbb{R}$ is defined to be

$$Q_\theta(x,y) = -\langle y, \log f_\theta(x)\rangle,$$

where $x \in E$, y is one-hot encoding in $\mathcal{Y}$, θ are model parameters of f_θ, and $\langle .,. \rangle$ is the inner product.

The corresponding empirical cross entropy loss function is given as the average of the above cross entropy loss function evaluated at all samples:

$$L(\theta|\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N}\langle y_i, \log f_\theta(x_i)\rangle.$$

Another way to interpret the cross entropy is through maximum likelihood estimation (MLE). The cross entropy loss function can be regarded as the negative log-likelihood function of θ, given the observation of the input–output pairs. Assuming all the samples are mutually independent, the likelihood function is given as the product of the conditional probability of the output, i.e.,

$$\prod_{i=1}^{N}\langle f_\theta(x_i), \bar{y}_i\rangle \to \max,$$

which is equivalent to minimizing the negative log-likelihood ratio, i.e.,

$$-\sum_{i=1}^{N}(\langle \log f_\theta(x_i), \bar{y}_i\rangle) \to \min.$$

That is exactly the cross entropy loss function, denoted by $L(\theta|\mathcal{D})$ (see Definition 2.3). Thus it is noted that the optimal parameters from minimizing the cross entropy are the same as those obtained by maximizing the likelihood. The cross entropy empirical loss has an additive form, which allows parallel computation benefit for each sample.

In the next stage of the optimization to find the optimal parameter θ^* to minimize $L(\theta|\mathcal{D})$, we usually make the further assumption that f_θ is differentiable w.r.t. θ. The numerical optimization methods we have discussed in Section 2.1.3 can be exploited here as well.

2.2.4 Prediction and validation

Once we obtain the optimal parameters θ^*, the prediction is straightforward. For any new input data x_*, use the output label with the highest estimated conditional probability as the estimator for the output,

$$\hat{y}_* = \arg\max_{i \in \mathcal{Y}} f^{(i)}_{\theta^*}(x_*),$$

where $f^{(i)}_{\theta^*}(x_*)$ is the i^{th} coordinate of $f_{\theta^*}(x_*)$.

At the final stage, we need to specify the metric of the goodness of fit. There are various performance measures, e.g., the accuracy, the confusion matrix, etc.

2.2.4.1 *Accuracy*

In classification, the dimension of the model output $f_{\theta^*}(x)$ is n_o, which represents the estimated conditional probability of each output. Let $\hat{Y}_{\text{prob}}$ denote the matrix of size (N, n_o),

$$\hat{Y}_{\text{prob}} = (f_{\theta^*}(x_i))_{i \in \{1,2,\ldots,N\}}. \tag{2.16}$$

For multi-class classification, the accuracy is one of the most popular measures and is defined as follows:

$$\sum_{i=1}^{N} \frac{\mathbf{1}(\hat{y}_i = y_i)}{N},$$

where $i \in \{1, 2, \cdots, N\}$, and y_i and $\hat{y}_i$ denote the actual output and the estimated output of the i^{th} sample, respectively.

2.2.4.2 *Confusion matrix*

Another way to measure the performance of a classifier is the confusion matrix. The column represents the estimated label for the classification problem and the row represents the true label. Let $M := (M_{i,j})_{i,j\in\mathcal{Y}}$ denote the confusion matrix, where $M_{i,j}$ denotes the number of samples with true label i and estimated label j. The better the prediction, the more diagonally dominant the confusion matrix M is.

The normalized confusion matrix is defined from the confusion matrix and denoted by $\hat{M} = (\hat{M}_{i,j})_{i,j\in\mathcal{Y}}$, where $\hat{M}_{i,j}$ is defined as follows:

$$\hat{M}_{i,j} = \frac{M_{i,j}}{\sum_{j\in\mathcal{Y}} M_{i,j}}.$$

$\hat{M}_{i,j}$ represents the empirical conditional probability of the sample being identified as j when it in fact belongs to class i. The better the prediction, the closer $\hat{M}$ is to the identity matrix.

2.2.4.3 *Other metrics for binary classification*

You may wonder why we need other metrics than accuracy to assess classification performance. When the data are extremely imbalanced, the trivial classifier (estimating all samples as the majority class) gives very high accuracy, which implies that accuracy is not an informative performance measure in this case. Next, we introduce some other commonly used metrics for the binary classification case: precision, recall, PR curve and ROC curve.

As shown in Figure 2.7, the confusion matrix of a binary classifier $M = (M_{j_1,j_2})_{j_1,j_2\in\{1,2\}}$ is a 2×2 matrix, where

- True Positive (TP, $M_{2,2}$): the number of samples that have actual label class 2 and predicted label class 2.
- False Positive (FP, $M_{1,2}$): the number of samples that have actual label class 1 and predicted label class 2.

Predicted Label / Actual Label	Class 1 (Negative)	Class 2 (Positive)
Class 1(Negative)	$M_{1,1}$ (TN)	$M_{1,2}$(FP)
Class 2(Positive)	$M_{2,1}$ (FN)	$M_{2,2}$(TP)

Figure 2.7. Confusion matrix of a binary classifier.

- True Negative (TN, $M_{1,1}$): the number of samples that have actual label class 1 and predicted label class 1.
- False Negative (FN, $M_{2,1}$): the number of samples that have actual label class 2 and predicted label class 1.

The *precision* of a binary classifier is defined as the percentage of true positive samples among all the samples with a predicted label of "positive":

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

The *recall*, also called the *sensitivity* or *true positive ratio* (TPR), is defined as the percentage of true positive samples among all the samples with an actual label of "positive":

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

From the above definition, we can see that one trivial way to get a high recall is to predict all the samples being "positive," which gives perfect recall of 100%. Thus the recall is typically used accompanied by the precision. A higher recall suggests a larger TP, which indicates a better performance of the classifier. But increasing the recall reduces the precision and vice versa. This is called the precision and recall trade-off.

Next, let us introduce the *precision–recall curve* (PR curve) based on the above concepts of precision and recall. For a classifier, f_{θ^*} gives an estimated probability (also called a score) to each possible output class. Instead of choosing the class label that gives the maximum score, alternatively for each given threshold value t, we assign the estimator for the output using the following equation:

$$\hat{y} = \begin{cases} 1, & \text{if } f_{\theta^*}(x) > t; \\ 2, & \text{if } f_{\theta^*}(x) \leq t. \end{cases} \tag{2.17}$$

Varying the threshold t, the corresponding precision and recall can be computed, and thus the PR curve is obtained.

The *receiver operating characteristic curve* (ROC curve) is another important metric of a binary classifier. Similar to the PR curve, varying the threshold t, the ROC curve is the curve of TP against FP. AUC stands for "area under the ROC curve." It measures the entire two-dimensional area enclosed by the ROC curve, a line from $(0, 0)$ to $(1, 0)$ and a line from $(1, 0)$ to $(1, 1)$.

2.2.4.4 *Numerical example*

In the following, we use the binary classification of identifying whether a digit image is a number 8 as a concrete example to show how to compute all the metrics we have discussed and implemented it using Scikit-Learn.

We use the MNIST dataset composed of digit images of the numbers 0–9.[6] The input data is a gray-valued image, and the output is the digit in the input image. Now we want to identify whether an input image is a digit 8 and construct a binary classifier where class 1 represents "non-8 digit" while class 2 represents "8 digit." The training dataset contains 60,000 handwritten digit images, including 54,149 non-8 digit samples and 5851 8 digit samples. It is easy to see that there are many more negative class cases than those of the positive class. Thus this is a class imbalance problem. Figure 2.8(left) shows the estimated output (score) of the first 15 samples, and each row has the sum 1. The second column represents the estimated probability of the class label being 2. The estimated class label is the label with the maximum probability, and Figure 2.8(right) provides the estimated output label.

conditional probability estimator		
	class 1 (non 8)	class 2 (digit 8)
0	0.999929	0.000071
1	0.999096	0.000904
2	0.999970	0.000030
3	0.877981	0.122019
4	0.999048	0.000952
5	0.982514	0.017486
6	0.993879	0.006121
7	0.918145	0.081855
8	0.962651	0.037349
9	0.993038	0.006962
10	0.995717	0.004283
11	0.345118	0.654882
12	0.999995	0.000005
13	0.999986	0.000014
14	0.931200	0.068800

estimated output	
	class 1 (non 8)
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False
10	False
11	True
12	False
13	False
14	False

Figure 2.8. The estimated output of the first 15 samples.

[6] `http://yann.lecun.com/exdb/mnist/`.

Code:

```python
from sklearn.metrics import confusion_matrix
# Y_test is a binary vector of the actual class label with dim (N, 1) where
↪  N is the number of samples;
# y_test_est is a binary vector of the estimated class label with dim (N,
↪  1).

cm = confusion_matrix(Y_test, y_test_est)
print('confusion matrix is {}'.format(cm))
```

Screen Output:
confusion matrix is

$$\begin{pmatrix} 8810, & 216 \\ 319, & 655 \end{pmatrix}.$$

Figure 2.9. Python code for computing the confusion matrix and the corresponding result.

We first compute the confusion matrix on the test set using confusion_matrix() in the Scikit-Learn package as shown in Figure 2.9.

Based on the confusion matrix, we can compute the corresponding accuracy via

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{TN} + \text{FP}} = \frac{8810 + 655}{10000} = 0.9465.$$

You may also use accuracy_score() in sklearn.metrics to compute the accuracy of $\hat{Y}$:

```python
from sklearn.metrics import accuracy_score
acc = accuracy_score(Y_test, y_test_est)
```

The accuracy is about 94.65%, which seems very good. But if one computes the precision and recall, the prediction is not that great.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{655}{216 + 655} = 0.7520$$
$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{655}{319 + 655} = 0.6725.$$

Similar to the accuracy case, you may use the following Python function to compute the precision and recall:

Code:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(Y_test,
↪  y_test_prob_est[:,1])
plt.plot(precisions, recalls, 'b')
plt.xlabel('precision', fontsize=14)
plt.ylabel('recall', fontsize=14)
plt.title('PR Curve')
plt.axis([0, 1, 0, 1])
```

Screen Output:

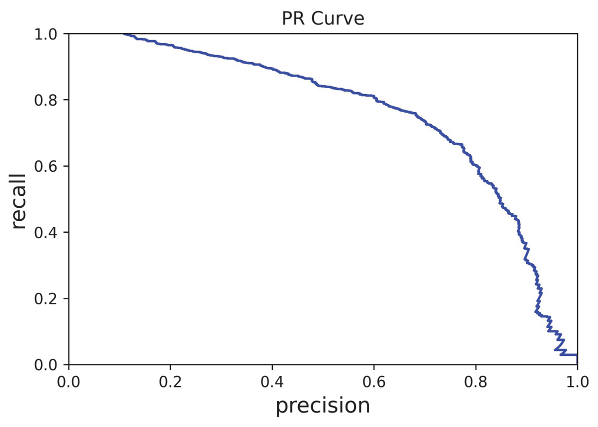


Figure 2.10. The code for the PR curve plot and the screen output.

```
from sklearn.metrics import precision_score, recall_score
precision = precision_score(Y_test, Y_test_est)
recall = recall_score(Y_test, y_test_est)
```

Figures 2.10 and 2.11 provide the code for computing the PR curve and ROC curve obtained by a binary classification task using the Scikit-Learn Python package. In this example, the AUC score is 0.9423, which is the area of the blue shaded region enclosed under the ROC curve.

Remark 2.1. When the positive class has much fewer samples than the negative class and the false positives are more important, one should choose the PR curve. For example, looking at the previous ROC curve and the AUC score, you may think that the classifier is really good. But this is mostly because there are few positives compared to the negatives.

Code:

```python
from sklearn.metrics import roc_curve, roc_auc_score
fps, tps, thresholds = roc_curve(Y_test, y_test_prob_est[:,1])
roc_auc_score_train = roc_auc_score(Y_test, y_test_prob_est[:,1])
import matplotlib.pyplot as plt
plt.plot(fps, tps, 'b')
plt.xlabel('false positive rate', fontsize=14)
plt.ylabel('true positive rate', fontsize=14)
plt.title('ROC Curve')
plt.axis([0, 1, 0, 1])
plt.fill_between(fps, 0, tps, facecolor='lightblue', alpha=0.5)
plt.text( 0.5, 0.8, 'roc auc score = '+str(round(roc_auc_score_train, 4)),
↪  fontsize=14)
plt.annotate("",
            xy=(0.3, 0.7), xycoords='data',
            xytext=(0.5, 0.8), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),)
```

Screen Output:

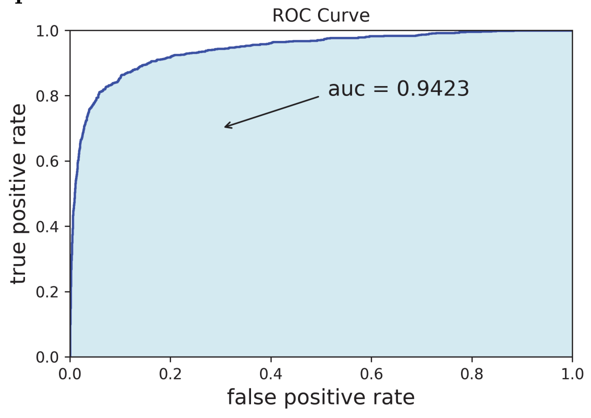


Figure 2.11. The ROC curve.

In conclusion, Table 2.11 provides a summary of the general framework of classification.

2.3 Model Ensemble

As the saying goes, two heads are better than one. There exists a similar principle in machine learning. One may wonder whether aggregating

Table 2.11. The framework of classification.

Dataset:	$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.
Model:	$f_\theta(x, y) \approx P(y\|x)$, $\forall x \in \mathbb{R}^d$, $y \in \mathcal{Y}$.
Empirical Loss:	$L(\theta\|\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N} \log(f_\theta(x_i, y_i)) \to \min$.
Optimization:	$\theta^* = \arg\min_{\theta}(L(\theta\|\mathcal{D}))$.
Prediction:	$\hat{y}_* = \arg\max_{y \in \mathcal{Y}} f_{\theta^*}(x_*, y)$.
Validation:	Accuracy, confusion matrix, etc.

different predictors can have better prediction performance than could be obtained from any of the constituent learning algorithms alone. The answer is yes for most cases. Ensemble learning is devoted to addressing this question.

2.3.1 Intuition of ensemble

An ensemble is nothing other than a collection of predictors that are combined together (e.g., the majority of all predictions) to give a final prediction. The reason that we use the ensemble method is that one can incorporate many predictors of the same output variable to improve the prediction performance over that of any single predictor.

We use the following simple numerical example to illustrate the idea behind ensemble methods. Assume that there is a binary classification problem with all sample labels being 2. Now say we only have a classifier with 55% accuracy; this means that it predicts correct class labels with probability 0.55. We simulate this classifier in Listing 2.1. Obviously, it is a weak learner as it only performs a little bit better than random guessing. So what should we do to improve the performance without the help of new learners? The ensemble method can help us out. We can simply combine multiple (e.g., 1000) identical weak learners and use majority voting to decide the estimated class for the 1000 learners. If most learners return 2, then the estimated class of the ensemble model is correct. We simulate 10,000 samples, and the accuracy of one weak learner is 55%, which is close to the setting of the problem. We assume that the weak learners are mutually independent. You will find that an ensemble model with 1000 weak learners should achieve nearly 100% accuracy, which is an amazing result. The accuracy of ensemble models with different numbers of learners is depicted in Figure 2.12. The accuracy gradually increases with an increasing number of learners.

```python
# importing mean()
from statistics import mean
def weak_learner():
    n = np.random.randint(0, 100)
    return 1 if n >= 45 else 0

# Majority voting method
def majority_voting(results:list):
    return 1 if results.count(1) > results.count(0) else 0

# Define ensemble model with 1000 weak learners
def ensemble_model(learner, num_learners = 1000):
    all_results = [learner() for i in range(num_learners)]
    return majority_voting(all_results)

# Simulate 10,000 samples to approximate the accuracy
num_samples = 10000
all_weak_learner_results = []
all_ensemble_model_results = []
for i in range(num_samples):
    weak_learner_result = weak_learner()
    ensemble_model_result = ensemble_model(weak_learner)
    all_weak_learner_results.append(weak_learner_result)
    all_ensemble_model_results.append(ensemble_model_result)

print('The weak learner only achieves accuracy
↪  of',mean(all_weak_learner_results))
print('The ensemble model achieves accuracy as high as',
↪  mean(all_ensemble_model_results))
```

Listing 2.1. Python code for a numerical example of ensemble methods.

As we can see from the above example, the ensemble method turns the weak learner into a strong learner, and the accuracy improves sharply. In practice, we have to face more complicated problems, e.g., multi-classification and regression problems. Though the ensemble method may not perform as amazingly as in the above example, it is the most popular method for model selection.

2.3.2 Homogeneous weak learners ensemble

In the following, we divide model ensemble methods into two types based on types of weak learners:

- Homogeneous weak learners (the base models have the same type):
  - Bagging/Pasting
  - Boosting

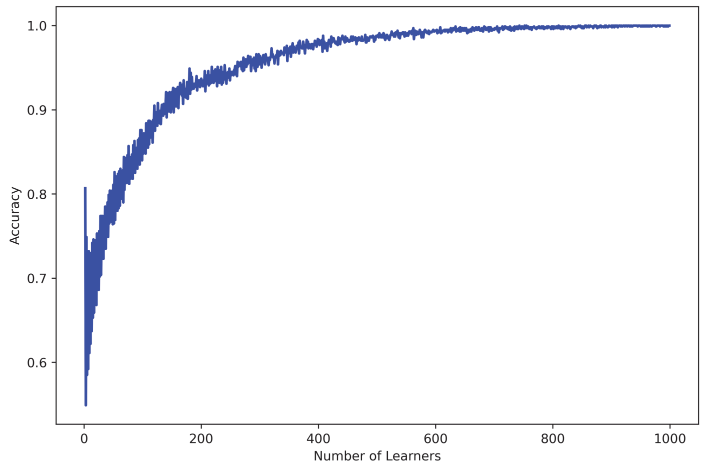


Figure 2.12. The accuracy of ensemble models with different numbers of learners.

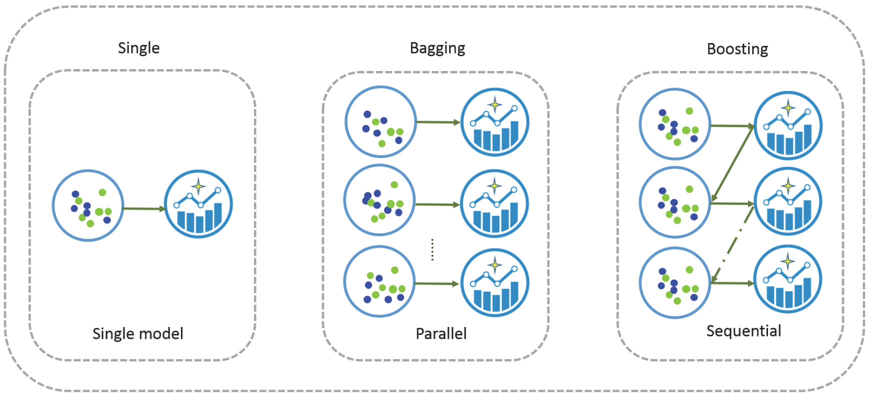


Figure 2.13. Homogeneous weak learners ensemble methods.

- Heterogeneous weak learners (the base models may have different types):
  – Stacking

As we can see in Figure 2.13, ensemble techniques of homogeneous weak learners are further classified into the following main types:

(1) To use the same training algorithm for predictors, but each time a subset of samples are randomly selected for training. In this case, we typically combine predictors using some model averaging techniques, e.g., weighted average, majority vote or normal average.

(2) To combine the predictors into the final predictor in a sequential manner, which is called boosting.

The first type of ensemble method usually involves aggregating many uncorrelated learners, which reduces error by reducing variance. Under this category, Bagging (short for bootstrap aggregating) [Breiman (1996)] and Pasting [Breiman (1999)] are the two major sub-classes. For Bagging, each observation is chosen with replacement to be used as input for each of the model. In contrast, for Pasting, each time a subset of data is randomly selected without replacement.

In the following, we focus on out-of-folds (OOF), which is another model ensemble method that falls into this category. OOF refers to a step in the learning process when using k-fold cross-validation in which the predictions from each set of folds are grouped into predictions of the training set. These predictions are now "out-of-folds," and thus the error can be calculated on these to get a good measure of how good your model is. As shown in Figure 2.14, the procedure is composed of the following steps, and the algorithm is outlined in Algorithm 1.

(1) Split the dataset into the training and testing set.
(2) Use stratified k-fold cross-validation in the training set and thus obtain k estimated models.
(3) Evaluate each estimated model on the testing set and therefore have k estimators of the testing data.

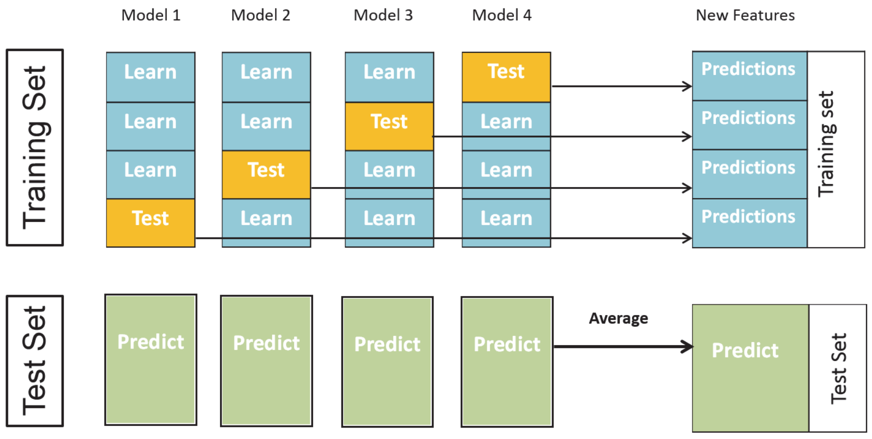


Figure 2.14. Illustration of OOF prediction procedure.

Algorithm 1: OOF Prediction Algorithm

```
1: Input: $\mathcal{D} = (X_{train}, Y_{train})$, $X_{test}$, $K$
2: Split the training set into $K$ folds, denoted as $\mathcal{D}_1, \mathcal{D}, \ldots, \mathcal{D}_K$;
3: Set $\hat{Y}_{test} = 0$;
4: for $i = 1 : K$ do
5:     Train the model in $\mathcal{D}/\mathcal{D}_i$ and obtain a model $T_i$;
6:     Calculate the predictor of the testing data using model $T_i$, denoted
       by $\hat{Y}^{(i)}$;
7:
           $$\hat{Y}_{test} = \hat{Y}_{test} + \hat{Y}^{(i)}.$$
8: end for
9: The final estimator of $\hat{Y}_{test}$ is given as follows:
           $$\hat{Y}_{test} = \frac{\hat{Y}_{test}}{K}.$$
10: Output: $\hat{Y}_{test}$.
```

(4) Average the k estimators of the testing data to get the final estimator of the testing data.

For the second type of ensemble method, the core idea of boosting is to update subsequent predictors based on the error of the previous predictors. Because new predictors are updated from learning mistakes by previous predictors, it takes fewer iterations to get close to ground-truth predictions. But the stopping criterion is essential in this case. If it is not appropriately chosen, it could easily lead to overfitting on training data.

Gradient boosting is an example of a boosting algorithm.[7] We devote the rest of this subsection to explain the gradient boosting method. It is a variant of the gradient descent algorithm that provides a way to combine weaker learners to get better estimation for the gradients and construct a final learner in order to provide better prediction.

Recall that the objective is to minimize the loss function $L(\theta|(X, Y))$. In gradient boosting, the weak learner (h_m) is used as a base model, and then a sequence of predictors $(f_m)_{m=1}^{M}$ is constructed, where the updating

[7]`https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d`.

rule of f_m is given as follows:

$$f_m = f_{m-1} + \gamma_m h_m, \tag{2.18}$$

where $h_m \in \mathcal{H}$, which is the set of base models, and $\gamma_m \in \mathbb{R}$ is a constant. In this case, f_m is an additive model: when m is increased by 1, the parameter of the model h_m is added to the parameters of f_m.

Compare the weight update rule of the gradient descent algorithm and Equation (2.18): h_m should serves as the gradient term $\nabla L(\theta|(X, Y)) := \nabla L(Y, f_M(X))$, where θ is the set of all parameters of f_M. However, at the m^{th} iteration, we cannot evaluate $\nabla L(\theta|(X, Y))$ as $(h_j)_{j=m}^{M}$ are unknown. It is natural to use $\nabla L(Y, f_{m-1}(X))$ to approximate the actual gradient. But $\nabla L(Y, f_{m-1}(X))$ may be noisy and it may not belong to any base model. Therefore this suggests using the base model h_m to fit the derivative terms $\nabla L(Y, f_{m-1}(X))$. Thus the update rule of the gradient boosting algorithm is proposed: at each m^{th} iteration we update f_m using

$$f_m(x) = f_{m-1}(x) - \gamma_m \nabla_{f_{m-1}} L(y, f_{m-1}(x)). \tag{2.19}$$

γ_m can be chosen by solving the following one dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} L(y - f_{m-1}(x) - \gamma \nabla_{f_{m-1}} L(y, f_{m-1}(x)).$$

Then the gradient boosting algorithm is given in Algorithm 2.

Let us consider the regression problem, which aims to minimize the quadratic loss function $L(\theta|(X, Y))$. Then the derivative term can be simplified to residuals as follows:

$$\nabla_f L(y, f(x)) = 2(y - f(x)).$$

In this case, h_m can be viewed as correcting the error terms by learning the residuals of the previous estimator f_{m-1}.

2.3.3 Heterogeneous weak learners ensemble

Stacking is a heterogeneous ensemble method to build a meta-model using predictors from various models. The main idea is to use the predictors of each model as new inputs and learn the relationship between the model predictors and the output.

As shown in Figure 2.15, the procedure of this type of model stack is outlined as follows:

(1) Split the dataset into the training and testing set.

Algorithm 2: Gradient Boosting Algorithm

```
1:  Input: $(x_i, y_i)_{i=1}^N$.
2:  Initialize $f_0$ by a constant $\gamma_0$ via the following equation:
        $\gamma_0 = \arg\min_{\gamma} L(y, \gamma);$
3:  for $m = 1 : M$ do
4:      for $i = 1 : N$ do
5:          Compute the residuals
                $r_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$
6:      end for
7:      Fit a base model learner $h_m$ to the target $r_{im}$, using the data
        $(x_i, r_{im})_{i=1}^n$.
8:      Solve the one dimensional optimization problem
            $\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i)).$
9:      Update $f_m$ using the following formula:
            $f_m(x) = f_{m-1}(x) + r_m h_m(x).$
10: end for
11: Output: $f_M$.
```

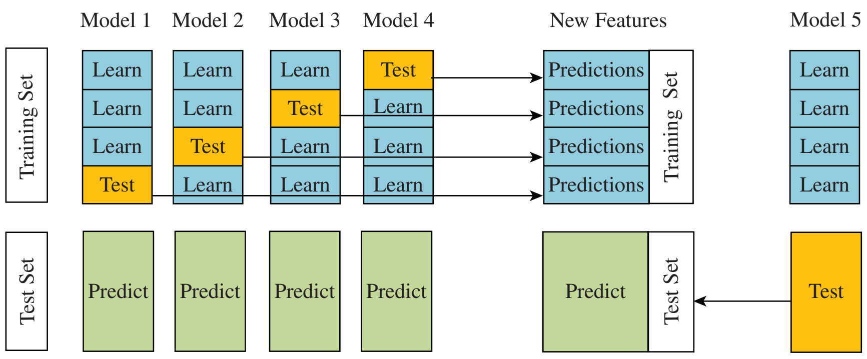


Figure 2.15. Illustration of stacking.

Table 2.12. Summary of three model ensemble methods.

- *Bagging*, often built on top of homogeneous weak learners by randomly selecting the subset for the training set with replacement and combining them by a deterministic averaging/voting process.
- *Boosting*, often built on top of homogeneous weak learners, which is constructed in a sequential and adaptive way (a base model depends on the previous ones) and combining them by a deterministic averaging/voting process.
- *Stacking*, often built on top of heterogeneous weak learners, which is constructed in parallel and combines them by training a meta-model to output a prediction based on the different weak learners.

(2) Use stratified k-fold cross-validation on the training set. Each sample in the training set appears only once in the validation set of the k-fold cross-validation. We add the predicted output as a new feature.
(3) Repeat above step for n models.
(4) Learn a new meta-model using the new features (and optionally the original input) as the input and output on the training set.
(5) Make a prediction using the meta-model on the testing data.

In conclusion, Table 2.12 gives a summary of the three main types of model ensemble method we have discussed.

2.4 Exercises

(1) What is the supervised learning problem?
(2) Is the forecasting of the future price of some stock a regression problem?
(3) What is the commonly used loss function in the regression problem?
(4) What is the cross entropy?
(5) What is a categorical variable?
(6) What is the difference between the regression problem and the classification problem?