Chapter 2

Supervised Learning

In the previous chapter, we looked at the definition of machine learning, some of its real-world applications, and various types of learning problems. In this chapter, we focus on a supervised learning problem. It is considered the most studied area in the machine learning community.

2.1 Setting

Recall that a learning problem consists primarily of a data set, a model class, and a learner. In supervised learning problems, the data set consists of an independent and identically distributed (i.i.d.) sample pair

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n) \tag{2.1}$$

where each pair $(\mathbf{x}_i, y_i)$ is drawn from some unknown distribution $P(X, Y)$ over $\mathcal{X} \times \mathcal{Y}$. We refer to each $(\mathbf{x}_i, y_i)$ as a training example, or an example for brevity. Each $\mathbf{x}_i$ is called a *data point* or a *feature vector* associated with the ith example. The y_i is the target value of the ith example we are interested in learning. The key assumption of supervised learning is that we know the target values of all data points. We will refer to $\mathcal{X}$ as an *input space* and $\mathcal{Y}$ as an *output* space. The input space $\mathcal{X}$ is the space in which the data point $\mathbf{x}_i$ take values. The target y_i take values in the output space $\mathcal{Y}$.

The collection of examples in (2.1) forms a *training set* of the learning problem. Mathematically speaking, the training set consists of an $n \times d$ matrix $\mathbf{X}$ and an $n\times$ vector $\mathbf{y}$ given by

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

The ith row of $\mathbf{X}$ corresponds to the d dimensional feature vector $\mathbf{x}_i$, while the ith element of the vector $\mathbf{y}$ corresponds to the true target value associated with the $i$the example.

How do we obtain or construct the feature vector $\mathbf{x}_i$ and the target value y_i for each training example? The answer will depend on the nature of learning problems and the data we are dealing with. In principle, the feature vector $\mathbf{x}_i$ should be constructed such that it contains useful information about the training data that will allow us to infer its target value y_i. In most cases, it is a design choice which information should be used. The target value y_i is usually provided by human experts.

To understand this step, we consider a couple of examples.

Example 1 : 7-segment digits recognition. Suppose we want to build a system that is capable of recognizing the 7-segment digits. Examples of 7-segment digits are depicted in Figure 2.1. To build the system, we first need to decide how to represent each digit in terms of a feature vector. In this example, there are 7 input sources (that is, LEDs) and there are 2 possible values for each of them (that is, on and off). Each digit can be displayed by turning on only respective LEDs.

Hence, one possibility is to represent each digit by a binary vector of length seven. That is, $\mathbf{x}_i \in \{0,1\}^7$. The value of x_{ij} indicates whether or not the jth LED of the ith example is on. To construct the target value y_i, we look at each digit $\mathbf{x}_i$ individually and specify which number it represents. Since there are 10 possible digits, we have $y_i \in \{0,1,2,\ldots,9\}$. We can write this representation in terms of the matrix $\mathbf{X}$ and the vector $\mathbf{y}$ as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \\ \mathbf{x}_6 \\ \mathbf{x}_7 \\ \mathbf{x}_8 \\ \mathbf{x}_9 \\ \mathbf{x}_{10} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}.$$

Figure 2.1: 7-segment digits

Above representation tells us that the digit $\mathbf{x}_5 = [0, 1, 1, 1, 0, 1, 0]$, for example, corresponds to the number 4. It is easy to see that this representation captures all information about the problem. It completely distinguish different digits from each other.

To recognize new digits, we can simply perform a *pattern matching*. That is, we can follow these two steps.

1. For each new digit $\mathbf{x}_t$, find the digit $\mathbf{x}_i$ in the data set $\mathbf{X}$ that matches $\mathbf{x}_t$ exactly.

2. Output the corresponding y_i.

Unfortunately, building the system based on pattern matching strategy can be time-consuming and is only applicable to some simple problems. It may not be the best solution in most of the real-world problems, as demonstrated in the next example.

Example 2 : MNIST handwritten digits recognition. Suppose we want to build a system that is capable of recognizing handwritten digits instead of the 7-segments digits. Examples of MNIST handwritten digits are depicted in Figure 2.2. Unlike the previous example, there are variations in the writing styles. Hence, the digits of the same number may appear differently. For example, digit 3 can be written in different ways.

How do we best represent each data point in this problem? The simplest solution is to try something similar to the previous example. That is, we can view each digit as an 28×28 black-and-white image. By flattening each image, we obtain a 784 dimensional binary vector. The target vector $\mathbf{y}$ can be obtained in a similar manner as in the previous example. We can write this representation in terms of the matrix $\mathbf{X}$ and the vector $\mathbf{y}$ as follows:

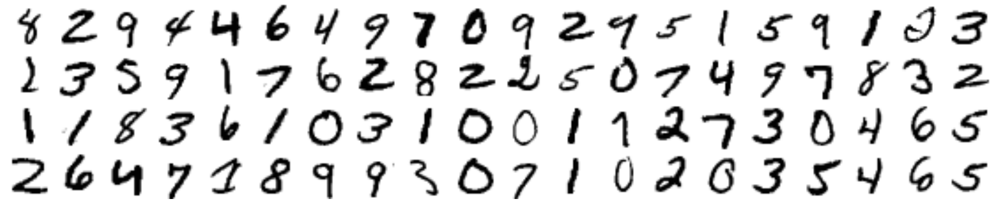


Figure 2.2: MNIST handwritten digits

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \vdots \\ \mathbf{x}_{76} \\ \mathbf{x}_{77} \\ \mathbf{x}_{78} \\ \mathbf{x}_{79} \\ \mathbf{x}_{80} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 0 & 0 & 1 & \cdots & 0 & 1 & 0 \\ 1 & 0 & 1 & \cdots & 1 & 0 & 1 \\ 1 & 0 & 1 & \cdots & 0 & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 0 & \cdots & 0 & 1 & 1 \\ 1 & 1 & 0 & \cdots & 1 & 1 & 1 \\ 1 & 0 & 1 & \cdots & 0 & 1 & 0 \\ 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 1 & 1 & 1 & \cdots & 0 & 1 & 1 \end{bmatrix}}_{\text{784 dimensions}}, \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 4 \\ 2 \\ 1 \\ \vdots \\ 4 \\ 9 \\ 3 \\ 6 \\ 5 \end{bmatrix}.$$

What if we are interested in using this data set for a prediction? Given new images of the digits, it may be tempted to perform pattern matching. However, unlike the last example, there exist several feature vectors $\mathbf{x}_i$ that correspond to the same number due to the writing variation. Moreover, it is likely that we will not find the examples in the data set that match our test data exactly. You may think collecting more data or changing data representation could potentially solve the problems. This is only partially true. It makes the problem simpler, but it does not solve the aforementioned problem. This is when the *machine learning* techniques come into rescue.

Having an abundant amount of data and a good data representation are a good starting point in any learning problems. In this chapter, we will look at several supervised learning techniques that can be used to tackle this problem.

Definition 1: supervised learning

Given a training set $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ of size n where an example $(\mathbf{x}_i, y_i)$ is an independent and identically distributed (i.i.d.) sample from some unknown distribution $P(X, Y)$ over a joint space $\mathcal{X} \times \mathcal{Y}$. Let $\ell : \mathcal{Y} \times \mathcal{Y} \to$

$[0, \infty)$ be a measurable loss function and $\mathcal{M} = \{f \mid f : \mathcal{X} \to \mathcal{Y}\}$ be a pre-specified class of functions from $\mathcal{X}$ to $\mathcal{Y}$. Supervised learning aims to find $f \in \mathcal{M}$ for which the expected risk $R : \mathcal{M} \to [0, \infty)$ given by

$$R(f) = \int \ell(f(\mathbf{x}), y) \, \mathrm{d}P(\mathbf{x}, y) \tag{2.2}$$

is minimized.

Since we do not have access to the underlying distribution $P(X, Y)$, it is not possible to optimize (2.2) directly. An *empirical risk minimization* (ERM) is the most commonly used framework for supervised learning problems. Rather than minimizing the expected risk, it aims to minimize the *empirical risk* defined as

$$\widehat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(\mathbf{x}_i), y_i) \tag{2.3}$$

evaluated with respect to the training set $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. Since we have access to the training set D, we can evaluate and optimize the empirical risk (2.3) directly. By the law of large numbers, $\widehat{R}(f) \to R(f)$ for any $f \in \mathcal{M}$ as n goes to infinity. Thus, in theory it is the right quantity to minimize. However, it is known that this may not be the right thing to do in practice as it could lead to *overfitting* which we will discuss in the following section.

Consider the example of handwritten digit recognition problem. In this case, the function f takes an image of handwritten digit $\mathbf{x}_i$ and outputs the corresponding number $f(\mathbf{x}_i)$. The loss function $\ell(f(\mathbf{x}_i), y_i)$ measures the loss incurred by predicted target $f(\mathbf{x}_i)$ while the true target is y_i. For example, a zero-one loss between $f(\mathbf{x}_i)$ and y_i will be one if $f(\mathbf{x}_i) \neq y_i$, and zero otherwise. The empirical risk $\widehat{R}(f)$ is essentially an average loss of f over the training data. For zero-one loss, $\widehat{R}(f)$ corresponds to the proportion of mistakes made by the function f on the training examples.

We will look into more details about the theory of the ERM framework in Chapter 3.

2.1.1 Loss Functions

A *loss function* $\ell(f(\mathbf{x}), y)$, given in Definition 1, is an important component in supervised learning problems. The purpose of the loss function $\ell(f(\mathbf{x}), y)$ is to measure the pointwise discrepancy between the true target value y and the predicted value $y' = f(\mathbf{x})$. There exist several loss functions in the literature.

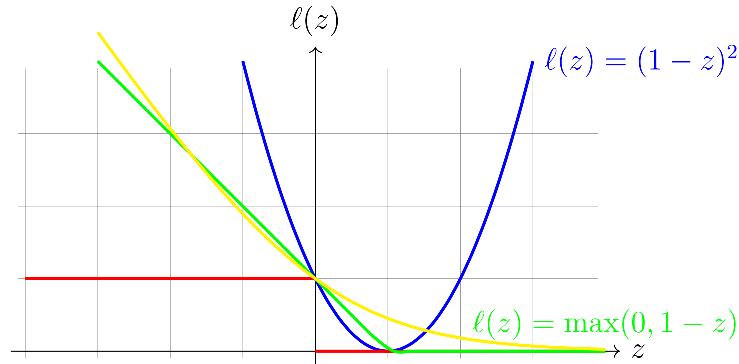


Figure 2.3: Plots of the commonly used loss functions.

For example, in a binary classification problem, one often considers a zero-one loss function

$$\ell(y, y') = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{otherwise} \end{cases}.$$

Hence, the empirical risk (2.3) with the zero-one loss corresponds to the number of mistakes, normalized by the sample size, made on the training set by the classifier f. Unfortunately, optimizing the zero-one loss function is known to be NP-hard. In general, the ERM will picks the solution f from $\mathcal{M}$ which minimizes an average loss over the training set. We provide examples of commonly used loss functions in Table 2.1.

Table 2.1: Commonly used loss functions

Loss	**Function**	**Problem**
Zero-one loss	$\ell(y, y') = \mathbf{1}_{y \neq y'}$	
Square loss	$\ell(y, y') = (y - y')^2$	
Hinge loss	$\ell(y, y') = \max(0, 1 - y \cdot y')$	
Absolute loss	$\ell(y, y') = \lvert y - y' \rvert$	
ϵ-insensitive loss	$\ell(y, y') = (\lvert y - y' \rvert - \epsilon)_+$	

The plots of these loss functions are illustrated in Figure 2.3.

2.2 Linear Algorithms

Now that we have introduced all the ingredients of the supervised learning, let us look at the simplest supervised learning problem : a *linear regression.*

Suppose we are interested in predicting a person's height given the weight of that person. We will do this using the machine learning approach.

First, we collect weight and height data from 30 people. The data are shown in Figure 2.4. It is clear that there is a linear positive correlation between weight and height of a person. Therefore, we consider the class of linear functions

$$\mathcal{M} = \{f(x) = w \cdot x + b \mid w, x \in \mathbb{R}\}$$

where w and b are parameters whose values are to be determined using the data. Since this is a regression problem, we will use the square loss function $\ell(f(x), y) = (f(x) - y)^2 = (w \cdot x + b - y)^2$. Hence, the empirical risk of this problem can be written as

$$\begin{aligned} \widehat{R}(f) &= \frac{1}{2}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \\ &= \frac{1}{2}\sum_{i=1}^{n}(w \cdot x_i + b - y_i)^2. \end{aligned} \tag{2.4}$$

To derive the solution of this problem, it will be more convenient to rewrite everything in terms of vectors and matrices. Having defined a *weight vector* $\mathbf{w} = [w, b]^\top$ and an augmented data $\mathbf{x}_i = [x_i, 1]^\top$, it is easy to see that $f(\mathbf{x}_i) = \mathbf{w}^\top\mathbf{x}_i = w \cdot x_i + b$. Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^\top$ be the data matrix and $\mathbf{y} = [y_1, y_2, \ldots, y_n]^\top$ be a vector of target values. Then, we can rewrite (2.4) as

$$\begin{aligned} \widehat{R}(\mathbf{w}) &= \frac{1}{2}\sum_{i=1}^{n}(\mathbf{w}^\top\mathbf{x}_i - y_i)^2 \\ &= \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}). \end{aligned} \tag{2.5}$$

The equation (2.5) is known as a sum of squared error (SSE). To find the function f that minimizes $\widehat{R}(f)$, we take the derivative of $\widehat{R}(f)$ with respect to the vector $\mathbf{w}$

$$\begin{aligned} \frac{d}{d\mathbf{w}}\widehat{R}(\mathbf{w}) &= \frac{d}{d\mathbf{w}}\left[\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})\right] \\ &= \frac{d}{d\mathbf{w}}\left[\frac{1}{2}(\mathbf{y}^\top\mathbf{y} - 2\mathbf{y}^\top\mathbf{X}\mathbf{w} + \mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w})\right] \\ &= \mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{y}^\top\mathbf{X}. \end{aligned}$$

Setting the derivative to zero yields the *normal equation*

$$\mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$$

$$\mathbf{w} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}. \tag{2.6}$$

That is, solving the problem of linear regression comes down to solving a system of linear equations. Moreover, the solution has an analytic form, *i.e.*, $\mathbf{w} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$. This is commonly known as a least square algorithm. While this seems appealing, computing the inverse $(\mathbf{X}^\top\mathbf{X})^{-1}$ can be numerically unstable. To solve (2.6) efficiently, one can resort to the *Cholesky decomposition* of $\mathbf{X}^\top\mathbf{X}$, *i.e.*, $\mathbf{X}^\top\mathbf{X} = \mathbf{R}^\top\mathbf{R}$ where $\mathbf{R}$ is an upper triangular matrix. Then, we can solve for $\mathbf{w}$ using the forward-backward substitution of

$$(\mathbf{X}^\top\mathbf{X})\mathbf{w} = (\mathbf{R}^\top\mathbf{R})\mathbf{w} = \mathbf{X}^\top\mathbf{y}.$$

Since $\mathbf{R}$ is upper triangular, forward-backward substitution can be solved efficiently.

An alternative approach is to turn the least square problem into an optimization problem and solve it using the *gradient descent* algorithm. That is, we minimize the least square objective

$$\mathbf{w} = \arg\min_{\mathbf{w}} \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$$

by repeating the following update equation until convergence

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + \alpha\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$$

where α is a step size parameter. An advantage of the gradient descent algorithm is that it does not require the calculation of the inverse $(\mathbf{X}^\top\mathbf{X})^{-1}$ which makes it relatively more numerically stable. Nevertheless, it relies on the step size parameter α which must be chosen carefully to ensure the convergence of the algorithm.

2.2.1 Generalization

When $\mathbf{X}^\top\mathbf{X}$ is not invertible, solving the least square is an *ill-posed* problem, that is, a problem in which the solution may not exist, may have more than one solution, or in which the solutions depend discontinuously upon the initial data. To see when this can happen, consider these three situations:

- if $n > d$, $\mathbf{X}^\top\mathbf{X}$ is of full rank and thus is invertible.
- if $n < d$, the rank of $\mathbf{X}^\top\mathbf{X}$ is smaller than d. Hence, it is not invertible.

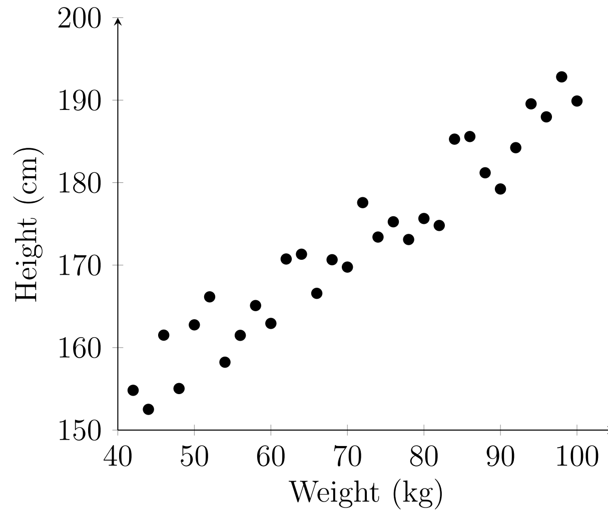


Figure 2.4: The data consisting of weights (kg) and heights (cm) of 30 people. By inspection, there seems to be a linear correlation between weight and height of a person.

- if the columns of $\mathbf{X}$ are colinear, the features are not linearly independent. Depending on the values of n and d, this may cause $\mathbf{X}^\top\mathbf{X}$ to be a singular matrix.

As we can see, the problem can become ill-posed either when the sample size n is too small compared to the dimension d or when the features are not linearly independent, or both. This is bad for machine learning as it can lead to *overfitting*. In the machine learning community, overfitting refers to a situation in which the learned model performs well on the training data, but fails on the previously unseen test data. As opposed to overfitting, the learned model is said to *generalize* well if it performs well on both the training data and the unseen test data. Metaphorically speaking, successful students are those who not only succeed at learning in class, but also excel in an exam.

To see why an ill-posedness could lead to a catastrophic effect in any learning problems. Consider the following situation. Let D_1 be a data set which consists of true data $\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n$ from some distribution $P(Z)$ and $\mathcal{M}$ be a fixed class of functions. Then, we have that $h_1 = F_{\boldsymbol{\theta}}(D_1, \mathcal{M})$ is the solution of a learning problem $F_{\boldsymbol{\theta}}$. Now, we construct another data set D by adding a small noise ε to D_1, *i.e.*, $D_2 = \{\mathbf{z}_1+\varepsilon, \mathbf{z}_2+\varepsilon, \dots, \mathbf{z}_n+\varepsilon\}$. By applying the same learning problem to this data set, we obtain $h_2 = F_{\boldsymbol{\theta}}(D_2, \mathcal{M})$. If $F_{\boldsymbol{\theta}}$ is ill-posed, its solutions will depend *discontinuously* upon the data sets. This means that h_2 may differ arbitrarily from h_1, even though D_1 and D_2

are slightly different. This is a serious problem because, in practice, we never observe a true data, but only a noisy version of it.

The aforementioned discussion implies that the least square algorithm is sensitive to noise in the data set. See Figure 2.5 for an illustration. In other words, the value of $\mathbf{w}$ will become large if the magnitude of the data is large, *i.e.*, overfitting. Hence, a straightforward solution to this problem is to penalize large value of $\mathbf{w}$. In general, penalizing $\mathbf{w}$ is an instance of complexity control strategies known as a *regularization*, which usually leads to better generalization.

To impose regularization strategies, we resort to the *regularized empirical risk minimization* (RERM) framework. In this framework, rather than minimizing $\widehat{R}(f)$ alone, we consider the objective function

$$\widehat{R}_\lambda(f) = \widehat{R}(f) + \lambda\Omega(f) \tag{2.7}$$

where $\Omega : \mathcal{M} \to [0,\infty)$ is called a regularization functional and $\lambda \in [0,\infty)$ is a regularization parameter. The majority of works in machine learning also consider $\Omega(f) = \Omega(\|f\|_{\mathcal{M}})$ where Ω is assumed to be a monotonically increasing function of $\|f\|_{\mathcal{M}}$. In principle, $\Omega(f)$ should be large if the function f is complicated, whereas it should be small for a simple function. The regularization parameter λ controls the importance of the regularization term compared to the empirical risk term.

Ridge regression. The well-known linear regression algorithm which uses the regularization strategy is a *ridge regression* algorithm. To penalize the value of $\mathbf{w}$ in (2.4), the ridge regression algorithm considers the minimization of the following risk functional

$$\begin{aligned} \widehat{R}_\lambda(\mathbf{w}) &= \frac{1}{2}\sum_{i=1}^{n}(f(\mathbf{x}_i) - y_i)^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\ &= \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) + \frac{1}{2}\mathbf{w}^\top\mathbf{w}. \end{aligned} \tag{2.8}$$

Since $\|\mathbf{w}\|_2^2 = \mathbf{w}^\top\mathbf{w} = \sum_{i=1}^d w_i^2$, $\widehat{R}_\lambda(\mathbf{w})$ penalizes large values of $\mathbf{w}$. Moreover, if $\lambda \to 0$, we obtain the least square solution, whereas if $\lambda \to \infty$, the vector $\mathbf{w}$ will approach a zero vector.

The solution of ridge regression can be obtained in a similar way as that of the least square. Taking the derivative of $\widehat{R}_\lambda(\mathbf{w})$ with respect to $\mathbf{w}$ and setting it to zero yield the normal equation

$$(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = \mathbf{X}^\top\mathbf{y}$$

$$\mathbf{w} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}. \quad (2.9)$$

where $\mathbf{I}_d$ denotes the $d \times d$ identity matrix. The reason the regularization in (2.9) leads to better solution than the one in (2.6) is left as an exercise to the reader.

To illustrate the effect of regularization, consider the regression problem depicted in Figure 2.5. The data are generated as follows:

$$y = 1.2x + 2.2 + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, 0.64)$$

First, we generated a fixed amount of training data using the above equation. Next, we intentionally added some noisy examples into the data set. The resulting training data are shown as black dots in Figure 2.5. Then, we applied both the least square and the ridge regression with $\lambda = 10$ on the training set. Finally, we generated a new set of data on which we evaluated both solutions in terms of the *mean square error* (MSE) $\mathcal{E}(f) = (1/n)\sum_{i=1}^n (y_i - f(x_i))^2$. The table below reports the MSE of both algorithms.

Algorithm	**Training Error**	**Test Error**
Least Square	7.7101	2.9767
Ridge Regression	7.9088	0.7389

It is clear that the ridge regression solution is the better solution, although the training error leads us to believe that this is not the case.

2.2.2 Bayesian Perspective

There is a probabilistic interpretation of both the least square and ridge regression. We can understand both the least square and the ridge regression from the Bayesian perspective. That is, the least square solution is a *maximum likelihood* estimate of the probabilistic linear model

$$Y = \mathbf{w}^\top X + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2)$$

where $\mathbf{w} \in \mathbb{R}^d$ is the parameter vector we want to learn. It follows that Y is distributed according to the Gaussian distribution $\mathcal{N}(\mathbf{w}^\top X, \sigma^2)$.

Given a data set $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ consisting of an i.i.d. sample from $P(X, Y)$, we can write down a likelihood function as

$$\begin{aligned} P(D \,|\, \mathbf{w}) &= P((\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \,|, \mathbf{w}) \\ &= \prod_{i=1}^n P((\mathbf{x}_i, y_i) \,|\, \mathbf{w}). \end{aligned} \quad (2.10)$$

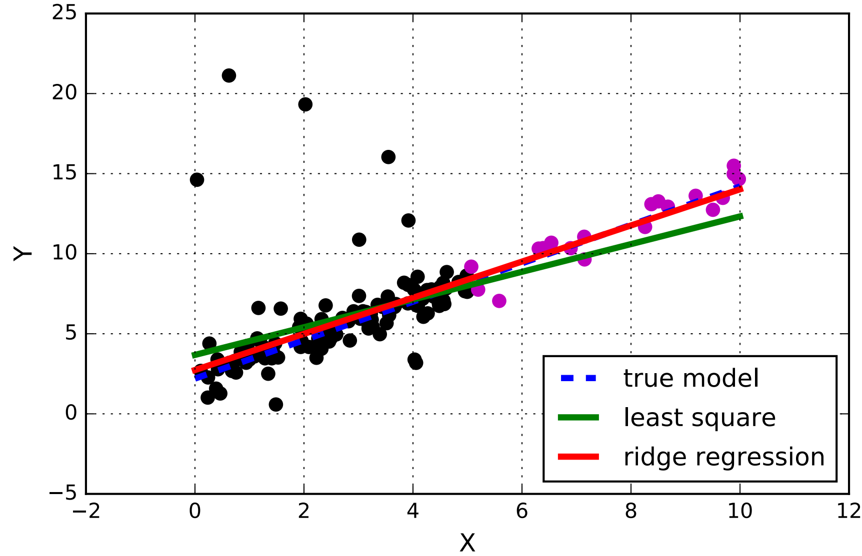


Figure 2.5: A comparison between the least square and the ridge regression. The black dots represent training data used to fit the models, while the magenta dots represent the test data used to evaluated the models.

We use the i.i.d. assumption in (2.10) to factorize the joint likelihood. Our goal is to choose the parameter $\mathbf{w}$ that maximizes the probability of observed data. In most cases, it will be more convenient to work with log-likelihood. Taking log function of the above equation yields

$$\begin{aligned}\log P(D \,|\, \mathbf{w}) &= \sum_{i=1}^{n} \log P((\mathbf{x}_i, y_i) \,|\, \mathbf{w}) \\ &= -\sum_{i=1}^{n} \frac{(y_1 - \mathbf{w}^\top \mathbf{x}_i)^2}{2\sigma^2} + C \end{aligned} \tag{2.11}$$

where C is a constant term that does not depend on $\mathbf{w}$. The maximum likelihood estimate, denoted by $\mathbf{w}_{\mathrm{ML}}$, can be obtained by maximizing (2.11) with respect to $\mathbf{w}$, *i.e.*,

$$\begin{aligned}\mathbf{w}_{\mathrm{ML}} &= \arg\max_{\mathbf{w}} \; \log P(D \,|\, \mathbf{w}) \\ &= \arg\max_{\mathbf{w}} \left\{ -\sum_{i=1}^{n} \frac{(y_1 - \mathbf{w}^\top \mathbf{x}_i)^2}{2\sigma^2} + C \right\} \\ &= (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \end{aligned} \tag{2.12}$$

That is, the maximum likelihood estimate $\mathbf{w}_{\mathrm{ML}}$ coincides with the least square solution.

Likewise, the solution of the ridge regression algorithm corresponds to a *maximum a posteriori* (MAP) estimate. In MAP estimation, our goal is to choose the parameter $\mathbf{w}$ that is most probable given observed data and prior belief. That is, we assume a priori that

$$\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma_w^2 \mathbf{I}).$$

Figure 2.6 illustrates two different prior distributions over $\mathbf{w}$. Under this prior, the log of posterior over $\mathbf{w}$ given the data D can be expressed using Bayes' rule as

$$\begin{aligned}
\log P(\mathbf{w} \,|\, D) &= \log P(D \,|\, \mathbf{w}) P(\mathbf{w}) \\
&= \sum_{i=1}^{n} \log P((\mathbf{x}_i, y_i) \,|\, \mathbf{w}) + \log P(\mathbf{w}) \\
&= \sum_{i=1}^{n} \log \left(\frac{e^{-(y_i - \mathbf{w}^\top \mathbf{x}_i)^2 / 2\sigma^2}}{\sqrt{2\pi}\sigma} \right) + \log \left(\frac{e^{-\|\mathbf{w}\|_2^2 / 2\sigma_w^2}}{\sqrt{2\pi}\sigma_w} \right) \\
&= -\sum_{i=1}^{n} \frac{(y_i - \mathbf{w}^\top \mathbf{x}_i)^2}{2\sigma^2} - \frac{\|\mathbf{w}\|_2^2}{2\sigma_w^2} + C \qquad (2.13)
\end{aligned}$$

where C denotes constant terms that does not depend on $\mathbf{w}$. Maximizing (2.13) yields the MAP estimate

$$\begin{aligned}
\mathbf{w}_{\mathrm{MAP}} &= \arg\max_{\mathbf{w}} \left\{ -\sum_{i=1}^{n} \frac{(y_i - \mathbf{w}^\top \mathbf{x}_i)^2}{2\sigma^2} - \frac{\|\mathbf{w}\|_2^2}{2\sigma_w^2} + C \right\} \\
&= \left(\mathbf{X}^\top \mathbf{X} + \frac{\sigma^2}{\sigma_w^2} \mathbf{I} \right)^{-1} \mathbf{X}^\top \mathbf{y}. \qquad (2.14)
\end{aligned}$$

Therefore, setting $\lambda = \sigma^2 / \sigma_w^2$ leads to the same solution as the ridge regression problem.

Remark 2.2.1. *The regularization parameter* $\lambda = \sigma^2/\sigma_w^2$ *in* (2.14) *is inversely proportional to the variance* σ_w^2 *of the prior. Intuitively, when* λ *is relatively large, we penalize the solutions* $\mathbf{w}$ *with large* $\|\mathbf{w}\|_2$ *and hence prefer* $\mathbf{w}$ *with smaller values of* $\|\mathbf{w}\|_2$*. Probabilistically speaking, this corresponds to choosing a prior distribution* $\mathcal{N}(\mathbf{0}, \sigma_w^2)$ *with a small value of* σ_w^2*,* i.e.*, the probability mass concentrates around the zero vector.*

Remark 2.2.2. *The probabilistic interpretation presented in this section highlights a key distinction between **generative** and **discriminative** models in machine learning.*

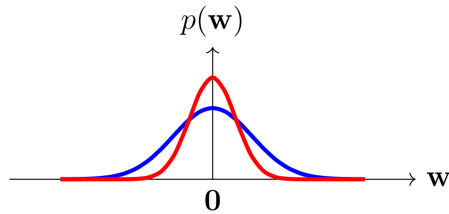
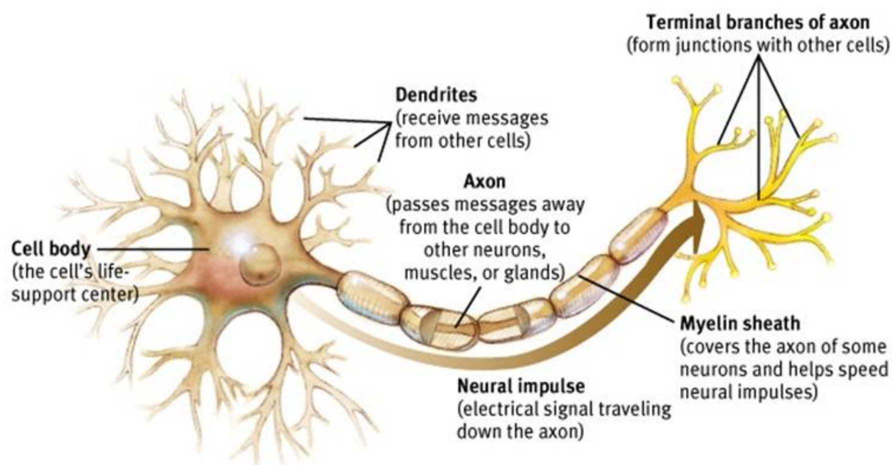


Figure 2.6: Gaussian distributions with different values of variance σ_w^2 as priors over parameter vector $\mathbf{w}$.



Credit: *http://www.appsychology.com*

Figure 2.7: A neuron.

2.2.3 Perceptrons

A perceptron algorithm is one of the oldest learning algorithms for a classification problem (Rosenblatt 1958).

Consider a simple classification problem. Given a data point $\mathbf{x}$, our goal is to decide which category the point $\mathbf{x}$ belongs. For example, $\mathbf{x}$ may be an image of handwritten digit and the task is to identify which number this digit represents. Another example is a medical diagnosis where $\mathbf{x}$ corresponds to medical information about a patient. The task is then to identify whether or not the patient has a certain disease, *e.g.*, cancer. When $\mathcal{Y} = \{-1, +1\}$, we have a binary classification problem. If $\mathcal{Y}$ consists of more than two classes, it is called a multi-class classification problem. In the following, we focus only on the binary classification problem, but it is straightforward to extend the binary classifier to multi-class classifier.

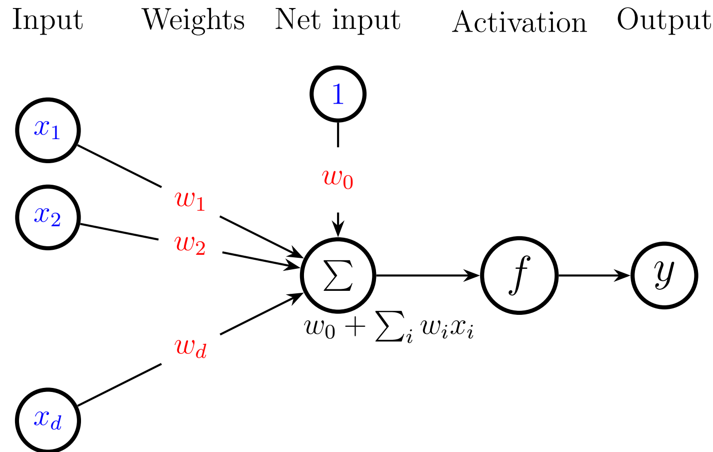


Figure 2.8: A schematic diagram of the perceptron.

Suppose we are given a data set $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ of size n as a training data where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y}$ are the true labels. Our goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$ that generalizes well to unseen data. Like in the case of regression problem, we will consider a linear function class $\mathcal{M} = \{f_{\mathbf{w}}\}$ parametrized by the parameter vector $\mathbf{w}$ in $\mathbb{R}^d$. For binary classification, $f_{\mathbf{w}}$ is called a *linear classifier* and is defined by

$$f_{\mathbf{w}}(\mathbf{x}) = \operatorname{sign}\left(w_0 + \sum_{i=1}^{d} w_i \cdot x_i\right) = \begin{cases} +1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + w_0 \geq 0 \\ -1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + w_0 < 0 \end{cases}. \tag{2.15}$$

Hence, the decision boundary, or the *separating hyperplane*, can be described by the following equation

$$w_0 + \langle \mathbf{w}, \mathbf{x} \rangle = w_0 + \sum_{i=1}^{d} w_i x_i = 0.$$

Let $\mathbf{w}' = (w_0, w_1, w_2, \ldots, w_d)^\top \in \mathbb{R}^{d+1}$ and $\mathbf{x}' = (1, x_1, x_2, \ldots, x_d)^\top \in \mathbb{R}^{d+1}$. Then, we can simplify the notation in (2.15) further as $\langle \mathbf{w}', \mathbf{x}' \rangle = w_0 + \langle \mathbf{w}, \mathbf{x} \rangle = w_0 + \sum_{i=1}^{d} w_i x_i$. Whenever it simplifies the presentation, we will simply write $\langle \mathbf{w}, \mathbf{x} \rangle$ as as a dot product in $\mathbb{R}^{d+1}$.

In contrast to the regression problem, what matters to us in this problem is the sign of $\langle \mathbf{w}, \mathbf{x} \rangle$, *i.e.*, which side of the hyperplane the point $\mathbf{x}$ lies, rather than its magnitude. We will refer to $\mathbf{w}$ as a weight vector. Figure 2.9 illustrates a linear separating hyperplane having $\mathbf{w}$ as a normal vector. The goal of learning algorithms like the perceptron algorithm is to find an optimal weight vector $\mathbf{w}^*$ given the training data. The weight vector $\mathbf{w}^*$ is said to

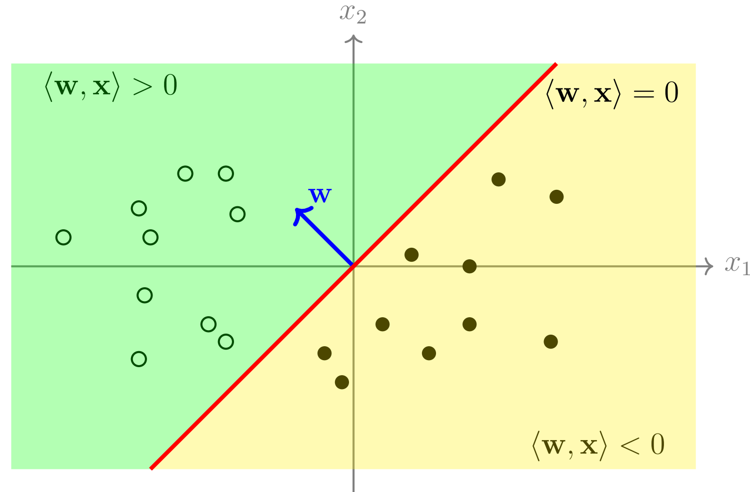


Figure 2.9: A geometric view a separating hyperplane defined by a normal vector $\mathbf{w}$ in $\mathbb{R}^2$.

be optimal if the classifier $f_{\mathbf{w}^*}(\mathbf{x}) = \langle \mathbf{w}^*, \mathbf{x} \rangle$ achieves the best generalization accuracy.[1]

Perceptron learning rule. The perceptron algorithm finds the optimal $\mathbf{w}^*$ in an online fashion. That is, starting with an initial weight vector $\mathbf{w}_0$, the algorithm processes one example at a time, and then update the weight vector according to the following update equation

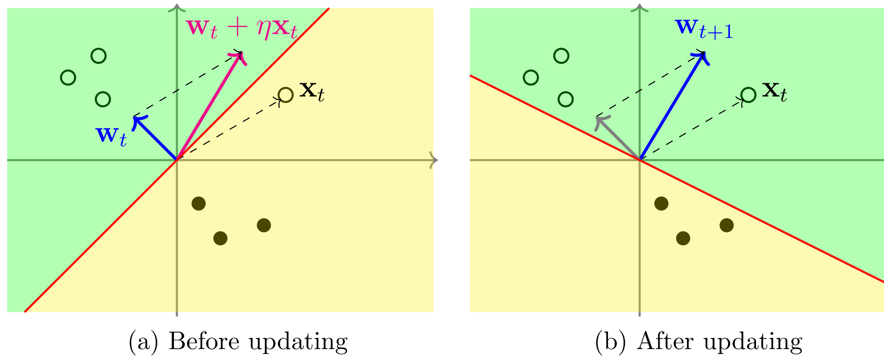
$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \mathbb{1}[f(\mathbf{x}_t) \neq y_t]\eta \cdot y_t \cdot \mathbf{x}_t \tag{2.16}$$

where $\eta > 0$ is a parameter representing the learning rate of the algorithm and $\mathbb{1}[\cdot]$ is an indicator function. In practice, we often initialize $\mathbf{w}_0$ by a zero vector and keep applying the update equation (2.16) until convergence or after reaching some number of iterations T. In words, it can be understood as follows:

- if the current example $\mathbf{x}_t$ is misclassified by the current weight vector $\mathbf{w}_t$, the new weight vector $\mathbf{w}_{t+1}$ should do better at classifying $\mathbf{x}_t$ the next time it sees this example.
- if the current example $\mathbf{x}_t$ is classified correctly, we leave the weight vector $\mathbf{w}_t$ as it is and move on to the next example.

[1] In practice, it is difficult to evaluate the true generalization accuracy as we do not have access to the test data during the training time. However, we can use the training error as a proxy and then impose some forms of the regularization to prevent overfitting.

Put differently, we adjust the weight vector $\mathbf{w}_t$ toward the direction of the input vector $\mathbf{x}_t$ if it is misclassified, and do nothing otherwise. To understand why this is the right thing to do, see Figure 2.10. Note that we may require several passes over the training set before reaching the optimal solution. Algorithm 1 summarizes the perceptron algorithm.



(a) Before updating (b) After updating

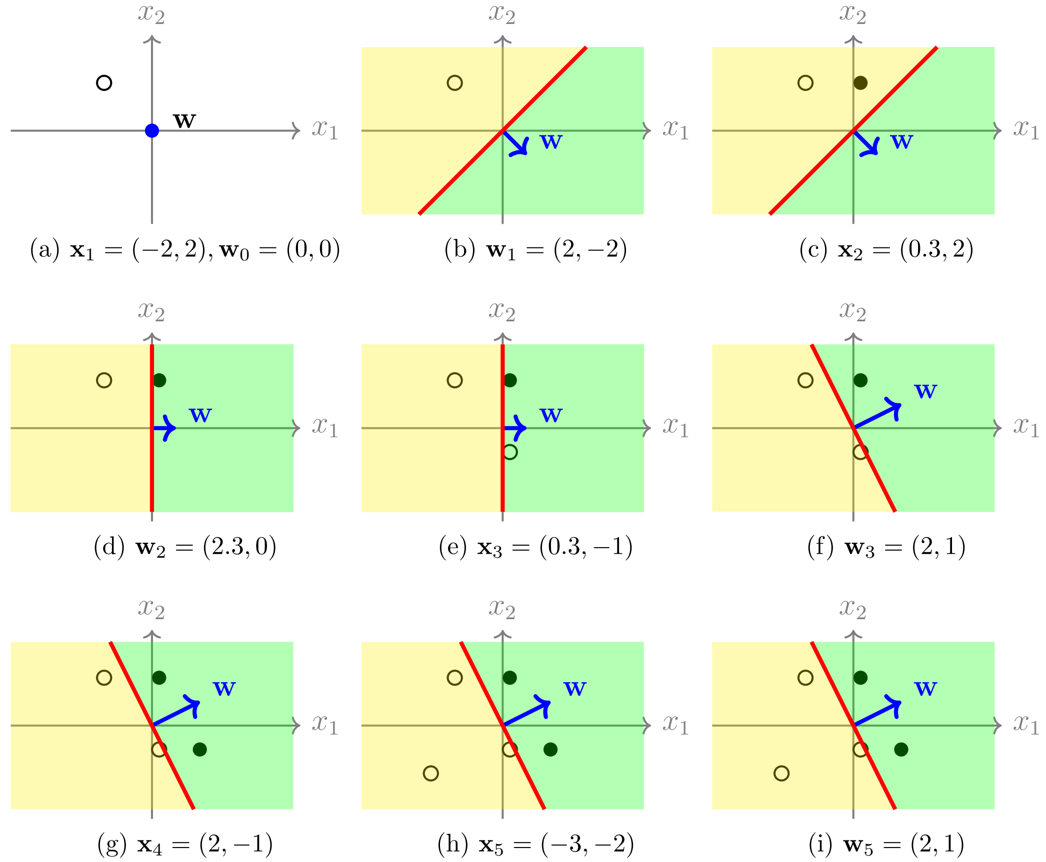
Figure 2.10: An illustration of the update equation (2.16). At first, the positive example $\mathbf{x}_t$ is misclassified by the classifier parametrized by $\mathbf{w}_t$ (left). Hence, we apply $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta\mathbf{x}_t$ which moves $\mathbf{w}_t$ closer to the vector $\mathbf{x}_t$. After the update, the classifier parametrized by $\mathbf{w}_{t+1}$ classifies $\mathbf{x}_t$ correctly while still maintaining the same accuracy on the remaining examples (right).

The behavior of the perceptron algorithm also depends on the learning rate parameter η. Intuitively, it controls the agressiveness of each update iteration. If η is small, the weight vector $\mathbf{w}_t$ will change gradually, whereas if η is large, the weight vector $\mathbf{w}_t$ will change abruptly at each iteration. The right value of η ensures that the perceptron learning rule will converge quickly. However, finding the right value of η involves fine-tuning and is sometimes more of an art than a science.

Remark 2.2.3. *Given training examples $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$, it is easy to see that the weight vector $\mathbf{w}_T$ obtained after applying the update equation (2.16) can be expressed as*

$$\begin{aligned}\mathbf{w}_T &= \eta \cdot y_1 \cdot \mathbf{x}_1 + \eta \cdot y_2 \cdot \mathbf{x}_2 + \cdots + \eta \cdot y_T \cdot \mathbf{x}_T \\ &= \sum_{i=1}^{n} \eta m_i y_i \mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i,\end{aligned}$$

where m_i is the number of times $\mathbf{x}_i$ is misclassified and $\alpha_i = \eta m_i y_i$. That is, the final solution of the perceptron learning rule can be written as a linear



(a) $\mathbf{x}_1 = (-2, 2), \mathbf{w}_0 = (0, 0)$ (b) $\mathbf{w}_1 = (2, -2)$ (c) $\mathbf{x}_2 = (0.3, 2)$

(d) $\mathbf{w}_2 = (2.3, 0)$ (e) $\mathbf{x}_3 = (0.3, -1)$ (f) $\mathbf{w}_3 = (2, 1)$

(g) $\mathbf{x}_4 = (2, -1)$ (h) $\mathbf{x}_5 = (-3, -2)$ (i) $\mathbf{w}_5 = (2, 1)$

Figure 2.11: A step-by-step illustration of the perceptron learning rule.

combination of the training points where the coefficient α_i *depends on the learning rate* η*, the number of times* $\mathbf{x}_i$ *is misclassified, and its true label* y_i*. As a result, the prediction function* $f(\mathbf{x})$ *can be written as*

$$f(\mathbf{x}) = \operatorname{sign}(\langle \mathbf{w}_T, \mathbf{x} \rangle) = \operatorname{sign}\left(\left\langle \sum_{i=1}^{n} \alpha_i \mathbf{x}_i, \mathbf{x} \right\rangle \right) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i \langle \mathbf{x}_i, \mathbf{x} \rangle \right).$$

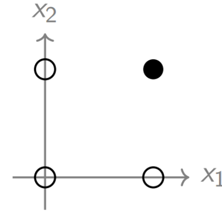
Note that we can now express the prediction function $f(\mathbf{x})$ *entirely in terms of the coefficients* $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ *the the dot product between* $\mathbf{x}$ *and the training points.*

Example 2.2.4 (AND function)**.** *Suppose we want to learn an AND function* $f(x_1, x_2) = x_1 \wedge x_2$ *using the perceptron learning rule. The training examples are given in the table below.*

Algorithm 1 Perceptron learning rule

```
1: procedure PERCEPTRON(w_0, η, D)
2:     w_1 ← w_0
3:     for t ← 1,...,T do
4:         ŷ_t ← sign(⟨w_t, x_t⟩)          ▷ classify x_t using the current w_t
5:         if ŷ_t ≠ y_t then
6:             w_{t+1} ← w_t + η · y_t · x_t   ▷ update w_t if x_t is misclassified
7:         else
8:             w_{t+1} ← w_t               ▷ do nothing if x_t is classified correctly
9:     return w_{T+1}
```

ID	x_1	x_2	$x_1 \wedge x_2$
1	*False*	*False*	*False*
2	*False*	*True*	*False*
3	*True*	*False*	*False*
4	*True*	*True*	*True*



To ease the learning process, we will represent the truth value `True` *and* `False` *by the numeric values* 1 *and* 0*, respectively. Table 2.2 shows a step-by-step procedure of the perceptron learning rule for learning the AND function. The learning rate* η *is chosen to be* 0.4.

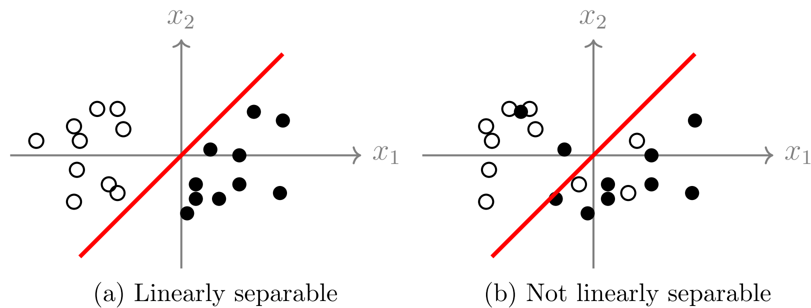
Example 2.2.4 possesses an essential property which ensures that $\mathbf{w}_t$ obtained using the perceptron learning rule will converge to the optimal solution as t increases. We call this property a *linear separability* of the data set. Informally, the data set is said to be linearly separable if there exists a linear hyperplane that can classify all the examples in the data set correctly. We give a formal definition of linear separability in Definition 2 and provide an illustration in Figure 2.12.

Definition 2: Linear separability

Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T) \in \mathbb{R}^d \times \{-1, +1\}$ be a sequence of T examples. These examples are said to be linearly separable with margin ρ if there

Table 2.2: A step-by-step perceptron learning for the AND problem. We denote the true label by y and the predicted label by $\hat{y}$. The learning rate η is chosen to be 0.4.

iteration	x_0	x_1	x_2	y	w_0	w_1	w_2	$\hat{y}$	Updated?
1	1	0	0	-1	0.0	0.0	0.0	1	yes
2	1	0	1	-1	-0.4	0.0	0.0	-1	no
3	1	1	0	-1	-0.4	0.0	0.0	-1	no
4	1	1	1	1	-0.4	0.0	0.0	-1	yes
5	1	0	0	-1	0.0	0.4	0.4	1	yes
6	1	0	1	-1	-0.4	0.4	0.4	1	yes
7	1	1	0	-1	-0.8	0.4	0.0	-1	no
8	1	1	1	1	-0.8	0.4	0.0	-1	yes
9	1	0	0	-1	-0.4	0.8	0.4	-1	no
10	1	0	1	-1	-0.4	0.8	0.4	1	yes
11	1	1	0	-1	-0.8	0.8	0.0	1	yes
12	1	1	1	1	-1.2	0.4	0.0	-1	yes
13	1	0	0	-1	-0.8	0.8	0.4	-1	no
14	1	0	1	-1	-0.8	0.8	0.4	-1	no
15	1	1	0	-1	-0.8	0.8	0.4	-1	no
16	1	1	1	1	-0.8	0.8	0.4	1	no



(a) Linearly separable (b) Not linearly separable

Figure 2.12: A linearly separable data set (left). A data set that is not linearly separable (right). Almost all data sets obtained from the real world applications are not linearly separable.

exist $\rho > 0$ and $\mathbf{w} \in \mathbb{R}^d$ such that

$$\rho < \frac{y_t\langle \mathbf{w}, \mathbf{x}_t\rangle}{\|\mathbf{w}\|}$$

for all $t = 1, \ldots, T$.

Given that the data set is linearly separable, the following theorem gives a mistake bound, which is an upper bound on the number of mistakes, made by the Perceptron algorithm when processing a sequence of training examples.

Theorem 2.2.5

Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T) \in \mathbb{R}^d \times \{-1, +1\}$ be a sequence of T examples which are linearly separable with margin ρ. Assume that there exists some $r > 0$ such that $\|\mathbf{x}_t\| \leq r$ for all $t = 1, \ldots, T$. Then, the number of mistakes made by the Perceptron algorithm when processing $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T)$ is bounded from above by r^2/ρ^2.

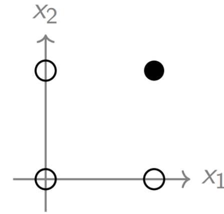
Proof. Let M be the total number of updates out of T rounds. It follows from the linear separability assumption that

$$
\begin{aligned}
M\rho \leq \frac{\mathbf{x} \cdot \sum_{t=1}^{M} y_t \mathbf{x}_t}{\|\mathbf{x}\|} &\leq \left\| \sum_{t=1}^{M} y_t \mathbf{x}_t \right\| && \text{(Cauchy-Schwarz inequality)} \\
&= \left\| \sum_{t=1}^{M} (\mathbf{w}_{t+1} - \mathbf{w}_t) \right\| && \text{(definition of updates)} \\
&= \|\mathbf{w}_{T+1}\| && \text{(telescoping sum)} \\
&= \sqrt{\sum_{t=1}^{M} \|\mathbf{w}_{t+1}\|^2 - \|\mathbf{w}_t\|^2} && \text{(telescoping sum)} \\
&= \sqrt{\sum_{t=1}^{M} \|\mathbf{w}_t + y_t \mathbf{x}_t\|^2 - \|\mathbf{w}_t\|^2} && \text{(definition of updates)} \\
&= \sqrt{\sum_{t=1}^{M} 2y_t \langle \mathbf{w}_t, \mathbf{x}_t \rangle + \|\mathbf{x}_t\|^2} && (2y_t \langle \mathbf{w}_t, \mathbf{x}_t \rangle \leq 0) \\
&\leq \sqrt{\sum_{t=1}^{M} \|\mathbf{x}_t\|^2} \leq \sqrt{Mr^2}.
\end{aligned}
$$

Rearranging terms gives $M \leq r^2/\rho^2$, which concludes the proof. □

Example 2.2.6 (XOR function)**.** *Suppose we want to learn an XOR function $f(x_1, x_2) = x_1 \oplus x_2$ using the perceptron learning rule. The training examples are given in the table below.*

ID	x_1	x_2	$x_1 \oplus x_2$
1	*False*	*False*	*False*
2	*False*	*True*	*True*
3	*True*	*False*	*True*
4	*True*	*True*	*False*



This is a well-known example of a data set which is not linearly separable. If we apply the perceptron learning rule to this problem, the algorithm will never converge.

When the training set is not linearly separable like in Example 2.3.1, the common approach in employing the perceptron algorithm is to stop updating $\mathbf{w}_t$ after reaching a pre-specified number of iterations T.

The models we consider so far are linear models. Unfortunately, a family of linear models is too restrictive for most practical applications. In what follows, we look at several ways to build non-linear extensions of such models.

2.3 Neural Networks and Deep Learning

The most straightforward way to extend the perceptron is to build an interconnected network of perceptrons in reminiscence of the brain as an interconnected network of neurons. In machine learning, this kind of models is commonly known as an artificial neural network (ANN) and multi-layer perceptron (MLP).

2.3.1 Multi-Layer Perceptrons

TBD

- Activation functions
  - Heaviside
  - Sigmoid
  - Hyperbolic Tanh
- Basis functions

Example 2.3.1 (XOR problem)**.** *Suppose we want to learn an XOR function* $f(x_1, x_2) = x_1 \oplus x_2$ *using the perceptron learning rule. The training examples are given in the table below.*