Contents

1. The Learning Problem
1.1 What Is Learning?
1.2 Types of Learning
1.3 Perceptron Learning for Classification
1.4 Adaptive Linear Neuron Algorithm for Perceptron Learning
1.5 The Learning Problem
1.6 Hoeffding Inequality
1.7 The Vapnik-Chervonenkis (VC) Dimension
1.8 Bias-Variance Decomposition
1.9 Expected Errors versus Number of Data Points
MATLAB Code
Problems
2. Linear Models
2.1 Linear Models for Classification
2.2 Linear Models for Regression
2.3 Logistic Regression
2.4 Nonlinear Transformations
2.5 Dimension Reduction
MATLAB Code
Problems
3. Overfitting, Regularization and Validation
3.1 What Is Overfitting?
3.2 Regularization Techniques
3.3 Validation
MATLAB Code
Problems
4. Support Vector Machine and Kernel Methods
4.1 Support Vector Machines
4.2 Kernel Trick and Soft-Margin Support Vector Machines
4.3 Radial Basis Functions for Classification and Regression
MATLAB Code
Appendices
A. Probability Theory
B. Optimization Methods and Software
C. A proof of the Hoeffding Inequality
Problems

References

[1] V. N. Vapnik, *Statistical Learning Theory*, Wiley, 1998.

[2] V. N. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed., Springer, 2000.

[3] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed., Springer, 2009.

[4] H. Stark and J. W. Woods, *Probability and Random Processes with Applications to Signal Processing*, 3rd ed., Prentice Hall, 2002.

[5] A. Antoniou and W.-S. Lu, *Practical Optimization–Algorithms and Engineering Applications*, Springer 2007.

[6] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004.

[7] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[8] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data – A Short Course*, AMLbook.com, 2012.

[9] F. Rosenblatt, "The perception: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386-408, 1958.

[10] G.-X. Yuan, C.-H. Ho, and C.-J. Lin, "Recent advances of large-scale linear classification," *Proc. IEEE*, vol. 100, no. 9, pp. 2584-2603, Sept. 2012.

[11] T. Poggio and S. Smale, "The mathematics of learning: Dealing with data," Notices of the AMS, vol. 50, no. 5, pp. 537-544, May 2003.

[12] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd edition, The Johns Hopkins University Press, 1989.

[13] A. Ng, *Lecture Notes, ML Class*, Stanford University, 2011.

[14] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," In *Machine Learning: Proceedings of the 21th International Conference (ICML '04)*, pp. 919-926, 2004.

[15] Y. Zhang and L. Xiao, "Stochastic prime-dual coordinate method for regularized empirical risk minimization," arXiv:1409.3257, Sept. 2014.

[16] W. B. Johnson and Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Conference in Modern Analysis and Probability (New Haven, Conn., 1982)*, Contemporary Mathematics **26**, Providence, RI: American Mathematical Society, pp. 189–206, 1984.

[17] I. T. Jolliffe, *Principal Component Analysis*, 2nd edition, Springer, 2002.

[18] Y. S. Abu-Mostafa's web link: http://www.work.caltech.edu/lectures.html , Lectures 14 – 16.

Chapter 1 The Learning Problem

1.1 What Is Learning?

Throughout we shall be concerned with signal processing aspects of machine learning. State it in other words, it is of our primary interest to study methods and algorithms that facilitate our *learning from data*. Most of the time in this course, the term *data* is taken to mean *measurements* that may come from real world, for example financial data from a stock exchange and precipitation readings from weather broadcasting, or from synthetic environments such as Monte-Carlo simulations that statistically evaluate certain computational algorithms.

Suppose we observe a data set, $\mathcal{D}$, which has an explicit and causal input-output structure. We want to utilize the data to learn the "*governing law*" (some authors in the literature call it *underlining process*, *target function, target distribution* [8], *target operator*, or *supervisor's operator* [1]) that is responsible for how the data were produced. Such a desire to learn is quite natural because understanding this governing law would greatly facilitate reliable prediction of what will occur next or, stated in a more formally, facilitate us to infer something *outside* $\mathcal{D}$. As a matter of fact, to infer what will occur outside $\mathcal{D}$ is the central goal of every (supervised) learning process.

At a glance, learning the governing law from a given set of data appears to be a hard mission, if not impossible.

Example 1.1

Figure 1.1 explains this point of view, where a data set may come from completely different "governing laws" – in this case the governing laws are two polynomial functions of order 2 and 5.

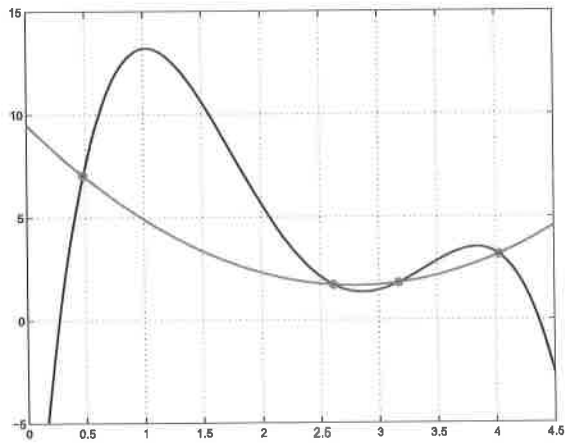


Figure 1.1 The data set $\{(x_i, y_i)$, for $i = 1, 2, 3, 4\}$, marked by small circles in red, may come from widely different governing laws which are represented by the blue and black curves respectively, casting doubts as whether or not one can infer the true governing law behind the data observed.

■

So is learning from data a hopeless business? As we will see shortly, the answer depends on how we look at the problem. Below we present an example of *learning by examples* and examine the

feasibility of learning from a probabilistic perspective.

Example 1.2

Suppose we have a box containing many balls which appear identical except the colors: they are either red or blue. The ratio of the number of red balls to the total number of balls is μ, in other words, the probability of red ball is μ. Note that μ is *not* random, in fact it is a constant. Obviously, μ is an important character of the box, thus we decide to conduct an experiment to learn μ by example. We try to learn μ by picking N balls at random from the box (each time a ball is picked, its color is recorded and the ball is then returned to the box) and computing the ratio ν of the number of red balls in the N samples to N. The value of ν is random because when the experiment is repeated a different value of ν might be obtained. On the other hand, it is quite clear that the true probability μ effects ν, and in this regard μ may be regarded as the governing law in this example, we try to learn the value of μ by examples (i.e. samples). To address the question of whether the learning is feasible, we notice the following:

- The probability of having a ν that is widely different from μ is small. For example, if μ = 0.9, and $N = 10$, the probability of having a $\nu \le 0.1$ is 9.1×10^{-9}, meaning that the event of (μ = 0.9) and the event of ($\nu \le 0.1$) are highly unlikely to occur simultaneously.
- When the sample number N is large, then with high probability ν well approximates μ. In fact, for any sample size N, the probability of "ν differs from μ by more than ε" is bounded from above as

$$P[|\nu-\mu|>\varepsilon]\le 2e^{-2\varepsilon^2 N} \tag{1.1}$$

for any $\varepsilon > 0$. The inequality in (1.1) is known as the *Hoeffding inequality* [8], see the appendix for a proof of (1.1). From (1.1) we see that for a given $\varepsilon > 0$, the chances of "ν deviating from μ by more than ε" are vanishing exponentially quickly as more (random) samples are utilized in computing ν.

In conclusion, *statistically* one can learn the (unknown) probability μ via ν as long as the sample size N is large enough, and the ability of statistical leaning in this particular case is quantified by (1.1). ■

1.2 Types of Learning

The basic assumption in our machine learning framework is that we have collected a set of observations which we called *data*, and the purpose of learning is to uncover the governing law, in one way or another, that is responsible for the production of the data observed. In the literature, this kind of learning strategy is called "*learning from examples*", and the observations used in the learning process are called *training data*.

Depending on the structure of the training data, machine learning methods may be categorized as *supervised* and *unsupervised* leaning.

A. *Supervised Learning*

A training data in supervised learning assumes the form $\{(\boldsymbol{x}_n, y_n), n = 1,2,...,N\}$ where $\boldsymbol{x}_n$ are examples from an input space $\mathcal{X}$ and $y_n \in \mathcal{Y}$ are produced by a (unknown) causal system (i.e., a

governing law) with $\boldsymbol{x}_n$ as input. For example, in a machine learning system that deals with credit card applications, $\boldsymbol{x}_n$ may represent a historic record of a customer and $y_n \in \mathcal{Y} = \{-1, 1\}$ labels the bank's approval or disapproval of the customer's application.

The most popular problems in supervised learning are *classification* and *regression*. In a classification problem, the goal is to assign each input (that is outside the training data) to one of the *finite number* of *discrete* categories. In a regression problem, the learning is aimed at finding a continuous-valued approximating function to fit a set of training data.

B. *Unsupervised Learning*

In unsupervised learning, we are just given a set of samples $\{\boldsymbol{x}_n, n = 1, 2, \ldots, N\}$, but the labels y_n are not available. Because y_n are missing, the available observations are called unlabeled data. The learning problems in this case are typically related to pattern classifications and structure identification (such as number of data clusters, centers of clusters, and clusters themselves) of the input data.

1.3 Perceptron Learning for Classification

A. *Perceptron Model and Perceptron Learning Algorithm (PLA)*

Let $\mathcal{X} = R^m$ and $\mathcal{Y} = \{-1, 1\}$ be the input and output spaces, respectively. Thus each input $\boldsymbol{x} \in \mathcal{X}$ is an m-dimensional vector whose components $\{x_i, i = 1, \ldots, m\}$ are called *features*. In a credit-card application problem, for example, these input components may include years in residence, salary, etc., and the output is a binary label 1 or –1, representing "approval" or "disapproval" of the application, respectively.

Perceptron, originally developed by F. Rosenblatt in 1958 [9], is based on a simple functional model

$$y = h(\boldsymbol{x}) = \text{sign}\left(b + \sum_{i=1}^{m} w_i x_i\right) \tag{1.2}$$

where sign(z) is the so-called *sign function* which equals 1 if $z > 0$ and –1 if $z < 0$; and b and w_i for $i = 1, \ldots, m$ are model parameters that can be trained by training data so as for (1.2) to make good decisions *outside* the training data. Note that, given parameters b and w_i's, the output y of the perceptron depends only on the *sign* of $b + \sum_{i=1}^{m} w_i x_i$, hence the equation

$$b + \sum_{i=1}^{m} w_i x_i = 0 \tag{1.3}$$

plays an important role in the decision making process. Geometrically, (1.3) represents an (m – 1)-dimensional hyperplane in the input space R^m, which is often called a *decision boundary*, that divides space R^m into two parts with the points $\boldsymbol{x}$ satisfying $b + \sum_{i=1}^{m} w_i x_i > 0$ (hence the associated output $y =$ 1) on side of the boundary and the points $\boldsymbol{x}$ satisfying $b + \sum_{i=1}^{m} w_i x_i < 0$ (hence an output $y = -$ 1) on the other side of the boundary. From this perspective, the classification problem at hand amounts to utilizing the information provided by the training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$ to tune parameters b and w_i's so that the decision boundary in (1.3)

separates data pairs $(\boldsymbol{x}_n, y_n)$ with $y_n = 1$ from those $(\boldsymbol{x}_n, y_n)$ with $y_n = -1$.

Before describing the perceptron learning algorithm (PLA), let us simplify the notation by defining $\boldsymbol{w} = [b \quad w_1 \quad \cdots \quad w_m]^T$ (in some literature the first component b is renamed as w_0) and $\boldsymbol{x} = [1 \quad x_1 \quad \cdots \quad x_m]^T$ so that (1.2) becomes

$$y = h(\boldsymbol{x}) = \text{sign}\left(\boldsymbol{w}^T \boldsymbol{x}\right) \tag{1.4}$$

Given training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$ where each $\boldsymbol{x}_k$ is now understood as an $(m+1)$-dimensional vector, the PLA updates the weights in an iterative manner. We start by setting the weight vector $\boldsymbol{w}$ in (1.4) to an initial guess $\boldsymbol{w}(0)$ and an iteration counter t to 0. Then for each pair $(\boldsymbol{x}_n, y_n)$ from the training data, we verify to see if the value of y in (1.4) at $\boldsymbol{x}_k$ coincides with y_k. With $\boldsymbol{w} = \boldsymbol{w}(0)$, (1.4) is said to classify $\boldsymbol{x}_n$ correctly if $\text{sign}\left(\boldsymbol{w}(0)^T \boldsymbol{x}_n\right) = y_n$, otherwise (1.4) is said to misclassify $\boldsymbol{x}_n$. If no data pairs in $\mathcal{D}$ are misclassified, then $\boldsymbol{w}(0)$ is claimed to be a solution that is expected to work well outside the training data. Otherwise, weight $\boldsymbol{w}$ is updated as follows.

Suppose the current weight $\boldsymbol{w}(t)$ misclassifies several data pairs from $\mathcal{D}$, we pick one of these pairs *at random*, denote it by $(\boldsymbol{x}(t), y(t))$, and update $\boldsymbol{w}(t)$ to $\boldsymbol{w}(t+1)$ using

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + y(t)\boldsymbol{x}(t) \tag{1.5}$$

With $\boldsymbol{w} = \boldsymbol{w}(1)$ and $t = 1$, the above procedure repeats itself and again, there are two possible outcomes: no misclassifications are found for the training data so that $\boldsymbol{w}(1)$ is a solution weight vector, or misclassifications occur and (1.5) is applied to update from $\boldsymbol{w}(1)$ to $\boldsymbol{w}(2)$, and so on. The iteration continues until the current $\boldsymbol{w}$ classifies all data pairs from $\mathcal{D}$ correctly.

B. *Remarks*

- To understand how (1.5) works, suppose $\boldsymbol{w}(t)$ misclassifies $(\boldsymbol{x}(t), y(t))$ in the training data. In other words, we have $\text{sign}(\boldsymbol{x}(t)^T \boldsymbol{w}(t)) = -y(t)$. Multiplying (1.5) by $\boldsymbol{x}(t)^T$ yields

$$\boldsymbol{x}(t)^T \boldsymbol{w}(t+1) = \underbrace{\boldsymbol{x}(t)^T \boldsymbol{w}(t)}_{\text{its sign is } -y(t)} + y(t) \cdot \underbrace{\boldsymbol{x}(t)^T \boldsymbol{x}(t)}_{\substack{\text{this part is always} \\ \text{positive, so the sign} \\ \text{of this term is } y(t)}} \tag{1.6}$$

We see that the last term in the equation always tries to fix the misclassification regardless of the value $y(t)$.

- The PLA iterations based on (1.5) are guaranteed to converge as long as the training data $\mathcal{D}$ is separable by a linear decision boundary. Here convergence is meant to identify a weight vector $\boldsymbol{w}^*$ which yields a decision boundary $\boldsymbol{x}^T \boldsymbol{w}^* = 0$ that separates the points (in $\mathcal{D}$) labeled by 1 from those points (in $\mathcal{D}$) labeled by -1. It can be shown [8] that the number of iterations for the PLA to converge is bounded from above by

$$t \le \frac{R^2 \| \boldsymbol{w}^* \|^2}{\rho^2} \tag{1.7a}$$

where

$$\rho = \min_{1 \le n \le N} y_n (\boldsymbol{x}_n^T \boldsymbol{w}^*) \text{ and } R = \max_{1 \le n \le N} \| \boldsymbol{x}_n \| \tag{1.7b}$$

Example 1.3

A set of training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,...,15\}$ with $x_n \in R^2$ and $y_n \in \{1,-1\}$ are given below, where each column of three numbers is arranged as $[\boldsymbol{x}_n; y_n]$. The data are supposedly associated with a credit application record with the components of $\boldsymbol{x}_n$ representing years in residence and salary (in 10K dollars), respectively, of the nth applicant, and y_n representing the bank's approval (+1) or otherwise (–1).

Columns 1 through 10

1.0000	1.5000	2.1000	3.1000	3.5000	4.0000	5.0000	5.9000	6.9000	7.9000
4.5000	6.5000	3.5000	4.9000	6.5000	4.2000	5.0000	2.7000	3.7000	2.2000
-1.0000	1.0000	-1.0000	-1.0000	1.0000	-1.0000	1.0000	-1.0000	1.0000	-1.0000

Columns 11 through 15

8.4000	9.0000	10.0000	10.5000	11.2000
2.6000	1.6000	5.0000	0.8000	1.5000
1.0000	-1.0000	1.0000	-1.0000	1.0000

Fig. 1.2a depicts the training data where the points in blue are those in the training data with $y_n = +1$ while the points in red are those with $y_n = -1$. Applying PLA to the training data with initial weight set to $\boldsymbol{w}(0) = [0\ 0\ 0\ ]^T$, it took the PLA 183 iterations to converge to the solution weights as

$$\boldsymbol{w}^* = \begin{bmatrix} -27 & 2 & 4.2 \end{bmatrix}^T$$

which yields a linear decision boundary

$$-27 + 2x_1 + 4.2x_2 = 0$$

This decision boundary represented by a straight line in black shown in Fig. 1.2b. We note the decision boundary successfully separates the two classes of training data.

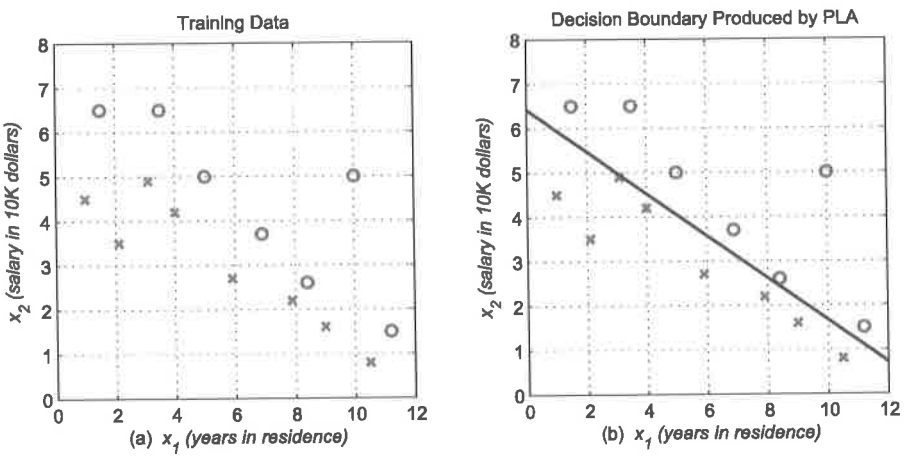


Figure 1.2

The MATLAB code implementing PLA for Example 1.3 is ex1_3_PLA, see the code at the end of this chapter.

1.4 Adaptive Linear Neuron Algorithm for Perceptron Learning

A variant of PLA, known as the adaptive linear neuron (*Adaline*) algorithm for perceptron learning [8], modifies the updating law in (1.5) by taking into account the actual difference between the value $\boldsymbol{w}(t)^T \boldsymbol{x}(t)$ and label $y(t)$. To be specific, in each iteration one picks a data pair from the training data at random (hence it may or may not be misclassified by the current weight) and call it $(\boldsymbol{x}(t), y(t))$, then defines

$$\rho(t) = \boldsymbol{w}(t)^T \boldsymbol{x}(t)$$

and updates the weight vector as follows

$$\boldsymbol{w}(t+1) = \begin{cases} \boldsymbol{w}(t) + \eta \cdot (y(t) - \rho(t)) \cdot \boldsymbol{x}(t) & \text{if } y(t) \cdot \rho(t) \le 1 \\ \boldsymbol{w}(t) & \text{otherwise} \end{cases} \tag{1.8}$$

where $\eta > 0$ is a scaling constant that needs to be set carefully for convergence.

Example 1.4

Apply *Adaline* algorithm to the problem in Example 1.3. With $\boldsymbol{w}(0) = [0\ 0\ 0]^T$ and $\eta = 0.03$, it took *Adaline* 727 iterations to yield a weight vector

$$\boldsymbol{w}^* = \begin{bmatrix} -45.5944 & 3.5885 & 6.7679 \end{bmatrix}^T$$

hence a linear decision boundary

$$-45.5944 + 3.5885 + 6.7679 x_2 = 0$$

This decision boundary is represented by a straight line in black in Fig. 1.3b. Note that the decision boundary also successfully separates the two classes of training data.

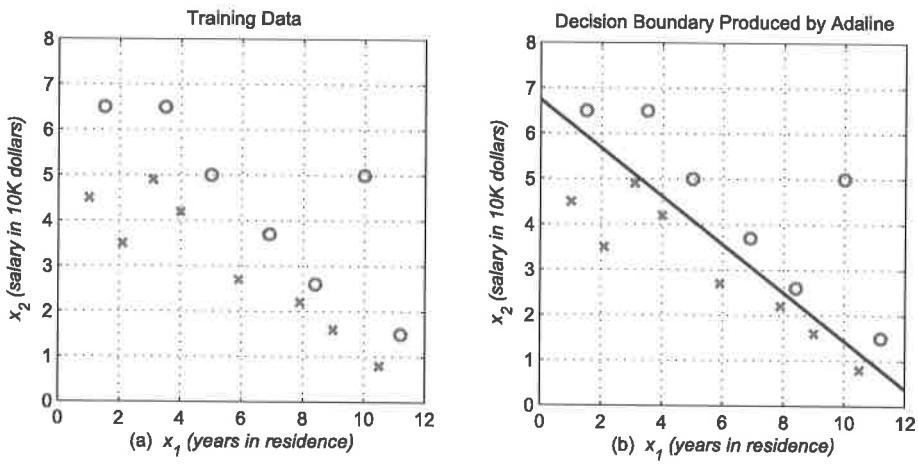


Figure 1.3

The MATLAB code that implements *Adaline* for Example 1.4 is ex1_3_adaline, see the code at the end of this chapter.

1.5 The Learning Problem

The supervised learning problem is illustrated in Fig. 1.4. The figure, which is from reference [8], depicts the basic structure of the problem as well as the data and information flow within and

between various functional blocks. As such, the figure is useful for us to understand the inner working mechanism of supervised machine learning systems. In what follows we offer further explanations and comments on each of these functional blocks.

• In supervised learning, we are given a set of training data $\{(\boldsymbol{x}_n, y_n), n = 1,2,...,N\}$ and a set of hypotheses $\mathcal{H}$. The training data are produced by an unknown governing law which we want to learn, in Fig. 1.4 this governing law is seen as *unknown target distribution* $P(y \mid \boldsymbol{x})$ (or unknown target function $y = f(\boldsymbol{x})$). In compliance with the terminology in the literature, from now on we shall use *target function* (or *target distribution*) to mean the governing law we want to uncover.

In practical applications, we encounter noisy targets when the output is not uniquely determined by the input. A general model for a noisy target is the conditional distribution $P(y \mid \boldsymbol{x})$ which, by definition, is the probability that for a given input $\boldsymbol{x}$ the output assumes value y. A concrete way to think of the conditional distribution model for a noisy target is to express it as sum of a deterministic target function and a noise distribution, namely

$$y = f(\boldsymbol{x}) + \varepsilon \tag{1.9}$$

where function $f(\boldsymbol{x})$ is deterministic and ε is a random noise. For example, for the noise target in (1.9) with a Gaussian white noise ε with mean zero and variance σ^2, the target distribution is given by

$$P(y \mid \boldsymbol{x}) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{y} e^{-(z-f(\boldsymbol{x}))^2/2\sigma^2} dz$$

It is worth pointing out that a deterministic target function may be considered as a special case of the more general model based conditional distributions. As a matter of fact, the conditional probability distribution given by

$$P(y \mid \boldsymbol{x}) = \begin{cases} 1 & \text{if } y = f(\boldsymbol{x}) \\ 0 & \text{if } y \neq f(\boldsymbol{x}) \end{cases}$$

models exactly a deterministic function $f(\boldsymbol{x})$.

• The hypothesis set $\mathcal{H}$ contains a (finite or infinite) set of candidate target functions (distributions) which is supposed to be rich enough to include the target function itself or candidates that are good approximations of the target function. On the other hand, it is very important to avoid using an unnecessarily large $\mathcal{H}$. Intuitively, a large hypothesis set makes it difficult for the learning algorithm to search and find an appropriate final hypothesis. It is true that the in-sample error reduces as the hypothesis set gets larger, however, having a small in-sample error is merely a step in a machining learning process, but not its goal -- a candidate hypothesis cannot be considered appropriate unless it yields a small out-of-sample error. We shall revisit this issue later with analysis tools such as VC bound and bias-variance decomposition.

• Based on the training data $\mathcal{D}$ and hypothesis set $\mathcal{H}$, it is the "learning algorithm" block that identifies a candidate hypothesis, say $g(\boldsymbol{x})$, such that $g(\boldsymbol{x}) \approx f(\boldsymbol{x})$. The candidate hypothesis $g(\boldsymbol{x})$ is shown in the "final hypothesis" block as an outcome of the learning algorithm.

• From the points made above, we realize that the target distribution and training examples are dictated by the problem, while the learning algorithm and hypothesis set are solution tools we have to choose. Together the hypothesis set and learning algorithm are referred to as the *learning*

model [8].

• The "unknown input distribution" block is connected to "training examples" because the x_n's in the training data are generated in a way that obeys a certain probabilistic distribution $P(\boldsymbol{x})$. For example, the x_n's may be obtained by sampling a certain domain in input space $\mathcal{X}$ uniformly randomly.

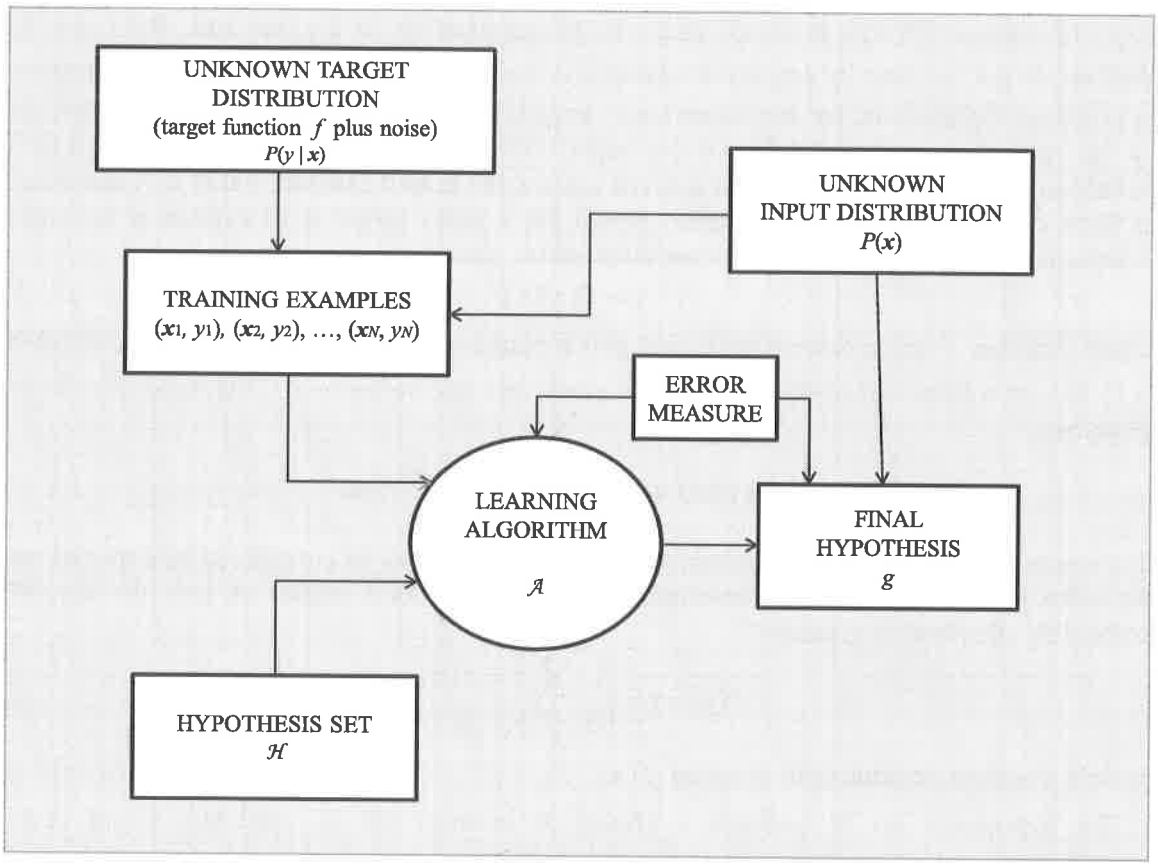


Figure 1.4 The supervised learning problem [8].

• The "unknown input distribution" block is also connected to the "final hypothesis" block because in order to evaluate the closeness of $g(\boldsymbol{x})$ to $f(\boldsymbol{x})$, the data $\boldsymbol{x}$ used must obey the *same distribution* $P(\boldsymbol{x})$ as in the training data.

• The "error measure" block is seen to be connected to "learning algorithm" as well as the "final hypothesis" block. This is because a learning algorithm inevitably involves computational error analysis, where having an adequately defined error measure is critical. Likewise, the justification of $g(\boldsymbol{x}) \approx f(\boldsymbol{x})$ clearly requires an error measure. We remark that in the literature error measure is often called *loss function*.

Let $h \in \mathcal{H}$ be a hypothesis that approximates target function f. We use $E(h, f)$ to denote an error measure to quantify the closeness between a hypothesis h and target function f. For the in-sample error, an error measure is substantiated through errors at individual input points $\boldsymbol{x}_n$, namely

$e(h(\boldsymbol{x}_n), f(\boldsymbol{x}_n))$, and the average of this pointwise error is taken as an error measure, namely

$$E_{\text{in}}(h, f) = \frac{1}{N}\sum_{n=1}^{N} e(h(\boldsymbol{x}_n), f(\boldsymbol{x}_n)) = \frac{1}{N}\sum_{n=1}^{N} e(h(\boldsymbol{x}_n), y_n) \tag{1.10}$$

The performance of hypothesis h is not justified by whether or not its in-sample error is small but by whether or not its error outside the training data is small. A formal term to call the process of applying a hypothesis $h(\boldsymbol{x})$ to outside training data in the input space is *generalization*, and generalization performance is evaluated by an error measure, called *out-of-sample* error and denoted by $E_{\text{out}}(h, f)$, that computes average of the *same* type of pointwise error $e(h(\boldsymbol{x}), f(\boldsymbol{x}))$ over the entire input space. Statistically the "average" becomes expectation, and the out-of sample error is given by

$$E_{\text{out}}(h, f) = E_x[e(h(\boldsymbol{x}), f(\boldsymbol{x}))] \tag{1.11}$$

In practice, the expectation in (1.11) is estimated by the average of the pointwise error over a finite set of points outside the training data, namely

$$E_{\text{out}}(h, f) \approx \frac{1}{K}\sum_{k=1}^{K} e(h(\boldsymbol{x}_k), f(\boldsymbol{x}_k)) = \frac{1}{K}\sum_{k=1}^{K} e(h(\boldsymbol{x}_k), y_k), \quad \text{for } (\boldsymbol{x}_k, y_k) \in \text{testing data} \tag{1.12}$$

We remark that for notation simplicity the "f" in both in-sample and out-of-sample error measures are often omitted so they can be written as $E_{\text{in}}(h)$ and $E_{\text{out}}(h)$, respectively.

- Several specific realizations of error measure that are found useful in machine learning are as follows (where only in-sample error measures are considered).

(i) *Classification error*

$$\begin{aligned} E_{in}(h, f) &= \text{fraction of } \mathcal{D} \text{ where } f \text{ and } h \text{ disagree} \\ &= \frac{1}{N}\sum_{n=1}^{N} [\![h(\boldsymbol{x}_n) \neq f(\boldsymbol{x}_n)]\!] \end{aligned} \tag{1.13}$$

where $[\![\text{statement}]\!]$ equals 1 if the statement is true, and equals 0 if the statement is false.

(ii) *L_2 error*

$$E_{in}(h) = \sum_{n=1}^{N} (h(\boldsymbol{x}_n) - y_n)^2 \tag{1.14}$$

(iii) *L_1 error*

V速度 A准确性

$$E_{in}(h) = \sum_{n=1}^{N} | h(\boldsymbol{x}_n) - y_n | \tag{1.15}$$

(iv) *Error measure for logistic regression*

Logistic regression is a type of probabilistic classification model used for predicting a class labels based on input features. It is worth mentioning that as a classifier the target distribution in logistic regression is a Bernoulli distribution, hence it relates to a hypothesis $h(\boldsymbol{x})$ in the form of

$$P(y \mid \boldsymbol{x}) = \begin{cases} h(\boldsymbol{x}) & \text{for } y = 1 \\ 1 - h(\boldsymbol{x}) & \text{for } y = -1 \end{cases} \tag{1.16}$$

Consequently, $h(\boldsymbol{x})$ is restricted to [0, 1] and acts like a probability. For these reasons hypotheses in logistic regression assume the form

$$h(\boldsymbol{x}) = \theta(\boldsymbol{w}^T \boldsymbol{x}) \tag{1.17a}$$

决策函数

where θ is defined by

$$\theta(z) = \frac{e^z}{1+e^z} \tag{1.17b}$$

and is called *logistic function*. As shown in Fig. 1.5, function $\theta(z)$ goes from 0 to 1 monotonically as z goes from $-\infty$ to ∞. We observe that function $\theta(z)$ in (1.17b) obeys $\theta(-z) = 1 - \theta(z)$. This in conjunction with (1.16) and (1.17a) implies that the conditional probability in (1.16) can be expressed as

$$P(y \mid \boldsymbol{x}) = \theta(y\boldsymbol{w}^T \boldsymbol{x}) \tag{1.18}$$

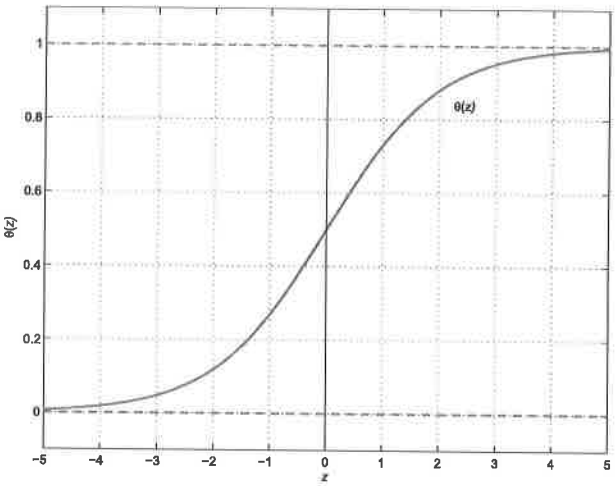


Figure 1.5 Function $\theta(z)$

The error measure for logistic regression is based on *maximum likelihood* (See Sec. A.1.5). Suppose the training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$ are i.i.d., the probability of having this data set is given by

$$\prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n) \tag{1.19}$$

where $P(y \mid \boldsymbol{x})$ is related to hypothesis $h(\boldsymbol{x})$ via (1.17). We maximize the probability in (1.19) in order for $h(\boldsymbol{x})$ to be a good hypothesis. Since maximizing (1.19) is equivalent to minimizing negative logarithm of the probability, in conjunction with (1.18) this leads us to minimizing

$$\frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{P(y_n \mid \boldsymbol{x}_n)}\right) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{\theta(y_n \boldsymbol{w}^T \boldsymbol{x}_n)}\right) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}\right)$$

Therefore, the in-sample error measure for logistic regression problems is defined by

$$E_{in}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}\ln\left(1+e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}\right) \tag{1.20}$$

1.6 Hoeffding Inequality

A. *Hoeffding inequality*

The Hoeffding inequality was introduced early in Example 1.1 for learning the probability of a box of balls with two colors. We recall the Hoeffding inequality

$$P[|\nu-\mu|>\varepsilon] \le 2e^{-2\varepsilon^2 N} \tag{1.1}$$

for any $\varepsilon > 0$, where μ is the probability of (picking a) red ball and ν is the probability of a red ball in a sample of N balls. To see the implication of the inequality in learning, suppose we have a target function $f(\boldsymbol{x})$ and a hypothesis $h(\boldsymbol{x})$. In both training and testing, the event of $h(\boldsymbol{x}_n) = f(\boldsymbol{x}_n)$ is associated with a blue ball and the event of $h(\boldsymbol{x}_n) \ne f(\boldsymbol{x}_n)$ is associated with a red ball. In this way, a learning problem may be connected to the ball model as described in Example 1.2. Moreover, the correspondence between $h(\boldsymbol{x}_n) \ne f(\boldsymbol{x}_n)$ and a red ball implies the correspondence between the in-sample error $E_{\text{in}}(h,f)$ and probability ν, and the correspondence between the out-of-sample error $E_{\text{out}}(h,f)$ and probability μ. Consequently, in the context of machine learning, the Hoeffding inequality implies that

$$P[|E_{\text{in}}(h)-E_{\text{out}}(h)|>\varepsilon] \le 2e^{-2\varepsilon^2 N} \tag{1.21}$$

It should be stressed that (1.21) is valid only if for the case of a hypothesis that was *fixed before* the data set is generated. For a learning problem with a set $\mathcal{H}$ of finite M hypothesis, the question becomes how to bound the probability $P[|E_{\text{in}}(g)-E_{\text{out}}(g)|>\varepsilon]$ where g is the final hypothesis. Since how the final hypothesis is identified depends on the data set used, g is not fixed before the data set is generated, bounding $P[|E_{\text{in}}(g)-E_{\text{out}}(g)|>\varepsilon]$ is not as simple as plugging g into (1.21). In the case of a finite set of M hypothesis in $\mathcal{H} = \{h_1, h_2, \ldots, h_M\}$, the event $|E_{\text{in}}(g)-E_{\text{out}}(g)|>\varepsilon$ occurs if at least one of the events $(|E_{\text{in}}(h_1)-E_{\text{out}}(h_1)|>\varepsilon)$, $(|E_{\text{in}}(h_2)-E_{\text{out}}(h_2)|>\varepsilon)$, …, $(|E_{\text{in}}(h_M)-E_{\text{out}}(h_M)|>\varepsilon)$. Therefore we have

$$P[(|E_{\text{in}}(g)-E_{\text{out}}(g)|>\varepsilon)] \le \sum_{m=1}^{M} P[(|E_{\text{in}}(h_m)-E_{\text{out}}(h_m)|>\varepsilon)]$$

which in conjunction with (1.21) leads to

$$P[(|E_{\text{in}}(g)-E_{\text{out}}(g)|>\varepsilon)] \le 2Me^{-2\varepsilon^2 N} \tag{1.22}$$

Note that the bound in (1.22) is very conservative because the events $(|E_{\text{in}}(h_1)-E_{\text{out}}(h_1)|>\varepsilon)$, $(|E_{\text{in}}(h_2)-E_{\text{out}}(h_2)|>\varepsilon)$, …, $(|E_{\text{in}}(h_M)-E_{\text{out}}(h_M)|>\varepsilon)$ are usually highly overlapping with each other. Another problem with (1.22) is that it makes sense only when the hypothesis set is finite, which is hardly the case for many practical learning problems.

B. *Why do we care about estimations in (1.21) and (1.22)?*

The type of estimations in (1.21) and (1.22) is important because it offers insights of how a good learning strategy could be conceived. In case of a finite hypothesis set $\mathcal{H}$ with a small M, for

example, (1.22) implies with a probability at least $1-\delta$ that

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \sqrt{\frac{1}{2N}\ln\frac{2M}{\delta}} \tag{1.23}$$

Note that given M and δ the second term on the right-hand side of (1.23) can be made arbitrarily small provided that the samples in the training data, N, is large enough. Under this circumstance, (1.23) reveals that the out-of-sample error $E_{\text{out}}(g)$ shall be small as long as the in-sample error $E_{\text{in}}(g)$ is small.

1.7 The Vapnik-Chervonenkis (VC) Dimension

As noted above, (1.23) fails to work for learning systems with infinite hypothesis set $\mathcal{H}$. Besides, for a finite hypothesis set $\mathcal{H}$, (1.23) does not take highly likely overlap among the events $(|E_{\text{in}}(h_1)-E_{\text{out}}(h_1)|>\varepsilon)$, $(|E_{\text{in}}(h_2)-E_{\text{out}}(h_2)|>\varepsilon)$, …, $(|E_{\text{in}}(h_M)-E_{\text{out}}(h_M)|>\varepsilon)$ into account.

The VC dimension is a key quantity in the statistical learning theory [1], [2] that resolves these issues. There are plenty of examples where the "working domain" contains infinitely many items, yet each item can be characterized with a finite (and same) set of basic elements or parameters: infinite number of vectors and points in space R^n, for example, can all be expressed as linear combinations of (the same) n basis vectors.

Definition 1.1 [3] *VC dimension*

In plain terms, the VC dimension of a hypothesis set of the form $\{h(\boldsymbol{x},\boldsymbol{w})\}$ is defined as the largest number of points (in some configuration) that can be shattered by members of $\{h(\boldsymbol{x},\boldsymbol{w})\}$ (a set of points is said to be *shattered* by a hypothesis set $\mathcal{H}$ if, no matter how one assigns a binary label to each point, there exists a member of $\mathcal{H}$ that can perfectly separate them). ■

Example 1.5

Here we consider input space R^2. Suppose $\mathcal{H}$ is the set of perceptrons, i.e., $h(\boldsymbol{x},\boldsymbol{w}) = \text{sign}(w_0 + w_1x_1 + w_2x_2)$, whose decision boundaries are straight lines on the plane. Since the largest number of points a straight line can separate regardless of how these points are labeled is three, the VC dimension of hypothesis set $\mathcal{H} = \{h(\boldsymbol{x},\boldsymbol{w}) = \text{sign}(w_0 + w_1x_1 + w_2x_2)\}$, denoted by d_{vc}, is three. Note that the VC dimension in this case coincides with the number parameters that characterize the hypothesis set. ■

This example indicates that although the number of hypothesis in $\mathcal{H}$ is infinite, depending on the "complexity" of the hypothesis set, its VC dimension might be finite, and it measures the effective parameters or degree of freedom of the hypothesis set.

Example 1.6

The d-dimensional perceptrons $\{h(\boldsymbol{x},\boldsymbol{w}) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x})\}$ with $\boldsymbol{x} = [1 \quad x_1 \quad \cdots \quad x_d]^T$ is $d_{\text{vc}} = d + 1$ [8]. ■

The importance of the notion of VC dimension may be appreciated via the following theorem which provides a generalization bound that is way better than (1.23).

Theorem *VC generalization bound* [8]

For any tolerance $\delta > 0$,

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}} \tag{1.24}$$

with probability $\ge 1-\delta$, where $m_{\mathcal{H}}$ is called the *growth function* for hypothesis set $\mathcal{H}$, which is shown to be a polynomial in N and bounded by

$$m_{\mathcal{H}}(N) \le N^{d_{vc}} + 1 \qquad \blacksquare \tag{1.25}$$

By combining (1.24) with (1.25), we obtain a generalization bound that is directly related the VC dimension d_{vc} as

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\frac{4(2N)^{d_{vc}}+4}{\delta}} \tag{1.26}$$

Several remarks on (1.24) and (1.26) are now in order.

- Bounds in (1.24) and (1.26) are widely applicable as they are valid for all hypothesis sets, learning algorithms, input spaces, probability distributions, and binary target functions. As such, however, these bounds turn out to be quite loose. On all accounts, there are two points worth mentioning: First, although the bounds are loose, they tend to be *equally* loose for different learning models, hence the bounds remain useful for comparing the performance of these models. Second, analysis based on VC dimension is found useful in practice because learning models with lower d_{vc} are found to generalize better than those with higher d_{vc}. Moreover, requiring N (the number of example in training data) to be at least $10 \times d_{\text{vc}}$ for decent performance has been as a popular rule of thumb [8].
- A very important contribution of the bounds in (1.24) and (1.26) is that they establish the feasibility of learning models with infinite hypothesis sets. Specifically, these bounds assert that as long as the VC dimension of the hypothesis $\mathcal{H}$ is finite, the generalization error can be made arbitrarily close to the in-sample error when sufficiently many training examples are utilized. The generalization performance can be characterized by parameters ε and δ where ε is the error tolerance of $E_{\text{out}}(g)$ deviating from $E_{\text{in}}(g)$, and the confidence parameter δ denotes the probability that the error tolerance ε is violated. By the definition of ε and (1.24), it follows that

$$\sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}} \le \varepsilon$$

which leads to

$$N \ge \frac{8}{\varepsilon^2}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta} \tag{1.27}$$

The counterpart of (1.27) for the bound in (1.26) is given by

$$N \ge \frac{8}{\varepsilon^2}\ln\frac{4(2N)^{d_{vc}}+4}{\delta} \tag{1.28}$$

These lower bounds for N describe *sample complicity* for achieving desirable generalization performance (ε,δ). As expected, these bounds are quite loose, namely the number of examples needed to achieve the performance are way more than necessary. Nevertheless, the bounds are explicitly indicative of the feasibility of learning as long as the VC dimension d_{vc} is *finite*.

- Let the second term on the right-hand side of (1.24) be denoted by $\Omega(N,\mathcal{H},\delta)$, namely

$$\Omega(N,\mathcal{H},\delta)=\sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}}\leq\sqrt{\frac{8}{N}\ln\frac{4(2N)^{d_{vc}}+4}{\delta}} \tag{1.29}$$

so we can write

$$E_{\text{out}}(g)\leq E_{\text{in}}(g)+\Omega(N,\mathcal{H},\delta) \tag{1.30}$$

In a typical learning problem the training data $\mathcal{D}$ is given, hence N is fixed. Now suppose the confidence parameter δ is also given. The term $\Omega(N,\mathcal{H},\delta)$ in this case is determined by the VC dimension hence the model complexity. As such, $\Omega(N,\mathcal{H},\delta)$ serves as a *penalty* for the generalization performance. On the other hand, increasing model complexity (i.e. allowing $\mathcal{H}$ to include more hypotheses) in general reduces the in-sample error because with a richer $\mathcal{H}$ we are likely to fit the training data better. Therefore, we are in a scenario of having an increasing $\Omega(N,\mathcal{H},\delta)$ and a decreasing $E_{\text{in}}(g)$ as the model complexity goes up, in other words the generalization performance depends on the interplay between the two factors. This situation is illustrated in Fig. 1.6 [8].

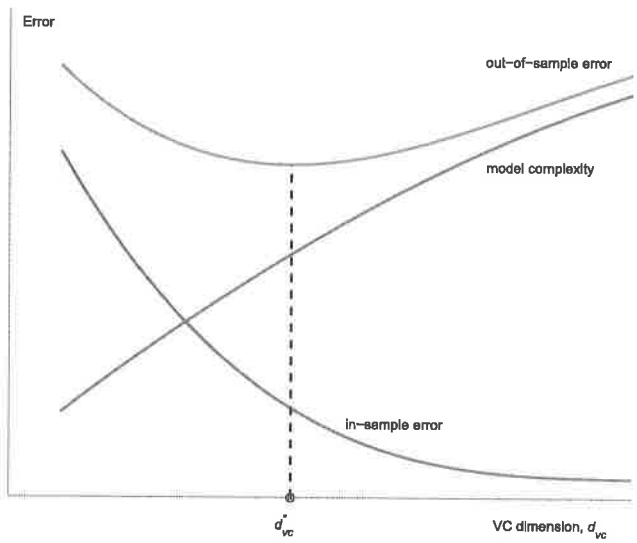


Figure 1.6 As VC dimension increases, in-sample error reduces while penalty $\Omega(N,\mathcal{H},\delta)$ due to the increase of model complexity goes up. As a result, the out-of-sample error will not be small if the hypothesis set $\mathcal{H}$ is too small or too large. The optimal model with VC dimension d^*_{vc} is a compromise that minimizes the combination of $E_{\text{in}}(g)$ and $\Omega(N,\mathcal{H},\delta)$ [8].

1.8 Bias-Variance Decomposition

The VC analysis provides a generalization bound as a sum of $E_{\text{in}}(g)$ plus a penalty term $\Omega(N,\mathcal{H},\delta)$. In this section, we study generalization performance through another approach known as *bias-variance decomposition* [3][7][8]. The approach is well suited for L_2 error measures and the results are statistically intuitive. The following analysis follows [8].

From (1.11) and (1.14), it follows that when the L_2-error is employed the out-of-sample error for

the final hypothesis $g(\boldsymbol{x})$ is given by

$$E_{\text{out}}(g^{(\mathcal{D})}) = E_{\boldsymbol{x}}\left[\left(g^{(\mathcal{D})}(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2\right] \tag{1.31}$$

where the notation $g^{(\mathcal{D})}$ stresses that the final hypothesis depends on training data $\mathcal{D}$. To obtain a generalization performance bound that is independent of any particular selection of $\mathcal{D}$, we compute the expectation of $E_{\text{out}}(g^{(\mathcal{D})})$ with respect to $\mathcal{D}$. From (1.31), we can write

$$\begin{aligned} E_{\mathcal{D}}\left[E_{\text{out}}(g^{(\mathcal{D})})\right] &= E_{\mathcal{D}}\left[E_{\boldsymbol{x}}\left[\left(g^{(\mathcal{D})}(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2\right]\right] \\ &= E_{\boldsymbol{x}}\left[E_{\mathcal{D}}\left[\left(g^{(\mathcal{D})}(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2\right]\right] \\ &= E_{\boldsymbol{x}}\left[E_{\mathcal{D}}[g^{(\mathcal{D})}(\boldsymbol{x})^2] - 2E_{\mathcal{D}}[g^{(\mathcal{D})}(\boldsymbol{x})]f(\boldsymbol{x}) + f(\boldsymbol{x})^2\right] \end{aligned} \tag{1.32}$$

where $E_{\mathcal{D}}[g^{(\mathcal{D})}(\boldsymbol{x})] = \bar{g}(\boldsymbol{x})$ represents an average function where the average is performed with respect to all possible data set of size N. Obviously $\bar{g}(\boldsymbol{x})$ may be approximated by generating enough number of data sets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$, applying the learning algorithm to each $\mathcal{D}_k$ to obtain $g^{(\mathcal{D}_k)}(\boldsymbol{x})$, and computing $\bar{g}(\boldsymbol{x}) \approx \frac{1}{K}\sum_{k=1}^{K} g^{(\mathcal{D}_k)}(\boldsymbol{x})$. It now follows from (1.32) that

$$\begin{aligned} E_{\mathcal{D}}\left[E_{\text{out}}(g^{(\mathcal{D})})\right] &= E_{\boldsymbol{x}}\underbrace{\left[\left(\bar{g}(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2\right]}_{\text{bias}(\boldsymbol{x})} + E_{\boldsymbol{x}}\underbrace{\left[E_{\mathcal{D}}\left[\left(g^{(\mathcal{D})}(\boldsymbol{x}) - \bar{g}(\boldsymbol{x})\right)^2\right]\right]}_{\text{variance}(\boldsymbol{x})} \\ &= E_{\boldsymbol{x}}[\text{bias}(\boldsymbol{x})] + E_{\boldsymbol{x}}[\text{var}(\boldsymbol{x})] \\ &= \text{bias} + \text{var} \end{aligned} \tag{1.33}$$

In the case of a small hypothesis set $\mathcal{H}$, the chances of finding a $g(\boldsymbol{x})$ from $\mathcal{H}$ that well approximates the target function $f(\boldsymbol{x})$ are not great, hence the bias is expected to be large. Also, when $\mathcal{H}$ contains small number of candidate hypotheses, the chances of having a large difference between $\bar{g}(\boldsymbol{x})$ and $g^{(\mathcal{D}_k)}(\boldsymbol{x})$ is not great either, hence the variance in this case is expected to be small. As the model complexity increases, the approximation of target function by the average final hypothesis is likely to improve, thus the bias term in (1.33) will in general reduce. On the other hand, as more candidate hypotheses are included in $\mathcal{H}$, different training data sets are more likely to lead to different final hypotheses, hence an increased variance. Eq. (1.33) indicates that the average out-of-sample error depends on the model complexity through both the bias and variance terms as quantified by (1.33). We also remark that as functions of model complexity, bias and especially variance are not necessarily monotonic. In fact the variance can be viewed as a measure of instability in the sense that it reflects the sensitivity of learning outcomes to small variations in the data.

Example 1.7

This example is aimed at illustrating the bias-variance decomposition. A total of $K = 400$ sets of training data are generated, each consists of $N = 10$ points $\{(x_n, y_n), n = 1, 2, \ldots, 10\}$ where x_n are sampled uniformly randomly from interval [0, 1], and y_n's were produced by target function $y_n = f(x_n)$ where

$$f(x) = \cos(2\pi x) + 0.5\sin(3.5\pi x)$$

We examine ten learning models, each consists of all mth-order polynomials, with $m = 0, 1, \ldots, 9$. For a given hypothesis set $\mathcal{H}_m$ of mth-order polynomials of the form

$$h(x, \boldsymbol{w}) = w_0 + w_1 x + \cdots + w_m x^m \tag{1.34}$$

the final hypothesis was selected by minimizing the in-sample error measure

$$E_{\text{in}}(h, \boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}\left(h(x_n, \boldsymbol{w}) - y_n\right)^2$$

with respect to $\boldsymbol{w}$, where $\{(x_n, y_n), n = 1, 2, \ldots, 10\}$ is one of the training data set, say $\mathcal{D}^{(k)}$. Let $\boldsymbol{w}^*$ be the minimizer obtained, the final hypothesis is then identified as $g^{(\mathcal{D}_k)}(x) = h(x, \boldsymbol{w}^*)$. In this way, the average final hypothesis $\bar{g}(\boldsymbol{x})$ is found to be

$$\bar{g}(\boldsymbol{x}) = \frac{1}{400}\sum_{k=1}^{400} g^{(\mathcal{D}_k)}(x)$$

and the bias-variance decomposition (1.33) was carried out to obtain the bias and variance for that particular value of m which is obviously an indicator of the model complexity. The simulations were repeated for each of $m = 0, 1, \ldots, 9$, and the bias, variance as well as sum of the two quantities as average out-of-sample error with respect to m are depicted in Fig. 1.7.

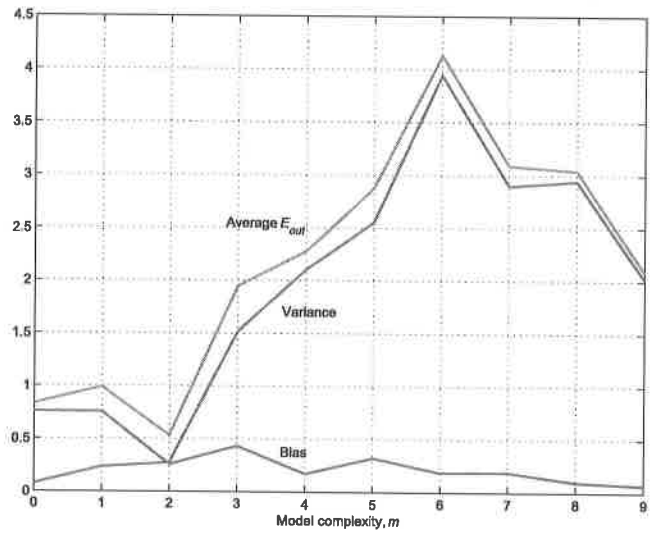


Figure 1.7 Bias and variance versus model complexity for Example 1.7.

We observe from the figure that the hypothesis set consisting of 2nd-oder polynomials appears to be a good choice as it offers a reasonable bias and is relatively insensitive to the variations in training data. This observation is confirmed by Fig. 1.8, where the 400 2nd-order final hypotheses (in blue) are shown in comparison with the target function $f(x)$ (in red). For comparison, Fig. 1.9 displays 400 6th-order final hypotheses versus the target function where the model is shown to be highly unstable against the variations in training data. The average final hypotheses $\bar{g}(\boldsymbol{x})$ obtained for the 2nd-order and 6th-order models versus target function are shown in Figs. 1.10(a) and (b), respectively. The MATLAB code for the simulations described in Example 1.7 is ex1_7.m which can be found at the end of this chapter.

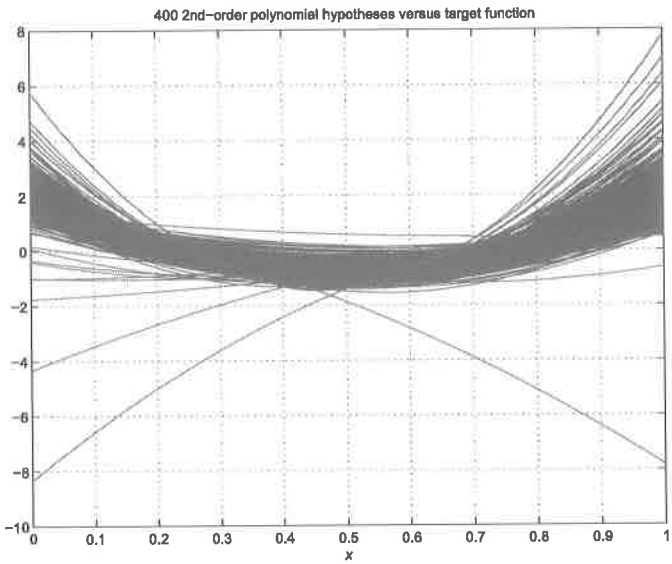


Figure 1.8 Second-order polynomial final hypotheses (in blue) versus target function (in red).

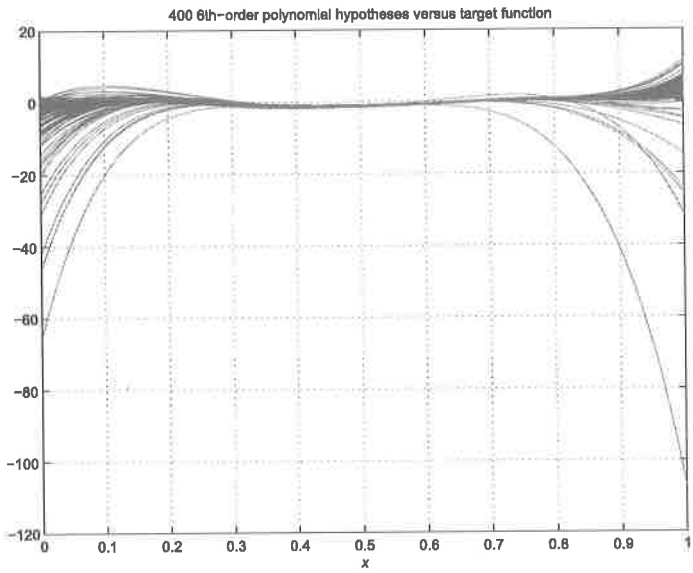
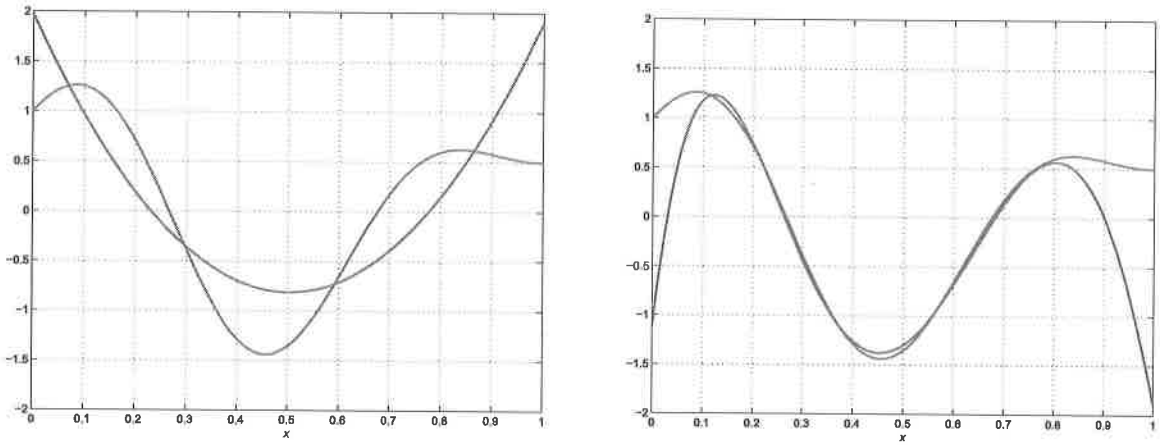


Figure 1.9 Sixth-order polynomial final hypotheses (in blue) versus target function (in red).

1.9 Expected Errors versus Number of Data Points

Through the VC bounds and bias-variance decomposition we have seen the general behavior of in-sample and out-of-sample errors with respect to model complexity. The VC bound is explicitly related to the size N of the training data, and it is also useful for analyzing the behavior of E_{in}, E_{out}, as well as their relation as N varies. In summary, we can state the following:

- Regardless of model complexity, as N gets larger E_{out} reduces and the difference between E_{out} and E_{in} narrows.

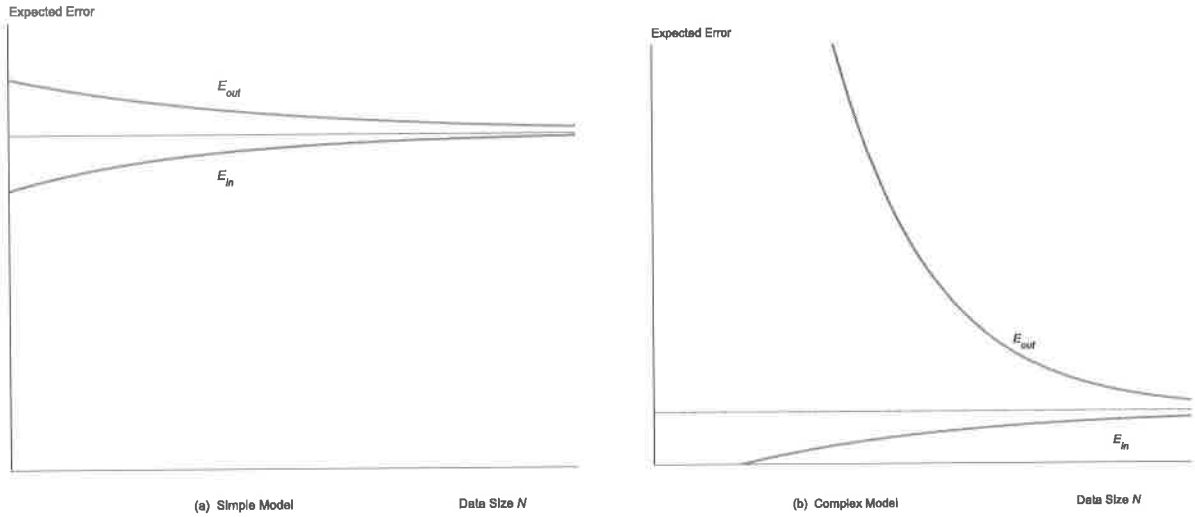


(a) (b)

Figure 1.10 Average final hypotheses $\bar{g}(\boldsymbol{x})$ obtained from (a) 2nd-order and (b) 6th-order models (in blue) versus target function (in red).

- Regardless of model complexity, the in-sample error will increase as size N increases because it becomes increasingly difficult to minimize in-sample error measure that has to take care of individual errors at more data points. However, since the data points are governed by a underlining target function $f(\boldsymbol{x})$ (or a conditional probability distribution), errors at data points are not at all random but rather coherent. As a matter of fact, as $N \to \infty$ the in-sample error will converge to $E_x[e(g(\boldsymbol{x}), f(\boldsymbol{x}))]$ (such as $E_x[|g(\boldsymbol{x}) - f(\boldsymbol{x})|^2]$). In other words, the in-sample error will reach a steady state as the sample size continues to grow.
- With a simple model (i.e. a small VC dimension), E_{out} will quickly go down to approach the level of E_{in}. However, it is usually very hard for a simple hypothesis to fit a large number of data points, thus the steady-state level of the in-sample error (as well as out-of-sample error) will be high.
- Opposite to the above case, with a complex model (hence a larger VC dimension), E_{out} will approach the level of E_{in} much more slowly. However, as data size N increases the steady-state level for both in-sample and out-of-sample errors are at a lower level.

The points made above are illustrated in Fig. 1.11a for the case of a simple model and Fig. 1.11b for the case of a complex model. In the literature, the type of curves in Fig. 1.11 are called learning curves [8].



(a) (b)

Figure 1.11 Expected in-sample and out-of-sample errors vs. data size N for (a) simple model and (b) complex model [8].

MATLAB Code Used in This Chapter

(1) File name: ex1_3_PLA.m

```
% Perceptron learning algorithm (PLA) as applied to  training data in Example 1.3.
% Example: load data_ex2_3; [wt,t] = ex1_3_PLA(x,y,xp,xn,[0 0 0]',9);
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 1, 2015.
function [wt,t] = ex1_3_PLA(x,y,xp,xn,w0,st)
N = length(y);
t = 0;
wt = w0(:);
D = [ones(N,1) x'];
y = y(:);
flag = 0;
rand('state',st)
while flag == 0,
   Nw = randperm(N);
   tu = 0;
   tv = 0;
   while tu < N & tv == 0,
       ti = Nw(tu + 1);
       if sign(D(ti,:)*wt) ~= y(ti),
         tv = 1;
         t_ind = ti;
       end
       tu = tu + 1;
   end
   if tv == 0,
     flag = 1;
   else
     wt = wt + y(t_ind)*D(t_ind,:)';
     t = t + 1;
   end
end
z = sign(D*wt) - y;
ind = find(z ~= 0);
L = length(ind);
disp('number of perceptron iterations:')
t
disp('final weights:')
wt
disp('in-sample error:')
length(ind)/N
figure(1)
subplot(121)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
grid
xlabel('(a)  \itx_1 (years in residence)')
ylabel('\itx_2 (salary in 10K dollars)')
```

```
axis([0 12 0 8])
axis square
title('Training Data')
hold off
subplot(122)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
grid
xlabel('(b)  \itx_1 (years in residence)')
ylabel('\itx_2 (salary in 10K dollars)')
p1 = 0;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 12;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([0 12 0 8])
axis square
title('Decision Boundary Produced by PLA')
hold off
```

(2) File name: ex1_3_adaline.m

```
% Adaptive linear neuron (Adaline) algorithm as applied to training data in Example 1.3.
% Example: load data_ex2_3;  [wt,t] = ex1_3_adaline(x,y,xp,xn,[0 0 0]',0.03,9);
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 1, 2015.
function [wt,t] = ex1_3_adaline(x,y,xp,xn,w0,et,st)
N = length(y);
t = 0;
wt = w0(:);
D = [ones(N,1) x'];
y = y(:);
flag = 0;
rand('state',st)
while flag == 0,
   Nw = randperm(N);
   tu = 0;
   tv = 0;
   while tu < N & tv == 0,
        ti = Nw(tu + 1);
        if y(ti)*(D(ti,:)*wt) <= 1,
          tv = 1;
          t_ind = ti;
        end
        tu = tu + 1;
   end
   if tv == 0,
     flag = 1;
   else
     yi = y(t_ind);
     di = D(t_ind,:);
```

```
      wt = wt + et*(yi-di*wt)*di';
      t = t + 1;
    end
end
z = sign(D*wt) - y;
ind = find(z ~= 0);
L = length(ind);
disp('number of perceptron iterations:')
t
disp('final weights:')
wt
disp('in-sample error:')
length(ind)/N
figure(1)
subplot(121)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
grid
xlabel('(a)  \itx_1 (years in residence)')
ylabel('\itx_2 (salary in 10K dollars)')
axis([0 12 0 8])
axis square
title('Training Data')
hold off
subplot(122)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
grid
xlabel('(b)  \itx_1 (years in residence)')
ylabel('\itx_2 (salary in 10K dollars)')
p1 = 0;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 12;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([0 12 0 8])
axis square
title('Decision Boundary Produced by Adaline')
hold off
```

(3) File name: ex1_7.m

```
% Bias-variance decomposition as discussed in Example 1.7.
% Example:  [bias,variance] = ex1_7(10,9,400,5);
% Written by W.-S. Lu, University of Victoria.
% Last modified: Dec. 15, 2014.
function [bias,variance] = ex1_7(N,M,K,st)
xe = 0:1/999:1;
xe = xe(:);
```

```
f = cos(2*pi*xe) + 0.5*sin(3.5*pi*xe);
bias = zeros(M+1,1);
variance = bias;
e_out = bias;
rand('state',st)
S = rand(N,K);
p = max(N,M+1);
if p > N,
  a = ones(p-N,1);
else
  a = [];
end
for t1 = 1:(M+1),
   gb = 0;
   Gx = [];
   for t2 = 1:K,
      x = S(:,t2);
      xx = [x; a];
      V = fliplr(vander(xx));
      X = V(1:N,1:t1);
      y = cos(2*pi*x) + 0.5*sin(3.5*pi*x);
      gw = flipud(pinv(X)*y);
      gb = gb + gw;
      gx = polyval(gw,xe);
      Gx = [Gx gx];
   end
   gb = gb/K;
   gbx = polyval(gb,xe);
   vw = 0;
   for i = 1:K,
      vw = vw + (Gx(:,i) - gbx).^2;
   end
   variance(t1) = mean(vw/K);
   bias(t1) = mean((gbx - f).^2);
   e_out(t1) = bias(t1) + variance(t1);
end
figure(1)
m = 0:1:M;
plot(m,bias,'b-','linewidth',1.5)
hold on
plot(m,variance,'r-','linewidth',1.5)
plot(m,e_out,'m-','linewidth',1.5)
xlabel('Model complexity, \itm')
grid
hold off
```

Problems

1.1 [8] A sample of heads and tails is created by tossing a coin a number of times. Suppose we have a number of coins that generate different samples independently. For a given coin, let the probability of heads (we call it probability of error) be μ. The probability of obtaining k heads in N tosses of this coin is known to obey the binomial distribution given by

$$P[k \mid N, \mu] = \binom{N}{k} \mu^k (1-\mu)^{N-k}$$

where

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}$$

Remember the training error is $\nu = \frac{k}{N}$.

(i) Assume the sample size is N = 10. If all coins have the same $\mu = 0.05$, compute the probability that at least one coin will have $\nu = 0$ for the case of 1 coin, 1000 coins, and 1,000,000 coins. Repeat the problem for $\mu = 0.8$.

(ii) For the case of N = 6 and 2 coins with $\mu = 0.5$ for both coins, plot the probability $P[\max_{i=1,2} |\nu_i - \mu| > \varepsilon]$ for ε in the range [0, 1]. On the same plot, depict the bound that would be obtained using the Hoeffding inequality. Remember that for a single coin, the Hoeffding bound is given by

$$P[|\nu - \mu| > \varepsilon] \le 2e^{-2N\varepsilon^2}$$

(Hint: To evaluate $P[\max\ldots]$, use $P[A \text{ or } B] = P[A] + P[B] - P[A \text{ and } B] = P[A] + P[B] - P[A]P[B]$ if A and B are independent events.)

1.2 [8] In this problem we derive a form of the law of large numbers, known as the *Chebyshev bound*, which is a bound that decreases linearly in N.

(i) Let z be a nonnegative random variable and t be a positive constant. Show that

$$P[z \ge t] \le \frac{E[z]}{t}$$

(ii) Let u be a random variable with mean μ and variance σ^2, show that for any $t > 0$

$$P[(u-\mu)^2 \ge t] \le \frac{\sigma^2}{t}$$

(iii) Let $u_1, u_2, \ldots, u_N$ be i.i.d. random variables, each with mean μ and variance σ^2, and $u = \frac{1}{N}\sum_{n=1}^{N} u_n$. Show that

$$P[(u-\mu)^2 \ge t] \le \frac{\sigma^2}{N \cdot t}$$

1.3 [8] In this problem we derive a form of the law of large numbers which is an *exponential* bound known as the *Chernoff bound*.

(i) Let z be a random variable, t be a positive constant, and s be a positive parameter. Show that

$$P[z \geq t] \leq e^{-st} E_z[e^{sz}]$$

(Hint: Use the result from part (i) of Prob. 1.2 and the fact that e^{sz} is monotonically increasing in z.)

(ii) Let $u_1, u_2, \ldots, u_N$ be i.i.d. random variables and $u = \frac{1}{N}\sum_{n=1}^{N} u_n$. Show that for any n

$$P[u \geq t] \leq \left(e^{-st} E_{u_n}[e^{su_n}]\right)^N$$

(iii) Suppose $P[u_n = 0] = P[u_n = 1] = \frac{1}{2}$. Evaluate $E_{u_n}[e^{su_n}]$ as a function of s and minimize $e^{-st} E_{u_n}[e^{su_n}]$ with respect to s for fixed t with $0 < t < 1$.

(iv) Based on the results from part (iii), show that for $0 < \varepsilon < \frac{1}{2}$

$$P[u \geq E(u) + \varepsilon] \leq 2^{-\beta N}$$

where $0 < \varepsilon < \frac{1}{2}$ $\beta = 1 + (\frac{1}{2} + \varepsilon)\log_2(\frac{1}{2} + \varepsilon) + (\frac{1}{2} - \varepsilon)\log_2(\frac{1}{2} - \varepsilon)$ and $E(u) = \frac{1}{2}$.

1.4 [8] In this problem you are asked to create a binary-valued target function and a data set $\mathcal{D}$. The input space is $\mathcal{X} = R^2$.

(i) Choose a random line in the plane as your target function. Describe it by equation $w_0 + w_1 x_1 + w_2 x_2 = 0$. Report the numerical values of w_i for $i = 0, 1, 2$.

(ii) Generate a random data set of size $N = 20$. To ensure a meaningful problem for perceptron learning, make sure the data points appear on both sides of the straight line created in part (i). Map each data point on one side of the line to +1, and those on the other side of the line to –1. In this way, you have a training data $\{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, 20\}$.

(iii) Apply the PLA to the data set. Report the numerical values of the initial weight vector $\boldsymbol{w}(0)$ and the algorithm's convergence in terms of number of updates. Report the numerical values of the final hypothesis obtained from PLA. Comment on its performance in terms of how well it separates the data set (for this purpose, it is best to include a plot) and how close it is to the true target function.

1.5 [8]

(i) Repeat Problem 1.4 using another randomly generated data set of size $N = 100$.

(ii) Repeat Problem 1.4 using another randomly generated data set of size $N = 1000$.

(iii) Comment on the results obtained from parts (i) and (ii) with respect to solution accuracy and running time.

1.6 [8] Repeat Problem 1.4 using the Adaline algorithm.

1.7 [8] Suppose one was asked to measure the length of the longer side of a cell phone, a total of N times, that yield N data points. The measurements are then indexed in an ascending manner as $y_1 \leq y_2 \leq \cdots \leq y_N$. Now one tries to figure out a representative value of the length, which in the language of machine learning means a hypothesis set $\mathcal{H}$ that contains all real-valued constants.

(i) Show that the hypothesis $h \in \mathcal{H}$ (i.e. a constant) that minimizes the L_2 in-sample error (which is the sum of squared deviations)

$$E_{in}(h) = \sum_{n=1}^{N} (h - y_n)^2$$

is the mean value of the measurements

$$h_{\text{mean}} = \frac{1}{N} \sum_{n=1}^{N} y_n$$

(ii) Find a hypothesis $h \in \mathcal{H}$ that minimizes the L_1 in-sample error (which is the sum of absolute deviations)

$$E_{in}(h) = \sum_{n=1}^{N} | h - y_n |$$

Show that the solution is the in-sample *median*, denoted by h_{med}, which is any value for which half of the data points are at most h_{med} and half of the data points are at least h_{med}.

(iii) If data point y_N is perturbed to $y_N + \varepsilon$ where ε becomes increasingly large so that data point y_N becomes an *outlier*. Under the circumstances, what will happen to the two solutions h_{mean} and h_{med}?

1.8 [8] Suppose $m_{\mathcal{H}}(N) = N + 1$ so $d_{\text{vc}} = 1$ and you have $N = 100$ training examples. Use the VC bound in (1.26) to deduce a bound for E_{out} with confidence 90% (i.e., $\delta = 0.1$). Repeat the problem with $N = 10{,}000$.

1.9 [8] It can be shown that

$$P\left[\frac{E_{\text{out}}(g) - E_{in}(g)}{\sqrt{E_{\text{out}}(g)}} > \varepsilon \right] \le c \cdot m_{\mathcal{H}}(2N) \cdot e^{\frac{-\varepsilon^2 N}{4}} \qquad \text{(P1.1)}$$

where c is a constant slightly greater than 6. The above bound is useful when one cares about *relative* generalization error.

(i) Use (P1.1) to show that with probability $1 - \delta$ the generalization bound

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \frac{\xi}{2}\left(1 + \sqrt{1 + \frac{4E_{\text{in}}(g)}{\xi}}\right)$$

holds, where $\xi = \frac{4}{N} \ln \frac{c \cdot m_{\mathcal{H}}(2N)}{\delta}$.

(ii) Show that $E_{\text{out}}(g) \to E_{\text{in}}(g)$ as $N \to \infty$. Hint: Show that

$$\frac{\xi}{2}\left(1 + \sqrt{1 + \frac{4E_{\text{in}}(g)}{\xi}}\right) \to 0 \text{ as } N \to \infty$$

1.10 [8] When there is noise in the data, $E_{\text{out}}(g^{(\mathcal{D})}) = E_{x,y}\left[\left(g^{(\mathcal{D})}(\boldsymbol{x}) - y(\boldsymbol{x})\right)^2\right]$ where $y(\boldsymbol{x}) = f(\boldsymbol{x}) + \varepsilon$. Suppose we model the noise ε as a random variable with mean zero and variance σ^2, show that the bias-variance decomposition becomes

$$E_{\mathcal{D}}\left[E_{\text{out}}(g^{(\mathcal{D})})\right] = \text{bias} + \text{var} + \sigma^2$$

1.11 [8] Consider a simple learning problem where the input space is one-dimensional; the input variable x is uniformly distributed in [–1, 1]; the data set consists of 2 points $\{x_1, x_2\}$; and the

target function is $f(x) = x^2$ thus the full data set is $\mathcal{D} = \{(x_1, x_1^2), (x_2, x_2^2)\}$. The hypothesis set $\mathcal{H}$ consists of functions of the form $h(x) = ax + b$, and the learning algorithm returns with a straight line as g that tries to fit these two data points. We are interested in the test performance E_{out} of the learning algorithm with respect to squared error measure, bias, and variance.

(i) Derive the analytic expression for the average function $\bar{g}(x)$.

(ii) Describe an experiment that you would run to numerically determine the $\bar{g}(x)$, E_{out}, bias, and variance.

(iii) Run the experiment and report your results. Compare E_{out} with bias + variance. Provide a plot of $\bar{g}(x)$ and $f(x)$ (in the same plot).

(iv) Compute analytically what E_{out}, bias, and variance should be.

Chapter 2 Linear Models

This chapter aims at developing linear models for learning from data. Specifically we study linear models for classification, regression, and probability estimation. The materials presented in this chapter are primarily based on [8].

3.1 Linear Models for Classification

A. *The Model and Why It should Work*

The hypothesis set $\mathcal{H}$ for classifying data into two classes can be characterized by

$$h(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x}) \tag{2.1}$$

where weight vector $\boldsymbol{w} \in R^{d+1}$ parameterizes the hypothesis set, and feature vector assumes the form $\boldsymbol{x} = [1 \quad x_1 \quad \cdots \quad x_d]^T$. The binary classification problem can be stated as follows: Given a training set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,...,N\}$ where $y_n \in \{1, -1\}$, find a weight vector $\boldsymbol{w}^*$ such that hypothesis $g(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^{*T}\boldsymbol{x})$ generalizes well over the input space $\mathcal{X}$.

A natural question arising here is whether the linear model in (2.1) can offer a satisfactory solution in terms of small generalization errors? To this end we recall the VC bound from Chapter 1, namely,

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\frac{4(2N)^{d_{\text{vc}}}+4}{\delta}}$$

where VC dimension d_{vc} for linear model (2.1) is known to be $d + 1$, hence we obtain

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + O\left(\sqrt{\frac{d}{N}\ln N}\right) \tag{2.2}$$

Although the VC bound as shown in (2.2) is not tight, it remains instructive that good generalization performance can be assured if the number of samples in the training set is sufficiently large and the in-sample error E_{in} is small.

B. *Finding $\boldsymbol{w}^*$ by the Pocket Algorithm*

Suppose we are given a training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,...,N\}$ with N sufficiently large. From the analysis made above, our focus is to find a $\boldsymbol{w}^*$ that makes E_{in} small, where the error measure is given by

$$E_{\text{in}}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}[\![\text{sign}(\boldsymbol{w}^T\boldsymbol{x}_n) \ne y_n]\!] \tag{2.3}$$

which is essentially the percentage of misclassified samples in data set $\mathcal{D}$. To this end, the perceptron learning algorithm (PLA) and *Adaline* from Chapter 1 are two candidate algorithms as they produce zero E_{in} as long as the data are linearly separable. The problems is, unfortunately, PLA as well as *Adaline* do not converge for non-separable data sets, but the data sets in many practical machine learning problems are non-separable.

A variant of the PLA, called *pocket algorithm* (PA), runs PLA iterations but keeps only the best

weight vector achieved so far in its "pocket", and produces the best weight vector obtained after a given number of iterations as the final hypothesis. In this way, the pocket algorithm is able to deal with non-separable data sets with improved performance.

The Pocket Algorithm (PCA)

Input: A training data set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, ..., N\}$ and an initial $\boldsymbol{w}(0)$. Compute $E_{\text{in}}(\boldsymbol{w}(0))$. Set $k = 0$.

For $k = 0, 1, \ldots, K-1$, do

Step 1: Do one PLA iteration, namely, pick one of the misclassified pairs *at random*, denote it by $(\boldsymbol{x}(k), y(k))$, and calculate

$$\tilde{\boldsymbol{w}} = \boldsymbol{w}(k) + y(k)\boldsymbol{x}(k) \tag{2.4}$$

Step 2: Evaluate $E_{in}(\tilde{\boldsymbol{w}})$.

If $E_{in}(\tilde{\boldsymbol{w}}) < E_{in}(\boldsymbol{w}(k))$, update $\boldsymbol{w}(t)$ to $\boldsymbol{w}(k+1) = \tilde{\boldsymbol{w}}$, else set $\boldsymbol{w}(k+1) = \boldsymbol{w}(k)$.

■

Example 2.1

Apply PLA and PCA to a non-separable training data set of 2000 samples known as "double semi-circle". As shown in Fig. 2.1, each semi-circle contains 1000 random samples. Each semi-circle is characterized by radius r and thickness *thk*. Each sample in the upper semi-circle is labeled by $y = -1$ while each sample in the lower semi-circle is labeled by $y = 1$. The center of the upper semi-circle is aligned with the middle of the edge of the lower semi-circle. If the gap between the edges of the two semi-circles is denoted by *sep*, then evidently a positive *sep* means the samples in the upper semi-circle are linearly separable from the samples in the lower semi-circle, and a negative *sep* means they are not. In this example, we are deal with non-separable training data with $r = 10$, $thk = 5$, and $sep = -1$.

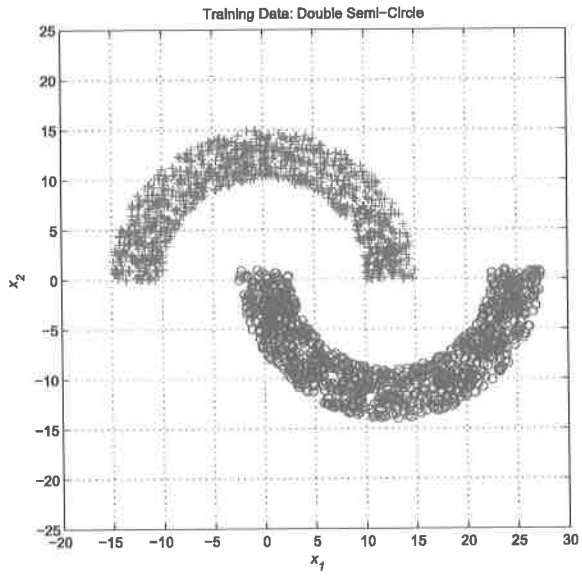


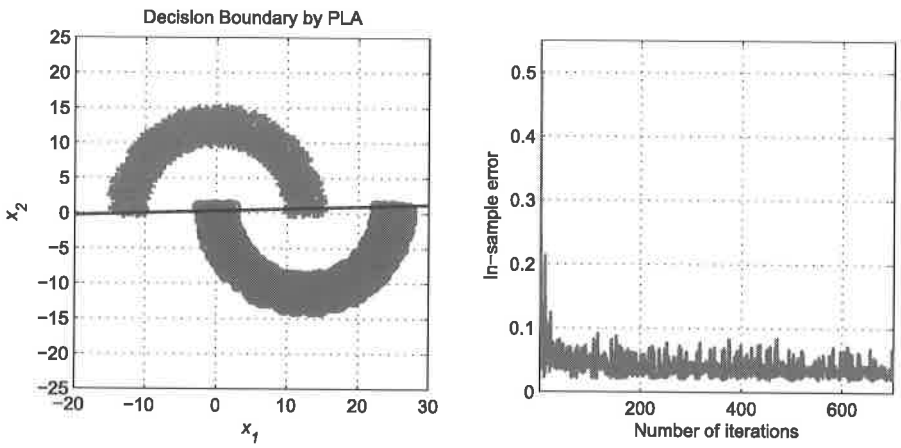
Figure 2.1 A data set of 2000 random samples in double semi-circle with $r = 10$, $thk = 5$, and $sep = -1$.

The MATLAB function producing the training data and Fig. 2.1 is data_semi_circle.m which is

enclosed at the end of this chapter.

Because the training data is non-separable, the PLA does not converge. Furthermore, because the PLA is not a descent algorithm, in-sample error typically goes down and up in a randomly oscillatory fashion. With initial weight w0 = zeros(3,1), the PLA was applied to the above data set. The in-sample error after 701 iterations was found to be $E_{\text{in}} = 0.0190$. Out of $N = 2000$ sample points, 1962 points were classified correctly. The in-sample error obtained versus iteration number is shown in Fig. 2.2b. Over the first 701 iterations, the smallest in-sample error occurs at the 701th iteration which produces the weight

$$\boldsymbol{w}(701) = \begin{bmatrix} 47 & 4.4703 & -148.2111 \end{bmatrix}^T$$



(a) (b)

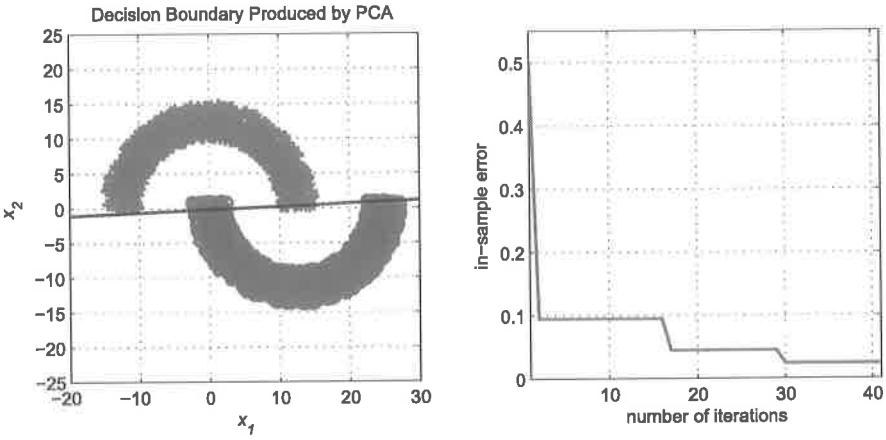
Figure 2.2 PLA applied to the double semi-circle data set (a) decision boundary obtained after 701 iterations, (b) in-sample error pattern during the first 701 PLA iterations.

Next, with the same initial weight w0 = zeros(3,1), the PCA was applied to the same data set with 40 iterations. The in-sample error obtained versus iteration number is shown in Fig. 2.3b. Notice the non-increasing in-sample error pattern offered only by PCA. The in-sample error after 40 iterations was found to be $E_{\text{in}} = 0.0240$. Out of $N = 2000$ sample points, 1952 points were classified correctly. The final weight vector was obtained as

$$\boldsymbol{w}(40) = \begin{bmatrix} -0.8 & 0.1705 & -3.8645 \end{bmatrix}^T$$

The decision boundary is shown in Fig. 2.3a.

The MATLAB function the PCA for Example 2.1 is pocket_semi_circle.m which is enclosed at the end of this chapter.



(a) (b)

Figure 2.3 PCA applied to the double semi-circle data set (a) decision boundary obtained after 40 iterations, (b) in-sample error pattern during the first 40 PCA iterations.

2.2 Linear Models for Regression

A. *Linear Regression*

According to investopedia, "regression is a statistical measure that attempts to determine the strength of the relationship between one dependent variable (usually denoted by Y) and a series of other changing variables (known as independent variables)". In the context of machine learning, a regression model relates to a family hypotheses where each hypothesis is a function that depends on several input features $\{x_i, i = 1, \ldots, d\}$ and assumes *real-value* outputs. As such, a hypothesis in a regression model is a function that maps input $\boldsymbol{x}$ to real-valued y that are not necessarily integers. It follows that a *linear* regression simply refers to a *linear* function of the form

$$h(\boldsymbol{x}) = \sum_{i=0}^{d} w_i x_i = \boldsymbol{w}^T \boldsymbol{x} \tag{2.5}$$

where $\boldsymbol{x} = [1 \quad x_1 \quad \cdots \quad x_d]^T \in R^{d+1}$. In a linear regression problem, one is given training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,\ldots,N\}$ where y_n are real numbers and seeks to find an appropriate weight $\boldsymbol{w}^*$ for $h(\boldsymbol{x})$ in (2.5) so that $h(\boldsymbol{x})$ well predicts data outside $\mathcal{D}$.

For regression models a natural choice of the error measure, also known as *loss function*, is the so-called L_2-*loss* given by

最优化

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}\left(h(\boldsymbol{x}_n) - y_n\right)^2 \tag{2.6}$$

An error measure for out-of-sample error that is consisting with (2.6) is given by

$$E_{\text{out}} = E\left[\left(h(\boldsymbol{x}) - y\right)^2\right] \tag{2.7}$$

where the expectation is taken with respect to the joint probability distribution $P(\boldsymbol{x}, y)$ which is obviously unknown in a practical learning problem. A realistic way to evaluate E_{out} is to use an approximate measure such as

$$\hat{E}_{\text{out}}(h) = \frac{1}{M}\sum_{m=1}^{M}\left(h(\boldsymbol{x}_m) - y_m\right)^2 \tag{2.8}$$

that is evaluated over *testing* data set $\mathcal{T} = \{(\boldsymbol{x}_m, y_m), m = 1,2,\ldots,M\}$ which is drawn independently randomly outside $\mathcal{D}$.

B. *A Closed-Form Solution*

Since the hypothesis $h(\boldsymbol{x})$ (see (2.5)) depends on weight $\boldsymbol{w}$, the in-sample error in (2.6) is a function of $\boldsymbol{w}$. Combining (2.6) with (2.5), we can write

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}\left(h(\boldsymbol{x}_n) - y_n\right)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{x}_n - y_n\right)^2 = \frac{1}{N}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 \tag{2.9}$$

where

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

The expression of E_{in} in (2.9) shows that it is a convex quadratic function of $\boldsymbol{w}$. Consequently, the $\boldsymbol{w}^*$ that minimizes E_{in} is the one that nullifies the gradient of E_{in}. Computing the gradient and set it to zero, we obtain

$$\nabla_{\boldsymbol{w}}\left(E_{\text{in}}(h)\right) = \frac{2}{N}\left(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y}\right) = \boldsymbol{0}$$

which yields a solution

$$\boldsymbol{w}^* = \underbrace{\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T}_{\boldsymbol{X}^\dagger}\boldsymbol{y} \triangleq \boldsymbol{X}^\dagger\boldsymbol{y} \tag{2.10}$$

where $\boldsymbol{X}^\dagger = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T$ is the pseudo-inverse of $\boldsymbol{X}$. With the optimal weight given by (2.10), the linear regression function maps the training data $\boldsymbol{X}$ to

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}^* = \underbrace{\boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T}_{\boldsymbol{H}}\boldsymbol{y} \triangleq \boldsymbol{H}\boldsymbol{y}$$

where $\boldsymbol{H} = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T$ is called "hat matrix" [8]. It follows that the above prediction differs from the training data $\boldsymbol{y}$ by $\boldsymbol{y} - \hat{\boldsymbol{y}} = (\boldsymbol{I} - \boldsymbol{H})\boldsymbol{y}$, hence the minimized in-sample error becomes

$$E_{\text{in}} = \frac{1}{N}\|(\boldsymbol{I} - \boldsymbol{H})\boldsymbol{y}\|^2 \tag{2.11}$$

We remark that linear regression also applies to binary classification problems. In such cases the training data assumes the form $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,\ldots,N\}$ where $y_n \in \{1, -1\}$.

Example 2.2

Apply linear regression to the classification problem in Example 2.1.

Solution

For the problem in Example 2.1, we have $d = 2$ and $N = 2000$. In this case $\boldsymbol{X}$ is a matrix of size 2000 by 3, hence the matrix $\boldsymbol{X}^T\boldsymbol{X}$ that needs to be inverted is a 3 by 3 positive definite matrix. Applying the closed formula in (2.10) yields

$$\boldsymbol{w}^* = \begin{bmatrix} -0.0598 & 0.0162 & -0.0938 \end{bmatrix}^T$$

As is shown in Fig. 2.4, the decision boundary determined by $\boldsymbol{w}^*$ is a reasonable one, but not as good as those obtained by PLA and PCA. As a matter of fact, the in-sample error measured by

the percentage of misclassified pairs was found to be 0.0415. Out of $N = 2000$ sample points, 1917 points were classified correctly. For comparison we recall that the PLA yields a considerably lower in-sample error as 0.0190.

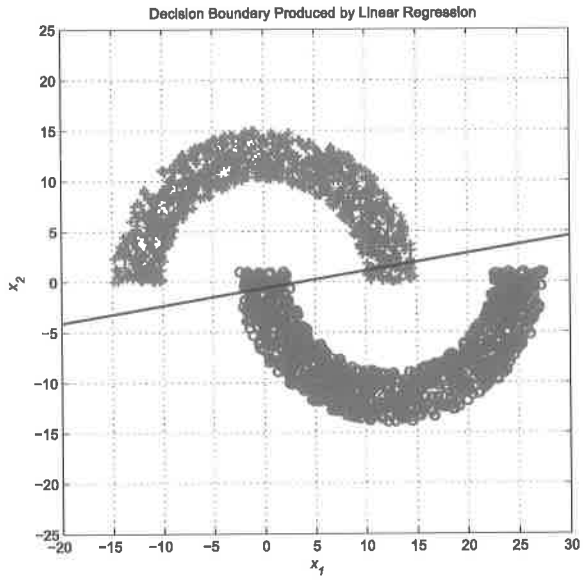


Figure 2.4 Decision boundary obtained by linear regression.

The MATLAB function that implements the linear regression algorithm for Example 2.2 is regression_semi_circle.m which is enclosed at the end of this chapter.

C. *Computing w^* by Numerical Optimization*

Formula (2.10) is compact and convenient to use, and there are stable numerical methods to compute $X^\dagger$ [12]. Nevertheless, the complexity of (2.10) grows rapidly as the dimension d of the input space and/or the size N of the training data increases.

An alternative approach to computing w^* (or an approximate of it) is to use numerical optimization methods that run inexpensive iterations to reach the solution. Candidate methods for this purpose include the steepest descent (also called gradient descent) method (SDM) where the input features are adequately scaled, and quasi-Newton algorithms such as DFP and BFGS algorithms. The key quantity required in these methods is the gradient of the loss function, which is given by

$$\nabla_w \left(E_{\text{in}}(h)\right) = \frac{2}{N} X^T \left(Xw - y\right) \tag{2.12}$$

The algorithmic details of the SDM and BFGS algorithms can be found in Sec. A.2 of Appendix A. It is worth noting that because the loss function is convex these gradient-based algorithms are guaranteed to converge to the global minimizer regardless of where the algorithm is initiated.

2.3 Logistic Regression

A. *Logistic Regression for Predicting a Probability*

Mathematically, a logistic regression function falls in between the binary-decision hypothesis

$h(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x})$ and continuous and unconstrained linear regression hypothesis $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$: like the linear classification and linear regression functions, a logistic regression function combines the components of the feature vector $\boldsymbol{x}$ linearly via a weight vector $\boldsymbol{w}$ as $\boldsymbol{w}^T\boldsymbol{x}$, on the other hand, unlike these functions, the logistic regression function produces an output between 0 and 1. Physically and more importantly, because its output is constrained in [0, 1], logistic regression fits well into problems where one seeks to predict a *probability*. Specifically, in logistic regression one is given an i.i.d. training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1,2,...,N\}$ where $y_n \in \{1, -1\}$ representing an event of interest (heart attack for example) occurs ($y = 1$) or does not occur ($y = -1$) and the goal is to identify a logistic regression function $h(\boldsymbol{x})$ that can be used to predict how probable a heart attack will occur to a patient whose features are given in terms of input vector $\boldsymbol{x}$. The reason behind the name "logistic regression" is the use of the logistic function $\theta(z)$ to map the linear combination $z = \boldsymbol{w}^T\boldsymbol{x}$ to a value between 0 and 1:

$$h(\boldsymbol{x}) = \theta(\boldsymbol{w}^T\boldsymbol{x}) \tag{2.13a}$$

where

$$\theta(z) = \frac{e^z}{1+e^z} \tag{2.13b}$$

is called *logistic function* in literature, see Fig. 2.4 for an illustration $\theta(z)$ which smoothly and monotonically increases from 0 to 1 with $\theta(0) = 0.5$.

By combining (2.13a) with (2.13b), we obtain the logistic regression hypothesis

$$h(\boldsymbol{x}) = \frac{e^{\boldsymbol{w}^T\boldsymbol{x}}}{1+e^{\boldsymbol{w}^T\boldsymbol{x}}} \tag{2.14}$$

For problems where the objective is to predict a probability rather than a deterministic answer, the target is not a function but a conditional probability distribution $P(y \mid \boldsymbol{x})$. In this situation, logistic regression can be employed to define a likelihood-function based error measure (see Sec. 1.5) as follows. First, one may relate $P(y \mid \boldsymbol{x})$ to the logistic regression function in (2.14) as

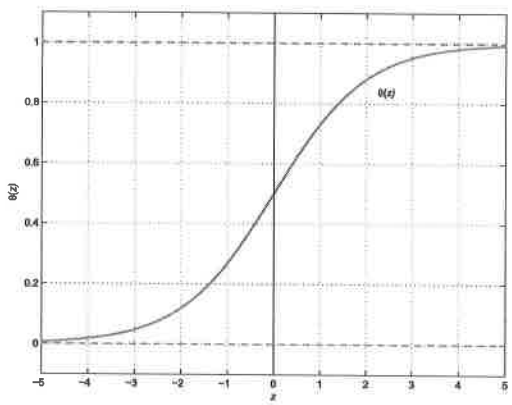


Figure 2.4 the logistic function $\theta(z) = e^z/(1+e^z)$.

$$P(y \mid \boldsymbol{x}) = \begin{cases} h(\boldsymbol{x}) & \text{for } y = 1 \\ 1 - h(\boldsymbol{x}) & \text{for } y = -1 \end{cases} \tag{2.15}$$

Using (2.13b), (2.14) and (2.15), it can readily be verified that (2.15) can be written compactly as

$$P(y \mid \boldsymbol{x}) = \theta(y \cdot \boldsymbol{w}^T \boldsymbol{x}) \tag{2.16}$$

Next, to find a weight $\boldsymbol{w}^*$ for $h(\boldsymbol{x}, \boldsymbol{w}^*)$ to best fit an i.i.d. data set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$, we try to maximize the conditional probability of getting all the y_n's in the data set given the corresponding $\boldsymbol{x}_n$'s where the conditional probability is modeled by the logistic model in (2.16). Because the data are i.i.d., we have

$$P(y_1, \cdots, y_N \mid \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N) = \prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n) = \prod_{n=1}^{N} \theta(y_n \boldsymbol{w}^T \boldsymbol{x}_n)$$

whose maximization is equivalent to minimizing its scaled negative logarithm, namely

$$-\frac{1}{N} \ln\left(\prod_{n=1}^{N} \theta(y_n \boldsymbol{w}^T \boldsymbol{x}_n)\right) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(\frac{1}{\theta(y_n \boldsymbol{w}^T \boldsymbol{x}_n)}\right) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}\right)$$

The analysis above suggests the logistic loss function as the in-sample error measure:

$$E_{\text{in}}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} \ln\left(1 + e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}\right) \tag{2.17}$$

B. *Finding $\boldsymbol{w}^*$ by Numerical Optimization*

The gradient and Hessian of $E_{\text{in}}(\boldsymbol{w})$ are given by

$$\nabla\left(E_{\text{in}}(\boldsymbol{w})\right) = -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \boldsymbol{x}_n}{1 + e^{y_n \boldsymbol{w}^T \boldsymbol{x}_n}} \tag{2.18}$$

and

$$\nabla^2\left(E_{\text{in}}(\boldsymbol{w})\right) = \frac{1}{N} \sum_{n=1}^{N} \frac{\boldsymbol{x}_n \boldsymbol{x}_n^T}{(1 + e^{y_n \boldsymbol{w}^T \boldsymbol{x}_n})^2} \tag{2.19}$$

respectively. Because the Hessian is unconditionally positive semidefinite, $E_{\text{in}}(\boldsymbol{w})$ is globally convex. Therefore, gradient-based optimization methods equipped with an adequate line search step (e.g. the back-tracking line search technique, see Sec. A.2 of Appendix A) shall converge to the global solution regardless of the initial point used. The SDM with adequately scaled input features and BFGS algorithm are good candidate optimization methods for minimizing $E_{\text{in}}(\boldsymbol{w})$ in (2.19). The Newton method might also be a good choice for problems where both d and N are moderate.

牛顿展开

Like the linear regression, we remark that logistic regression also applies to binary classification problems. In such cases, the probability produced by logistic regression can be used to label an input feature vector with $y = 1$ if the probability exceeds 0.5, which from (2.14) corresponds to $\boldsymbol{w}^T\boldsymbol{x} > 0$, and with $y = -1$ which corresponds to $\boldsymbol{w}^T\boldsymbol{x} < 0$ otherwise. It follows that when applying to binary classification problems logistic regression offers a linear decision boundary, namely $\boldsymbol{w}^T\boldsymbol{x} = 0$, see Example 2.3 below.

It is also interesting to note that logistic regression can also be applied to multi-class classification problem. As described in [13], a k-class classification problem can be solved as follows. Given an $\boldsymbol{x}$ in the input space and we want to classify it into one of a total of k classes.

First, we solve k *binary* classification problems, each singles out one (say the ith) of the k classes as "the class" and the union of the rest classes as "the other class". Denote the optimal weight obtained by $\boldsymbol{w}_{(i)}^{*}$. Next, with the given input $\boldsymbol{x}$ we compare the values $\{\boldsymbol{w}_{(i)}^{*T}\boldsymbol{x}, i=1,\ldots,k\}$ and identify the index i^{*} corresponding the maximum value $\boldsymbol{w}_{(i^{*})}^{*T}\boldsymbol{x}$ as the class to which $\boldsymbol{x}$ belongs. Note that in principle we are supposed to compute and compare the values of the logistic regression functions obtained as probability estimates. Since the logistic regression is monotonic, It suffices to evaluate and compare the values of $\{\boldsymbol{w}_{(i)}^{*T}\boldsymbol{x}, i=1,\ldots,k\}$.

Example 2.3

Apply logistic regression to the classification problem in Example 2.1.

Solution

The BFGS with backtracking (BT) line search was chosen to minimize the loss function in (2.17). With initial point $\boldsymbol{w}(0) = \boldsymbol{0}$ and it took 20 BFGS iterations to converge to the optimal weight

$$\boldsymbol{w}^{*} = \begin{bmatrix} 0.8078 & 0.0813 & -2.8802 \end{bmatrix}^{T}$$

which can be used to classify data $\boldsymbol{x}$ to one of two classes as

$$\hat{y} = \begin{cases} 1 & \text{if } h(\boldsymbol{x}) = \dfrac{e^{\boldsymbol{w}^{*T}\boldsymbol{x}}}{1+e^{\boldsymbol{w}^{*T}\boldsymbol{x}}} > \dfrac{1}{2} \\ -1 & \text{if } h(\boldsymbol{x}) = \dfrac{e^{\boldsymbol{w}^{*T}\boldsymbol{x}}}{1+e^{\boldsymbol{w}^{*T}\boldsymbol{x}}} < \dfrac{1}{2} \end{cases}$$

where function $h(\boldsymbol{x})$ is from (2.14) and is interpreted as predicting probability with 0.5 as the threshold. Out of N = 2000 sample points, 1961 points were classified correctly.The decision boundary is therefore determined by the equation

$$h(\boldsymbol{x}) = \frac{e^{\boldsymbol{w}^{*T}\boldsymbol{x}}}{1+e^{\boldsymbol{w}^{*T}\boldsymbol{x}}} = \frac{1}{2} \tag{2.20}$$

Evidently, an $\boldsymbol{x}$ satisfies (2.20) if and only if

$$\boldsymbol{w}^{*T}\boldsymbol{x} = 0 \tag{2.21}$$

Hence, like the PLA, PCA, and linear regression, the decision boundary of logistic-regression based classification is also a straight line. With the optimal weight $\boldsymbol{w}^{*}$, the decision boundary is depicted in Fig. 2.5 as a black line. For comparison purposes, the final in-sample error in terms of the percentage of misclassified data was evaluated and found to be 0.0195 (i.e. 1.95%), a performance very close to that of the PLA and considerably better than that of linear regression.

The MATLAB code that implements the logistic regression with BFGS-BT as applied to the double semi-circle data and produces Fig. 2.5 is logistic_BFGS_K.m which is enclosed at the end of this chapter.

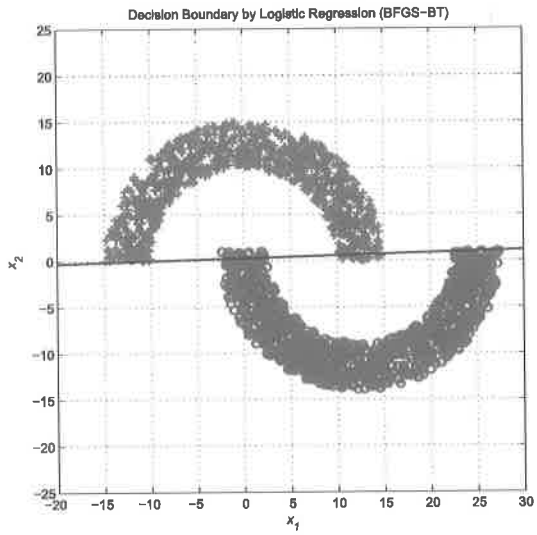


Figure 2.5 Decision boundary by logistic regression (with BFGS-BT).

For comparisons the SDM with BT line search was also used for minimizing the error measure in (2.17). As expected, SDM was slow as it took 200 iterations to converge (although the SDM iterations are slightly less complex than BFGS iterations). The algorithm yields an optimal weight

$$\boldsymbol{w}^* = \begin{bmatrix} 0.2997 & 0.0860 & -2.0634 \end{bmatrix}^T$$

which defines a decision boundary $\boldsymbol{w}^{*T}\boldsymbol{x} = 0$ shown in Fig. 2.6. The final in-sample error in terms of the percentage of misclassified data was found to be 0.0195 (i.e. 1.95%), the same performance as that of the logistic regression with BFGS-BT.

The MATLAB function that implements the logistic regression with SDM-BT as applied to the double semi-circle data and produces Fig. 2.6 is logistic_SDM_K.m which is enclosed at the end of this chapter.

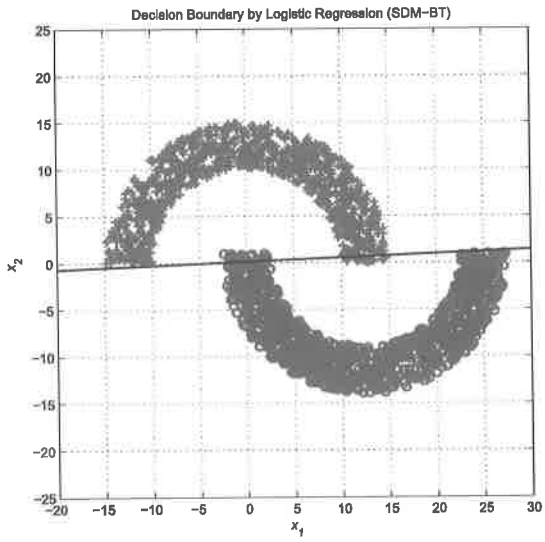


Figure 2.6 Decision boundary by logistic regression (with SDM-BT).

C. *Minimizing $E_{in}(w)$ Using Stochastic Gradient Descent (SGD)*

When the size of training data N is large, computing the gradient $\nabla(E_{in}(\boldsymbol{w}))$ as seen in (3.18) is expensive. If we write the gradient as

$$\nabla(E_{in}(\boldsymbol{w})) = \frac{1}{N}\sum_{n=1}^{N}\nabla\left(\ln\left(1+e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}\right)\right) = \frac{1}{N}\sum_{n=1}^{N}\frac{-y_n \boldsymbol{x}_n}{1+e^{y_n \boldsymbol{w}^T \boldsymbol{x}_n}} \tag{2.22}$$

we see that it is the arithmetic average of the gradients of individual error terms (at individual data points). The method of *stochastic gradient decent* (SGD) picks a training data point $(\boldsymbol{x}_n, y_n)$ uniformly *at random* and computes the gradient of the error term at $(\boldsymbol{x}_n, y_n)$, and use it to replace $\nabla(E_{in}(\boldsymbol{w}))$ in the SDM. From (2.22), the gradient of the nth error term, denoted by $e_n(\boldsymbol{w})$, is given by

$$\nabla e_n(\boldsymbol{w}) = \frac{-y_n \boldsymbol{x}_n}{1+e^{y_n \boldsymbol{w}^T \boldsymbol{x}_n}}$$

Obviously, the complexity of computing $\nabla e_n(\boldsymbol{w})$ is N times less than that of computing $\nabla(E_{in}(\boldsymbol{w}))$, a big savings when data size N is large. Further complexity reduction is achieved by using a constant step size $\alpha > 0$ when the SGD updates weight $\boldsymbol{w}(k)$ to $\boldsymbol{w}(k+1)$:

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \frac{\alpha y_n \boldsymbol{x}_n}{1+e^{y_n \boldsymbol{w}(k)^T \boldsymbol{x}_n}} \tag{2.23}$$

The success of SGD has to do with independently randomly picking an individual data point $(\boldsymbol{x}_n, y_n)$ in each iteration. There are a great deal of research for convergence analysis, performance evaluation, and development of improved SGD algorithms, see for example [14], [15] and the references therein.

Example 2.4

The SGD was applied to minimize the loss function in (2.17). With initial point $\boldsymbol{w}(0) = \boldsymbol{0}$ and an fixed $\alpha = 0.005$, it took 12,000 iterations to converge to an optimal weight

$$\boldsymbol{w}^* = [0.0916 \quad 0.0877 \quad -1.7096]^T$$

which defines a decision boundary $\boldsymbol{w}^{*T}\boldsymbol{x} = 0$ shown in Fig. 2.7a. The final in-sample error in terms of the percentage of misclassified data was found to be 0.0210 (i.e. 2.1%), a decent performance on comparing with other methods tested.

The performance of SGD may improve by using a more adequate initial weight. For instance, with a randomly generated

$$\boldsymbol{w}(0) = [1.030112096647569 \quad 0.703751395701793 \quad -1.646151244456372]^T$$

and a fixed $\alpha = 0.005$, after 12,000 iterations the SGD yields a weight vector

$$\boldsymbol{w}^* = [0.4966 \quad 0.0964 \quad -2.3410]^T$$

The associated decision boundary and a profile of in-sample error versus SGD iterations are shown in Fig, 2.8a and 2.8b, respectively. The final in-sample error in terms of the percentage of misclassified data was found to be 0.0190 (i.e. 1.9%), one of the best performance among the methods tested.

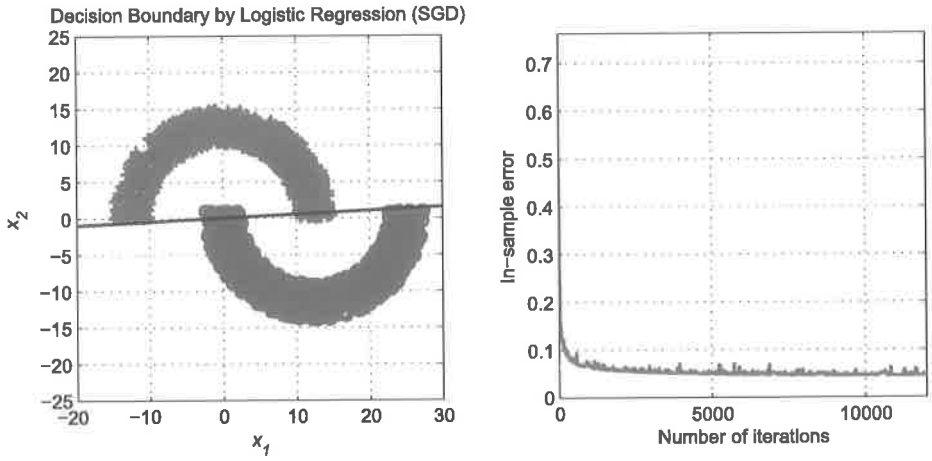


Figure 2.7 (a) Decision boundary by logistic regression (with SGD and $\boldsymbol{w}(0) = \mathbf{0}$) and (b) profile of in-sample error versus iterations.

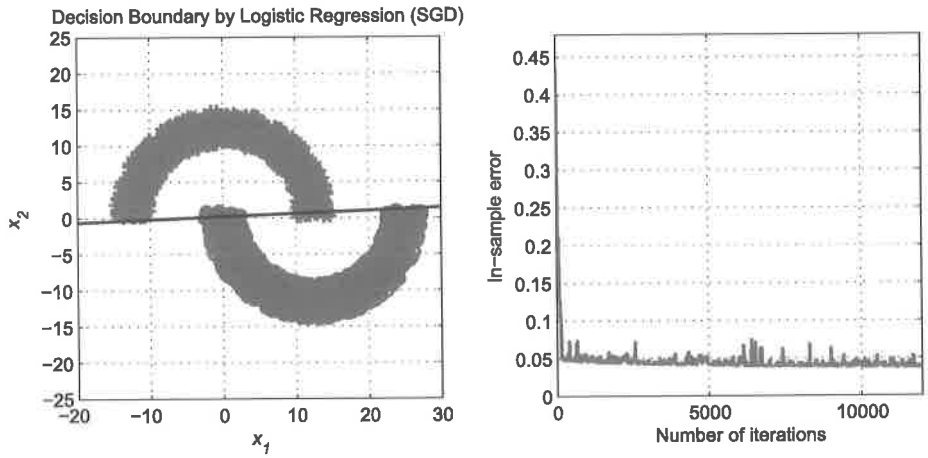


Figure 2.8 (a) Decision boundary by logistic regression (with SGD and $\boldsymbol{w}(0) = \text{randn}(3,1)$) and (b) profile of in-sample error versus iterations.

The MATLAB function that implements the logistic regression with SGD as applied to the double semi-circle data and produces Figs. 2.7 and 2.8 is logistic_SGD_K.m which is enclosed at the end of this chapter.

2.4 Nonlinear Transforms

A common thread among the linear models studied so far in this chapter is the use of the inner product $\boldsymbol{w}^T\boldsymbol{x}$ where both the hypothesis parameter $\boldsymbol{w}$ and feature vector $\boldsymbol{x}$ appear linearly. However the roles the vectors $\boldsymbol{w}$ and $\boldsymbol{x}$ play in a learning algorithm are different: a learning model is considered linear if the weight $\boldsymbol{w}$ *linearly* combines available features. On the other hand, the features from training data are known and treated as *constants*. Consequently, as long as the inner product assumes the form $\boldsymbol{w}^T$(combination of features), a learning model remains

linear regardless of how the input features are combined and manipulated because they are merely numerical constants determined by the training data. In this section, we examine the effect of using some *nonlinear* combinations of the individual features x_i's on the performance of several linear models studied earlier, especially the ability in dealing with non-separable data sets. For clarity, we follow [8] to denote a transform that maps a feature vector $\boldsymbol{x} \in R^d$ to a nonlinearly dilated feature vector $\boldsymbol{z} \in R^{\tilde{d}}$ (this Euclidean space is called Z space) by

$$\boldsymbol{z} = \Phi(\boldsymbol{x}) \tag{2.24}$$

and the weight associated with $\tilde{d}$ - dimensional Z space by $\tilde{\boldsymbol{w}} \in R^{\tilde{d}}$. Natural choices of Z space include polynomial type of dilations. For a 2-dimensional input space R^2 for example, the general 2nd-order and 3rd-order dilations assume the form

$$\Phi_2(\boldsymbol{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \end{bmatrix}^T \tag{2.25}$$

and

$$\Phi_3(\boldsymbol{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & x_1^3 & x_1^2x_2 & x_1x_2^2 & x_2^3 \end{bmatrix}^T \tag{2.26}$$

respectively. Note that as the order of nonlinearity increases, the dimension of the dilated feature space grows very quickly, this implies rapid increase of the VC dimension.

In principle, the methods addressed in this chapter so far can all be extended to include nonlinear transforms to work in dilated input space Z, and the inner product $\boldsymbol{w}^T\boldsymbol{x}$ involved in the models as well as in the equations that define decision boundaries in the case of classification problems is replaced by $\tilde{\boldsymbol{w}}^T\boldsymbol{z}$ where $\boldsymbol{z}$ is related to $\boldsymbol{x}$ via $\boldsymbol{z} = \Phi(\boldsymbol{x})$. In this way, the linear decision boundary $\boldsymbol{w}^T\boldsymbol{x} = 0$ is replaced by $\tilde{\boldsymbol{w}}^T\boldsymbol{z} = 0$ which via (2.24) becomes

$$\tilde{\boldsymbol{w}}^T\Phi(\boldsymbol{x}) = 0 \tag{2.27}$$

Using polynomial type of nonlinear transforms such as those in (2.25) and (2.26), the decision boundary in (2.27) is nonlinear and has better chances to separate the data that cannot be done with linear decision boundaries.

Example 2.5

Develop linear regression that works with a dilated input space Z with $\Phi(\boldsymbol{x}) = \Phi_3(\boldsymbol{x})$ and apply it to the classification problem in Example 2.1.

Solution

The regression function is this case is given by

$$h(\boldsymbol{z}) = h\left(\Phi_3(\boldsymbol{x})\right) = \sum_{i=0}^{9} \tilde{w}_i z_i = \tilde{\boldsymbol{w}}^T\boldsymbol{z} \tag{2.28}$$

where $\boldsymbol{z}$ is a 10-dimensional vector with $z_0 = 1, z_1 = x_1, z_2 = x_2, z_3 = x_1^2, z_4 = x_1x_2, z_5 = x_2^2, z_6 = x_1^3,$ $z_7 = x_1^2x_2, z_8 = x_1x_2^2, z_9 = x_2^3$. The loss function is given by

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}\left(h\left(\Phi_3(\boldsymbol{x}_n)\right) - y_n\right)^2$$

which in conjunction with (2.28) yields

$$E_{\text{in}}(h)=\frac{1}{N}\sum_{n=1}^{N}\left(h\left(\Phi_3(\boldsymbol{x}_n)\right)-y_n\right)^2=\frac{1}{N}\sum_{n=1}^{N}\left(\tilde{\boldsymbol{w}}^T\boldsymbol{z}_n-y_n\right)^2=\frac{1}{N}\left\|\boldsymbol{Z}\tilde{\boldsymbol{w}}-\boldsymbol{y}\right\|^2 \tag{2.29}$$

$$\boldsymbol{Z}=\begin{bmatrix}\boldsymbol{z}_1^T\\ \boldsymbol{z}_2^T\\ \vdots\\ \boldsymbol{z}_N^T\end{bmatrix},\quad \boldsymbol{y}=\begin{bmatrix}y_1\\ y_2\\ \vdots\\ y_N\end{bmatrix}$$

We see that $E_{\text{in}}(h)$ remains to be a convex quadratic function of $\tilde{\boldsymbol{w}}$ whose global minimum is achieved at

$$\tilde{\boldsymbol{w}}^*=\left(\boldsymbol{Z}^T\boldsymbol{Z}\right)^{-1}\boldsymbol{Z}^T\boldsymbol{y}\triangleq\boldsymbol{Z}^\dagger\boldsymbol{y} \tag{2.30}$$

Applying (2.30) to the data from Example 3.1, we obtain the numerical values of 10 components of $\tilde{\boldsymbol{w}}^*$ as

0.412665257054635
-0.030808874414424
-0.167834080860251
-0.006375722902075
-0.007360582606473
-0.005499749941547
0.000334340108927
0.000542429175521
0.000755834680100
0.000782595233551

As can be seen from Fig. 2.9, the decision boundary defined by $\tilde{\boldsymbol{w}}^{*T}\Phi_3(\boldsymbol{x})=0$ separates to two subsets of the data successfully, which simply means zero in-sample error in terms of the percentage of misclassified data.

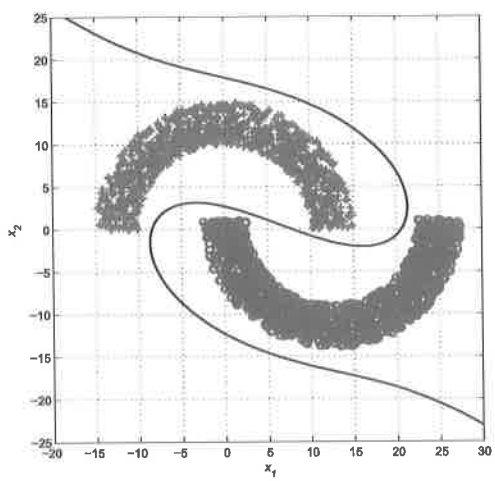


Figure 2.9 Decision boundary by linear regression with 3rd-order nonlinear transform.

The MATLAB function that implements linear regression with 3rd nonlinear transform as applied to the double semi-circle data and produces Fig. 2.9 is regression_NL, and is enclosed at the end of this chapter.

The next example presents simulation results concerning the logistic regression with nonlinear transform for classification.

Example 2.6

Apply logistic regression with 3rd-order nonlinear transform to the classification problem in Example 2.1.

Solution

The problem was addressed using two optimization techniques, namely SDM and BFGS.

(a) ***Using SDM-BT***

With a10-dimensional feature space Z the objective function for logistic regression assumes the form

$$E_{\text{in}}(\tilde{w}) = \frac{1}{N}\sum_{n=1}^{N}\ln\left(1+e^{-y_n\tilde{w}^T z_n}\right) \tag{2.31}$$

whose gradient is given by

$$\nabla\left(E_{\text{in}}(\tilde{w})\right) = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n z_n}{1+e^{y_n\tilde{w}^T z_n}} \tag{2.32}$$

The above expressions are almost the same as those in (2.18) and (2.19) merely replacing $\boldsymbol{w}$ and $\boldsymbol{x}_n$ with $\tilde{w}$ and z_n, respectively. Thus the changes in MATLAB code needed for implementing the algorithm are fairly minor. A significant difference between the two sets of formulas is the increase of the variables from 3 to 10. Associated with that, the data set we deal with here, namely $\{(z_n, y_n)$, for $n = 1, \ldots, 2000\}$, moves to the same high dimensional space and exhibits increasing numerical irregularity (see (2.28)). All these contribute to further slowness of the SDM. With $\boldsymbol{w}(0)$ = zeros(10,1), it took 10^5 iterations for SDM to converge to a weight vector $\boldsymbol{w}^*$ whose 10 components are shown below.

0.287065071143320
-0.044616276316051
-0.087377965695571
-0.059737864637012
-0.053026567234247
0.236067386235712
0.003612581213920
-0.016650016709875
0.071229795734063
-0.278067034535488

The associated decision boundary is shown in Fig, 2.10. The final in-sample error in terms of the percentage of misclassified data was found to be 0.003 (i.e. 0.3%), that outperforms all linear

methods without nonlinear transform, but not as good as that of the linear regression with 3rd-order nonlinear transform as demonstrated in Example 2.5.

Evidently, improved performance can be obtained by executing more SDM iterations. As a matter of fact, running 2×10^5 SDM iterations yields a $\boldsymbol{w}^*$ that reduces the in-sample error (in terms of the percentage of misclassified data) to zero. The components of the optimal weight are given by

0.748567483227266
-0.105175128517390
-0.131573469558948
-0.063641168135711
-0.097885507589359
0.472612616904333
0.003865924849607
-0.015072214074362
0.091181072818658
-0.368710258973415

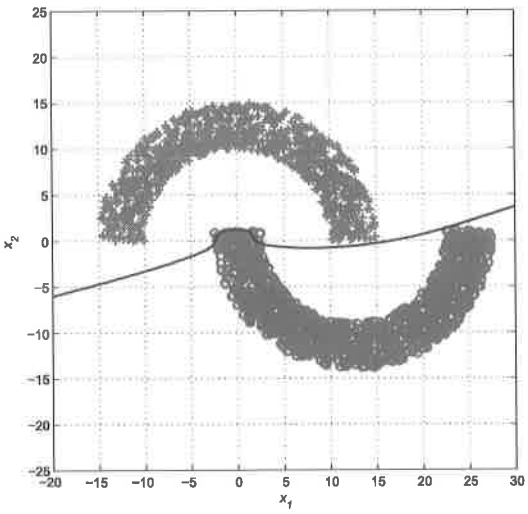


Figure 2.10 Decision boundary by logistic regression with 3rd-order nonlinear transform after 10^5 SDM iterations.

The associated decision boundary which successfully separates the two data subsets is shown in Fig, 2.11.

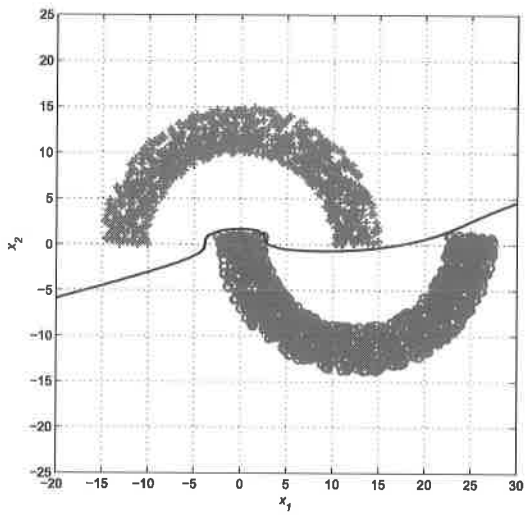


Figure 2.11 Decision boundary by logistic regression with 3rd-order nonlinear transform after 2×10^5 SDM iterations.

The MATLAB function that implements the logistic regression with 3rd-order nonlinear transform and SDM-BT as applied to the double semi-circle data is logistic_SDM_NL.m which is enclosed at the end of this chapter.

(b) ***Using BFGS-BT***

Logistic regression with 3rd-order nonlinear transform was also applied to the classification problem in Example 2.1, where the optimization was carried out by BFGS-BT. With $\boldsymbol{w}(0)$ = zeros(10,1), it took 34 iterations for BFGS to converge to a weight vector $\boldsymbol{w}^*$ whose 10 components are shown below.

5.208531372296072
-0.603211687574635
-2.614896708089146
-0.164307452838640
0.255000978291617
0.732072511422932
0.008795857137318
-0.018771408101358
-0.064320065099328
-0.073969868145694

The in-sample error in terms of percentage of misclassified data was zero and the associated decision boundary which successfully separates the two data subsets is shown in Fig, 2.12.

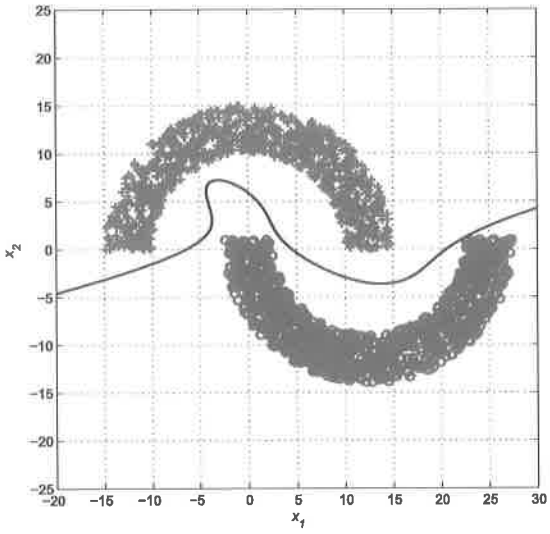


Figure 2.12 Decision boundary by logistic regression with 3rd-order nonlinear transform after 34 BFGS iterations.

2.5 Dimension Reduction

Many machine learning problems involve high dimensional data. In the case of classifying handwritten digits from MNIST (Mixed National Institute of Standards and Technology) database, for example, each digit is an image of size 28 by 28 (Fig. 2.13 shows 54 digits from the database), therefore the data are in 784-dimensional space. On the other hand, images of handwritten digits are known to live in low-dimensional manifolds, therefore low-dimensional representation of high-dimensional data of interest is possible in this case. This section introduces several basic tools that are found useful in materializing such low-dimensional representations. The general machinery of these tools is referred to as *dimension reduction* or *dimensionality reduction*.

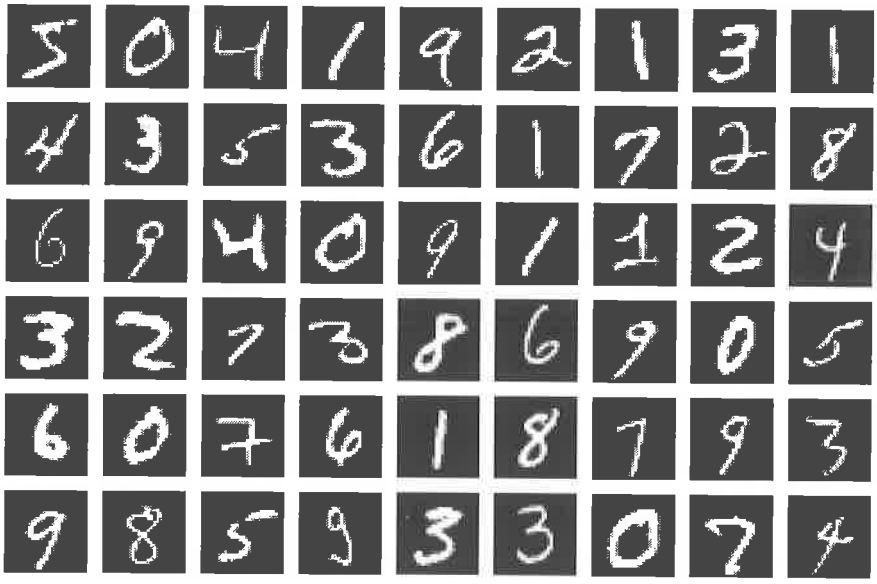


Fig. 2.13 Handwritten digits from MNIST database.

A. ***Singular Value Decomposition***

Let X be a real-valued matrix of size m by n whose rank is r, there exist orthogonal matrices $U \in R^{m\times m}$ and $V \in R^{n\times n}$ such that

$$X = U\Sigma V^T \tag{2.33}$$

where

$$\Sigma = \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix}$$

and $S = \mathrm{diag}\{\sigma_1, \sigma_2, \cdots, \sigma_r\}$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. The σ_i's are called *singular values* of X, and the columns of U and V are called left and right singular vectors of X, respectively, and (2.33) is said to be the *singular value decomposition* (SVD) of X. The concepts of singular values and singular vectors are closely related to more popular eigenvalues and eigenvectors. The connection can be seen immediately through the products

$$XX^T = U\begin{bmatrix} S^2 & 0 \\ 0 & 0 \end{bmatrix}U^T, \quad X^TX = V\begin{bmatrix} S^2 & 0 \\ 0 & 0 \end{bmatrix}V^T$$

In words, the nonzero singular values of X are the square roots of the nonzero eigenvalues of XX^T and X^TX; the left singular vectors are the eigenvectors of XX^T; and the right singular vectors are the eigenvectors of X^TX. The MATLAB function svd implements (2.33) as [U,S,V] = svd(X).

Following (2.33), the SVD of X can be written as

$$X = U_r S V_r^T \tag{3.34}$$

where U_r and V_r are the matrices composed of the first r columns of U and V, respectively.

Among other things, SVD is useful in finding optimal low-rank approximations of X. In effect, given a positive integer $r' < r$ where r is the rank of X, an optimal rank- r' approximation of X, denoted by X', can be constructed via (2.34) as

$$X_{r'} = U_{r'} S_{r'} V_{r'}^T \tag{2.35}$$

where $U_{r'}$ and $V_{r'}$ are the matrices composed of the first r' columns of U_r and V_r, respectively, and $S_{r'} = \mathrm{diag}\{\sigma_1, \sigma_2, \cdots, \sigma_{r'}\}$. The approximation error introduced by $X_{r'}$ in Frobenius norm is controlled by the 2-norm of the remaining singular values (the residue energy), namely

$$\| X - U_{r'} S V_{r'}^T \|_F = \left(\sum_{i=r'+1}^{r} \sigma_i^2 \right)^{1/2} \tag{2.36}$$

B. Dimension Reduction of Data Matrix X Using SVD

One may think of matrix X in (2.33) as a data array that collects a set of n data points $\{x_i, i = 1, 2, \ldots, n\}$ (as column vectors) in m-dimensional Euclidean space, that is, $X = [x_1\ x_2 \cdots x_n]$ whose rank is r. Using (2.34), we define $Z_r = SV_r^T$ which is a matrix of size r by n, and write it as $Z_r = [z_1\ z_2 \cdots z_n]$ with $z_i \in R^{r\times 1}$. It can be readily verified that the set $\{z_i, i = 1, 2, \ldots, n\}$ preserves the topological structure of the original data set $\{x_i, i = 1, 2, \ldots, n\}$ while reducing the dimension from m to r in the sense that the distance and "angle" between each pair of the data points as well as the length (energy) of each individual vector (signal) are preserved, namely,

$$\| z_i \| = \| x_i \|,\ \| z_i - z_j \| = \| x_i - x_j \| \text{ and } \langle z_i, z_j \rangle = \langle x_i, x_j \rangle \text{ for } 1 \le i, j \le n \tag{2.37}$$

A simple proof of (2.37) would be based on the fact that the two data sets differ only by U_r as seen from (2.34), whose columns are orthonormal. This SVD-based dimension reduction is particularly attractive when $r << m$.

What if $r << m$ does not hold? In this scenario one examines the r nonzero singular values to see if the number of the dominating singular values, denoted by r', is small. If so, based on (2.35) and (2.36) one can claim that

$$X \approx U_{r'} S_{r'} V_{r'}^T \tag{2.38}$$

Consequently, if we define $Z_{r'} = SV_{r'}^T$ and write it as $Z_{r'} = [z_1\ z_2 \cdots z_n]$ with $z_i \in R^{r'\times 1}$, then we have

$$\| z_i \| \approx \| x_i \|,\ \| z_i - z_j \| \approx \| x_i - x_j \| \text{ and } \langle z_i, z_j \rangle \approx \langle x_i, x_j \rangle \text{ for } 1 \le i, j \le n \tag{2.39}$$

demonstrating the validity of using $Z_{r'} = SV_{r'}^T$ as a low-dimensional approximation of the original data X.

Perhaps the biggest problem with the SVD-based dimension reduction is that of computing the SVD of X, especially for large size X. A well-known solution to this problem is the Johnson-Lindenstrauss lemma [16]. Let $P \in R^{p\times m}$ be a projection matrix with $p << m$ that is generated by sampling p row vectors independently and uniformly on the unit sphere in the m-dimensional space which are in turn orthonormalized with each row scaled to have its 2-norm equal to

$\sqrt{m/p}$. The dimension reduction of data matrix $\boldsymbol{X}$ is accomplished in two steps: First, we compute $\boldsymbol{Y} = \boldsymbol{PX}$ that projects the m-dimensional vectors $\boldsymbol{x}_i$'s to p-dimensional vectors $\boldsymbol{y}_i$'s where $\boldsymbol{y}_i$ is the ith column of $\boldsymbol{Y}$. Next, the SVD-based dimension reduction is applied to data set $\boldsymbol{Y}$:

$$\boldsymbol{Y} = \hat{\boldsymbol{U}}\hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^T = \underbrace{\hat{\boldsymbol{U}}_r}_{\substack{\text{first } r \\ \text{columns} \\ \text{of } \hat{\boldsymbol{U}}}} \cdot \hat{\boldsymbol{S}}_r \cdot \underbrace{\hat{\boldsymbol{V}}_r^T}_{\substack{\text{first } r \\ \text{rows} \\ \text{of } \hat{\boldsymbol{V}}^T}}$$

from which we obtain $\hat{\boldsymbol{Z}}_r = \hat{\boldsymbol{S}}_r \cdot \hat{\boldsymbol{V}}_r^T$. It can be shown (Gilbert-Park-Wakin 2012) that if

$$p \geq \frac{r\log(42/\varepsilon) + \log(2/\delta)}{f\left(\varepsilon/\sqrt{2}\right)}$$

where $\varepsilon \in (0, 1)$, $\delta \in (0, 1)$ and $f(\cdot)$ is quadratic, then with probability $1-\delta$

$$(1-\varepsilon)^{1/2} \leq \frac{\hat{\sigma}_j}{\sigma_j} \leq (1+\varepsilon)^{1/2}$$

for $j = 1, 2, \ldots, r$, and

$$\left\|\boldsymbol{v}_j - \hat{\boldsymbol{v}}_j\right\|_2 \leq \min\left\{\sqrt{2}, \frac{\varepsilon\sqrt{1+\varepsilon}}{\sqrt{1-\varepsilon}} \max_{i \neq j} \frac{\sqrt{2}\sigma_i\sigma_j}{\min\limits_{c\in[-1,1]}\{|\sigma_i^2 - \sigma_j^2(1+c\varepsilon)|\}}\right\}$$

C. <u>*Principal Component Analysis (PCA)*</u>

Again we consider a real-valued data matrix $\boldsymbol{X} = [\boldsymbol{x}_1\ \boldsymbol{x}_2 \cdots \boldsymbol{x}_n]$ of size m by n. Define the *average vector* and *covariance matrix* of the n component vectors $\boldsymbol{x}_i$'s as

$$\bar{\boldsymbol{x}} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i, \quad \boldsymbol{C} = \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T \tag{2.40}$$

respectively. Note that $\boldsymbol{C}$ is at least positive semidefinite if not positive definite, hence its SVD is identical to its eigen-decomposition

$$\boldsymbol{C} = \boldsymbol{USU}^T \tag{2.41}$$

where $\boldsymbol{U} = [\boldsymbol{u}_1\ \boldsymbol{u}_2 \cdots \boldsymbol{u}_m]$ is an orthogonal matrix and $\boldsymbol{S} = \mathrm{diag}\{\sigma_1, \sigma_2, \cdots, \sigma_m\}$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m \geq 0$, and $\boldsymbol{u}_i$ is $\boldsymbol{C}$'s ith eigenvector associated with ith eigenvalue σ_i. By taking into account only the K largest eigenvalues and the associated eigenvectors, we obtain a useful approximation of variance matrix $\boldsymbol{C}$ as

$$\boldsymbol{C} \approx \boldsymbol{U}_K\boldsymbol{S}_K\boldsymbol{U}_K^T \text{ with } \boldsymbol{U}_K = [\boldsymbol{u}_1\ \boldsymbol{u}_2 \cdots \boldsymbol{u}_K] \text{ and } \boldsymbol{S}_K = \mathrm{diag}\{\sigma_1, \sigma_2, \cdots, \sigma_K\} \tag{2.42}$$

The expression in (2.42) forms the basis of a data analysis tool known as principal component

analysis which finds numerous applications [17]. For the sake of machine learning the usefulness of (2.42) may be understood from two different but related perspectives as follows.

(1) Since $\boldsymbol{C}$ is the variance for $\{\boldsymbol{x}_i - \bar{\boldsymbol{x}}, i = 1,2,...,n\}$, its K eigenvectors $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_K\}$ capture the most significant variations between the individual data points and their average and each point $\boldsymbol{x}_i - \bar{\boldsymbol{x}}$ is well represented by its projection onto the K-dimensional subspace spanned by $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_K\}$, i.e.,

$$z_i = \boldsymbol{U}_K^T(\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \tag{2.43a}$$

In effect, point $\boldsymbol{x}_i$ is approximated by a point in that subspace (plus the average $\bar{\boldsymbol{x}}$) as

$$\boldsymbol{x}_i \approx \boldsymbol{U}_K z_i + \bar{\boldsymbol{x}} \tag{3.43b}$$

In this way, we see how the original data set $\boldsymbol{X}$ in R^m is reduced to an approximately equivalent data set $\boldsymbol{Z} = [z_1\ z_2 \cdots z_n]$ in space R^K (plus a single average vector $\bar{\boldsymbol{x}}$). This PCA-based dimensionality reduction becomes substantial when $K << m$.

(2) Now let us consider a case of supervised multi-class classification with a training set that consists of L data classes of the form $\{(\boldsymbol{x}_i^{(j)}, y_j), i = 1,2,...,n_j\}$ for j = 1, 2, …, L, and the problem is to classify a point $\boldsymbol{x}$ outside the training data to one of the L classes. To this end, PCA is applied to each data class $\{(\boldsymbol{x}_i^{(j)}, y_j), i = 1,2,...,n_j\}$ by first computing the average vector and covariance matrix as

$$\bar{\boldsymbol{x}}_j = \frac{1}{n_j}\sum_{i=1}^{n_j} \boldsymbol{x}_i^{(j)}, \quad \boldsymbol{C}_j = \frac{1}{n_j - 1}\sum_{i=1}^{n_j}(\boldsymbol{x}_i^{(j)} - \bar{\boldsymbol{x}}_j)(\boldsymbol{x}_i^{(j)} - \bar{\boldsymbol{x}}_j)^T$$

then performing eigen-decomposition of $\boldsymbol{C}_j$

$$\boldsymbol{C}_j = \boldsymbol{U}_j \boldsymbol{S}_j \boldsymbol{U}_j^T$$

and approximate it by

$$\boldsymbol{C}_j \approx \boldsymbol{U}_K^{(j)} \boldsymbol{S}_K^{(j)} \boldsymbol{U}_K^{(j)T}$$

where $\boldsymbol{U}_K^{(j)} = [\boldsymbol{u}_1^{(j)}\ \boldsymbol{u}_2^{(j)} \cdots \boldsymbol{u}_K^{(j)}]$ consists of K eigenvectors of $\boldsymbol{C}_j$ that are associated K largest eigenvalues and $\boldsymbol{S}_K^{(j)}$ is a diagonal matrix that collects the K largest eigenvalues in descent order. In a sense, the L data classes $\{\boldsymbol{x}_i^{(j)}, i = 1,2,...,n_j\}$ for j = 1, 2, …, L are well represented be L reduced "data" sets $\{\bar{\boldsymbol{x}}_j, \boldsymbol{U}_K^{(j)}\}$ for j = 1, 2, …, L, where $\bar{\boldsymbol{x}}_j$ represents the average character of $\{\boldsymbol{x}_i^{(j)}, i = 1,2,...,n_j\}$ and $\boldsymbol{U}_K^{(j)}$ defines a K-dimensional subspace that resides in R^m for satisfactory representation of variations of individual members in the jth class from its average $\bar{\boldsymbol{x}}_j$. The

dimension reduction becomes substantial when $K << m$. The classification of a testing point $\boldsymbol{x}$ is carried out as follows: (i) Project point $\boldsymbol{x}-\bar{\boldsymbol{x}}_j$ into the jth data class, i.e. compute $\boldsymbol{z}_j = \boldsymbol{U}_K^{(j)T}(\boldsymbol{x}-\bar{\boldsymbol{x}}_j)$; and approximate point $\boldsymbol{x}$ in the jth class as $\hat{\boldsymbol{x}}_j = \boldsymbol{U}_K^{(j)}\boldsymbol{z}_j + \bar{\boldsymbol{x}}_j$; (ii) Evaluate the difference between $\boldsymbol{x}$ and its approximation $\hat{\boldsymbol{x}}_j$ in 2-norm, i.e. , $e_j = \|\boldsymbol{x}-\hat{\boldsymbol{x}}_j\|_2$ for $j = 1, 2, \ldots, L$. Point $\boldsymbol{x}$ is classified to class j^* if e_{j^*} reaches the minimum among $\{e_j, j = 1, 2, \ldots, L\}$.

D. *<u>An Example: Classification of Handwritten Digits from MNIST Data Base Using PCA</u>*

The MNIST database provides 60,000 handwritten digits for training purposes and a separate 10,000 digits testing, where each digit is a black-and-white image of size 28 by 28. In this example the training images were partitioned into ten classes, with the images in the jth class representing digit j, for j = 0, 1, …, 9. We select from each class 500 images at random, then convert each image into a column vector of dimension 784 to construct a data a data matrix $\boldsymbol{X}_j$ of size 784 by 500. For convenience, these ten data matrices are combined to a signal data matrix X500 of size 784 by 5000 as a .mat file which you can download from the course website. The PCA-based classification method as described above was applied to classify the 10K testing digits. With $K = 24$ (see part C for the meaning of K), an error rate of 4.4% was achieved.

The MATLAB function that implements the simulation is class_pca_digits.m which is enclosed at the end if this chapter.

MATLAB Code Used in This Chapter

(1) data_semi_circle.m

```
% Example: [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% Written by W.-S. Lu, University of Victoria.
% Last modified: Dec. 30, 2014.
function [x,y,xp,xn] = data_semi_circle(r,thk,sep,N,st1,st2)
r1 = r;
r2 = r + thk;
ku = 0;
xn = [];
randn('state',st1)
while ku < N,
   xw = 12*randn(2,1);
   cr = 0;
   xm = norm(xw);
   if xm < r2 & xm > r1,
      cr = cr + 1;
   end
   if xw(2) > 0,
      cr = cr + 1;
   end
   if cr == 2,
      xn = [xn xw];
      ku = ku + 1;
   end
end
kd = 0;
xp = [];
randn('state',st2)
while kd < N,
   xw = 12*randn(2,1);
   cr = 0;
   xm = norm(xw);
   if xm < r2 & xm > r1,
      cr = cr + 1;
   end
   if xw(2) < 0,
      cr = cr + 1;
   end
   if cr == 2,
      xp = [xp xw];
      kd = kd + 1;
   end
end
ra = 0.5*(r1+r2);
xp(1,:) = xp(1,:) + ra;
xp(2,:) = xp(2,:) - sep;
x = [xp xn];
y = [ones(N,1); -ones(N,1)];
```

```
figure(1)
clf
plot(xp(1,:),xp(2,:),'bo')
hold on
plot(xn(1,:),xn(2,:),'r+')
grid
xlabel('\itx_1')
ylabel('\itx_2')
axis([-20 30 -25 25])
axis square
title('Training Data: Double Semi-Circle')
hold off
```

(2) pocket_semi_circle.m

```
% Pocket algorithm (PCA) as applied to training data known as semi-circle.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,t,ein_pocket] = pocket_semi_circle(x,y,xp,xn,0.1,[0 0 0]',15,26,40);
% Written by W.-S. Lu, University of Victoria. % Last modified: Jan. 25, 2015.
function [wt,t,ein_pocket] = pocket_semi_circle(x,y,xp,xn,lr,w0,st1,st2,K)
N = length(y);
rand('state',st1)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
t = 0;
w = w0(:);
w_pocket = w;
D = [ones(N,1) x'];
y = y(:);
dwt = (D*w >= 0);
acc = sum(y == 2*dwt - 1);
acc_pocket(1) = acc;
rand('state',st2)
N1 = randperm(N);
while t < K & acc_pocket(t+1) < N,
    ii = mod(t,N) + 1;
    i = N1(ii);
    fi = 2*dwt(i) - 1;
    w = w + lr*(y(i)-fi)*D(i,:)';
    dwt = (D*w >= 0);
    acc = sum(y == 2*dwt - 1);
    if acc > acc_pocket(t+1),
      w_pocket = w;
      acc_pocket(t+2) = acc;
    else
      acc_pocket(t+2) = acc_pocket(t+1);
    end
```

```
    t = t + 1;
end
disp(sprintf('Solution reached in %d iterations.', t));
disp('Final weights:')
wt = w_pocket
ent = N - acc_pocket(end);
Ein = ent/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly',N - ent));
figure(1)
subplot(121)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
grid
xlabel('\itx_1')
ylabel('\itx_2')
p1 = -20;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 30;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([-20 30 -25 25])
axis square
title('Decision Boundary Produced by PCA')
hold off
subplot(122)
ein_pocket = (N - acc_pocket)/N;
plot(1:1:t+1,ein_pocket,'b-','linewidth',1.5)
axis square
axis([1 t+1 0 1.1*ein_pocket(1)])
xlabel('number of iterations')
ylabel('in-sample error')
grid
```

(3) regression_semi_circle.m

```
% Linear regression as applied to training data known as semi-circle.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,Ein1,Ein2] = regression_semi_circle(x,y,xp,xn,5);
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 25, 2015.
function [wt,Ein1,Ein2] = regression_semi_circle(x,y,xp,xn,st)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
```

```
X = [ones(N,1) x'];
y = y(:);
wt = (inv(X'*X))*(X'*y);
disp('Final weights:')
wt
Ein1 = norm(X*wt-y)^2/N;
disp(sprintf('In-sample error was found to be %d.', Ein1));
dwt = (X*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
Ein2 = L/N;
disp(sprintf('Out of a total of %d samples,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
grid
xlabel('\itx_1')
ylabel('\itx_2')
p1 = -20;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 30;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([-20 30 -25 25])
axis square
title('Decision Boundary Produced by Linear Regression')
hold off
```

(4) logistic_BFGS_K.m

```
% To apply logistic regression to classify double nonseparable demi-circle
% data. BFGS algorithm is chosen to minimize the logistic loss function.
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 25, 2015.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,fs] = logistic_BFGS_K(x,y,xp,xn,'logistic_f','logistic_g',zeros(3,1),17,20);
function [wt,fs] = logistic_BFGS_K(x,y,xp,xn,fname,gname,w0,st,K)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
d1 = length(w0);
p = [N; d1; y; x(:)];
I = eye(d1);
k = 1;
wk = w0;
```

```
Sk = I;
gk = feval(gname,wk,p);
dk = -Sk*gk;
ak = bt_lsearch(wk,dk,fname,gname,p);
dtk = ak*dk;
wk_new = wk + dtk;
while K > k,
    gk_new = feval(gname,wk_new,p);
    gmk = gk_new - gk;
    D = dtk'*gmk;
    if D <= 0,
      Sk = I;
    else
      sg = Sk*gmk;
      sw0 = (1+(gmk'*sg)/D)/D;
      sw1 = dtk*dtk';
      sw2 = sg*dtk';
      Sk = Sk + sw0*sw1 - (sw2'+sw2)/D;
    end
    gk = gk_new;
    wk = wk_new;
    dk = -Sk*gk;
    ak = bt_lsearch(wk,dk,fname,gname,p);
    dtk = ak*dk;
    wk_new = wk + dtk;
    k = k + 1;
end
disp('Final weights:')
wt = wk_new
fs = feval(fname,wt,p);
disp(sprintf('Objective function at solution point: %d.', fs));
Dt = [ones(N,1) x'];
y = y(:);
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
grid
xlabel('\itx_1')
ylabel('\itx_2')
p1 = -20;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 30;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([-20 30 -25 25])
```

```
axis square
title('Decision Boundary by Logistic Regression (BFGS-BT)')
hold off
```

==========

```
function z = logistic_f(w,p)
N = p(1);
d = p(2) - 1;
y = p(3:N+2);
v = p((N+3):end);
x = vec2mat(v,d)';
D = [ones(1,N); x];
z = 0;
for i = 1:N,
   z = z + log(1 + exp(-y(i)*(w'*D(:,i))));
end
z = z/N;
```

==========

```
function z = logistic_g(w,p)
N = p(1);
d = p(2) - 1;
y = p(3:N+2);
v = p((N+3):end);
x = vec2mat(v,d)';
D = [ones(1,N); x];
z = 0;
for i = 1:N,
   ti = y(i)*D(:,i);
   z = z + ti/(1 + exp(w'*ti));
end
z = -z/N;
```

(5) logistic_SDM_K.m

```
% To apply logistic regression to classify double nonseparable demi-circle
% data. The SDM algorithm with BT is chosen to minimize the logistic loss function.
% Written by W.-S. Lu, University of Victoria.
% Last modified: Jan. 25, 2015.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,fk] = logistic_SDM_K(x,y,xp,xn,'logistic_f','logistic_g',zeros(3,1),17,200);
function [wt,fk] = logistic_SDM_K(x,y,xp,xn,fname,gname,w0,st,K)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
d1 = length(w0);
p = [N; d1; y; x(:)];
```

```
k = 0;
wk = w0;
fk(1) = feval(fname,wk,p);
while k < K,
    gk = feval(gname,wk,p);
    dk = -gk;
    ak = bt_lsearch(wk,dk,fname,gname,p);
    dtk = ak*dk;
    wk = wk + dtk;
    k = k + 1;
    fk(k+1) = feval(fname,wk,p);
end
disp('Final weights:')
wt = wk
fs = feval(fname,wt,p);
disp(sprintf('Objective function at solution point: %d.', fs));
Dt = [ones(N,1) x'];
y = y(:);
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
grid
xlabel('\itx_1')
ylabel('\itx_2')
p1 = -20;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 30;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([-20 30 -25 25])
axis square
title('Decision Boundary by Logistic Regression (SDM-BT)')
hold off
```

(6) logistic_SGD_K.m

```
% To apply logistic regression to classify double nonseparable demi-circle
% data. The SGD algorithm is chosen to minimize the logistic loss function.
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 25, 2015.
% Examples:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,fk] = logistic_SGD_K(x,y,xp,xn,'logistic_f',zeros(3,1),10,5,16,12000);
% A good initial point: w0 = [1.030112096647569   0.703751395701793 -1.646151244456372]';
% [wt,fk] = logistic_SGD_K(x,y,xp,xn,'logistic_f',w0,0.005,5,16,12000);
function [wt,fk] = logistic_SGD_K(x,y,xp,xn,fname,w0,a,st1,st2,K)
N = length(y);
```

```
rand('state',st1)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
d1 = length(w0);
p = [N; d1; y; x(:)];
D = [ones(1,N); x];
k = 0;
wk = w0;
fk(1) = feval(fname,wk,p);
rand('state',st2)
while k < K,
    Nw = randperm(N);
    i = Nw(1);
    ti = y(i)*D(:,i);
    dk = ti/(1+exp(wk'*ti));
    wk = wk + a*dk;
    k = k + 1;
    fk(k+1) = feval(fname,wk,p);
end
disp('Final weights:')
wt = wk
fs = feval(fname,wt,p);
disp(sprintf('Objective function at solution point: %d.', fs));
Dt = [ones(N,1) x'];
y = y(:);
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
subplot(121)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
grid
xlabel('\itx_1')
ylabel('\itx_2')
p1 = -20;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 30;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([-20 30 -25 25])
axis square
title('Decision Boundary by Logistic Regression (SGD)')
hold off
subplot(122)
```

```
plot(1:1:K+1,fk,'b-','linewidth',1.5)
xlabel('Number of iterations')
ylabel('In-sample error')
grid
axis square
axis([0 K+1 0 1.1*fk(1)])
```

(7) regression_NL.m

```
% To apply linear regression to classify nonseparable double demi-circle
% data using a dilated input space where vetors are dilated to include
% up to 3rd-order features.
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 25, 2015.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,Ein1,Ein2] = regression_NL(x,y,xp,xn,11);
function [wt,Ein1,Ein2] = regression_NL(x,y,xp,xn,st1)
N = length(y);
rand('state',st1)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
z1 = zeros(7,N);
for i = 1:N,
   z1(:,i) = [x(1,i)^2; x(1,i)*x(2,i); x(2,i)^2; x(1,i)^3; x(1,i)^2*x(2,i); x(1,i)*x(2,i)^2; x(2,i)^3];
end
z = [x; z1];
Z = [ones(N,1) z'];
y = y(:);
wt = (inv(Z'*Z))*(Z'*y);
disp('Final weights:')
wt
Ein1 = norm(Z*wt-y)^2/N;
disp(sprintf('In-sample error was found to be %d.', Ein1));
dwt = (Z*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
Ein2 = L/N;
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
[x1,x2] = meshgrid(-20:50/100:30,-25:50/100:25);
w0 = wt(1); w1 = wt(2); w2 = wt(3); w3 = wt(4); w4 = wt(5); w5 = wt(6);
w6 = wt(7); w7 = wt(8); w8 = wt(9); w9 = wt(10);
h1 = w0 + w1*x1 + w2*x2 + w3*(x1.^2) + w4*(x1.*x2) + w5*(x2.^2);
h = h1 + w6*(x1.^3) + w7*((x1.^2).*x2) + w8*(x1.*(x2.^2)) + w9*(x2.^3);
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
```

```
contour(x1,x2,h,v,'k-','linewidth',1.5);
grid
xlabel('\itx_1')
ylabel('\itx_2')
axis square
axis([-20 30 -25 25])
hold off
```

(8) logistic_SDM_NL

```
% To apply logistic regression with 3rd-order nonlinear transform
% to classify double nonseparable demi-circle data. SDM algorithm
% is chosen to minimize the logistic loss function.
% Written by W.-S. Lu, University of Victoria. Last modified: Jan. 25, 2015.
% Examples:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [wt,fs] = logistic_SDM_NL(x,y,xp,xn,'logistic_f','logistic_g',zeros(10,1),11,100000);
% [wt,fs] = logistic_SDM_NL(x,y,xp,xn,'logistic_f','logistic_g',zeros(10,1),11,200000);
function [wt,fs] = logistic_SDM_NL(x,y,xp,xn,fname,gname,w0,st,K)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
z1 = zeros(7,N);
for i = 1:N,
   z1(:,i) = [x(1,i)^2; x(1,i)*x(2,i); x(2,i)^2; x(1,i)^3; x(1,i)^2*x(2,i); x(1,i)*x(2,i)^2; x(2,i)^3];
end
z = [x; z1];
d1 = length(w0);
p = [N; d1; y; z(:)];
k = 0;
wk = w0;
while k < K,
    gk = feval(gname,wk,p);
    dk = -gk;
    ak = bt_lsearch(wk,dk,fname,gname,p);
    dtk = ak*dk;
    wk = wk + dtk;
    k = k + 1;
    % fk(k+1) = feval(fname,wk,p);
end
disp('Final weights:')
wt = wk
fs = feval(fname,wt,p);
disp(sprintf('Objective function at solution point: %d.', fs));
Z = [ones(N,1) z'];
y = y(:);
dwt = (Z*wt >= 0);
zt = dwt + dwt - y - 1;
```

```
L = sum(abs(zt))/2;
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
[x1,x2] = meshgrid(-20:50/100:30,-25:50/100:25);
w0 = wt(1); w1 = wt(2); w2 = wt(3); w3 = wt(4); w4 = wt(5); w5 = wt(6);
w6 = wt(7); w7 = wt(8); w8 = wt(9); w9 = wt(10);
h1 = w0 + w1*x1 + w2*x2 + w3*(x1.^2) + w4*(x1.*x2) + w5*(x2.^2);
h = h1 + w6*(x1.^3) + w7*((x1.^2).*x2) + w8*(x1.*(x2.^2)) + w9*(x2.^3);
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
contour(x1,x2,h,v,'k-','linewidth',1.5);
grid
xlabel('\itx_1')
ylabel('\itx_2')
axis square
axis([-20 30 -25 25])
hold off
```

(9) class_pca_digits.m

```
% To implement the example in Sec. 3.5.D for classifying handwritten digits
% from MNIST database.
% Input:
% X: Ten classes of input data, each is of size 784 by nt (here we use nt = 500).
% n: the actual size of each data matrix is 784 by n. Hence n is limited to n <= nt.
% K: the number of "eigen-digits" to be used in PCA. Typically K << m = 784.
% Te: testing data (here we use Te28 of size 784 by 10000).
% Lte: Labels for the testing data.
% Output:
% Js: the numerical values of testing digits obtained by PCA-based classification.
% er: classification error.
% Written by W.-S. Lu, University of Victoria. Last modified: March 9, 2015.
% Example: load X500; load Te28; load Lte28;
% [Js,er] = class_pca_digits(X500,500,24,Te28,Lte28);
function [Js,er] = class_pca_digits(X,n,K,Te,Lte)
U = [];
Xb = [];
t = size(X,2);
nt = t/10;
for i = 1:10,
   Tw = X(:,((i-1)*nt+1):(i*nt));
   Ti = Tw(:,1:n);
   xbi = mean(Ti')';
   Xh = Ti - xbi*ones(1,n);
   [u,s,v] = svd(Xh*Xh');
   U = [U u(:,1:K)];
   Xb = [Xb xbi];
end
```

```
Lte = Lte(:);
M = length(Lte);
Js = zeros(M,1);
for m = 1:M,
   xw = Te(:,m);
   e = zeros(1,10);
   for j = 1:10;
      Uj = U(:,((j-1)*K+1):(j*K));
      xbj = Xb(:,j);
      zj = Uj'*(xw-xbj);
      xjh = Uj*zj + xbj;
      e(j) = norm(xw-xjh);
   end
   [emin,ind] = min(e);
   Js(m) = ind - 1;
end
E = (Lte ~= Js);
er = sum(E)/M
```

Problems

2.1

(i) Download from the class website two random data sets named **xp** and xn, with size of xp being 2 by 91 and size of xn being 2 by 109. These data sets are linearly separable. You are asked to follow the steps described below to construct a training data set of the form $\mathcal{D} = \{(\boldsymbol{x}_n, y_n),$ $n = 1, 2, ..., 200\}$

Step 1: Generate label 1 for each point in xp: yp = ones(91,1), and label –1 for each point in xn: yn = -ones(109,1);

Step 2: Combine the two data sets into one: xw = [xp xn]; yw = [yp; yn];

Step 3: Generate a training set with 200 examples by randomly shuffling the rows of xw and accordingly the components of yw:

```
rand('state',5)
N1 = randperm(200);
x = xw(:,N1);
y = yw(N1);
```

Data set $\mathcal{D}$ generated above will be used as the training data for classification.

(ii) Download from the class website two random sets named as **xp_ns** and xn_ns, with size of xp_ns being 2 by 92 and size of xn_ns being 2 by 108. These data sets are non-separable by linear boundary lines. You are asked to follow the steps described below to construct a training data set of the form $\mathcal{D}_{\text{ns}} = \{(\boldsymbol{x}_n, y_n),\ n = 1, 2, ..., 200\}$

Step 1: Generate label 1 for each point in xp_ns: yp_ns = ones(92,1), and label –1 for each point in xn_ns: yn_ns = -ones(108,1);

Step 2: Combine the two data sets into one: xw = [xp_ns xn_ns]; yw = [yp_ns; yn_ns];

Step 3: Generate a training set with 200 examples by randomly shuffling the columns of xt and accordingly the components of yt:

```
rand('state',15)
N1 = randperm(200);
x = xw(:,N1);
y = yw(N1);
```

Data set $\mathcal{D}_{\text{ns}}$ generated above will be used as the training data for classification.

(iii) Apply PLA to the data set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n),\ n = 1, 2, ..., 200\}$ generated in part (i) with w(0) = zeros(3,1). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$; and include a plot showing the profile of E_{in} versus iterations. To earn credit, include your MATLAB code that implements PLA as applied to the present training data.

(iv) Apply a certain number of PLA iterations to the non-separable data set $\mathcal{D}_{\text{ns}} = \{(\boldsymbol{x}_n, y_n),$

$n = 1,2,...,200\}$ generated in part (ii) with w(0) = zeros(3,1). Note that PLA does not converge when dealing with nonseparable data, hence the algorithm needs to be modified to just run a given number (K) of iterations, then terminate. Try your best to figure out a good K that yields satisfactory results. Report the numerical results in terms of the weight $\boldsymbol{w}^*$ produced by PLA and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$; and include a plot showing the profile of E_{in} versus iterations. To earn credit, include your MATLAB code that implements PLA as applied to the present training data.

2.2

(i) Apply the pocket algorithm (PCA) to the data set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n),\ n = 1,2,...,200\}$ generated in part (i) of Prob. 2.1 with w(0) = zeros(3,1). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$; and include a plot showing the profile of E_{in} versus iterations. To earn credit, include your MATLAB code that implements PCA as applied to the present training data.

(ii) Apply a certain number of PCA iterations to the non-separable data set $\mathcal{D}_{\text{ns}} = \{(\boldsymbol{x}_n, y_n),\ n = 1,2,...,200\}$ generated in part (ii) of Prob. 2.1 with w(0) = zeros(3,1). Try your best to figure out a good K that yields satisfactory results. Report the numerical results in terms of the weight $\boldsymbol{w}^*$ produced by PCA and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$; and include a plot showing the profile of E_{in} versus iterations. To earn credit, include your MATLAB code that implements PCA as applied to the present training data.

2.3 [8] Recall that the Adaline algorithm for classification was introduced in Sec. 1.4. Here the same algorithm is derived with an optimization perspective.

(i) Define $E_n(\boldsymbol{w}) = \max\{0, 1 - y_n \boldsymbol{w}^T \boldsymbol{x}_n\}^2$. Prove that $E_n(\boldsymbol{w})$ is continuous and differentiable. Derive an expression for the gradient of $E_n(\boldsymbol{w})$.

(ii) Show that $E_n(\boldsymbol{w})$ is an upper bound for $[\![\text{sign}(\boldsymbol{w}^T \boldsymbol{x}_n) \neq y_n]\!]$. Therefore, $\frac{1}{N}\sum_{n=1}^{N} E_n(\boldsymbol{w})$ is an upper bound for the in-sample classification error $E_{\text{in}}(\boldsymbol{w})$ (see Eq. (1.14)).

(iii) Argue that the Adaline algorithm described in Sec. 2.4 performs stochastic descent on $\frac{1}{N}\sum_{n=1}^{N} E_n(\boldsymbol{w})$.

2.4 [8] Following the steps described below to derive an alternative-form linear programming (LP) algorithm, namely

$$\begin{aligned} &\underset{\boldsymbol{z}}{\text{minimize}} \quad \boldsymbol{c}^T \boldsymbol{z} \\ &\text{subject to: } \boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b} \end{aligned} \tag{P2.1}$$

that fits a linear model for classification.

(i) For linearly separable data, show that there exists a weight vector $\boldsymbol{w}$ such that

$$y_n(\boldsymbol{w}^T\boldsymbol{x}_n) \geq 1 \text{ for } n = 1,\ldots,N \tag{P2.2}$$

(ii) Formulate the problem of finding a separating w for separable data as an LP problem. You need to specify what the parameters $\boldsymbol{A}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are and what the optimization variable $\boldsymbol{z}$ is.

(iii) If the data is non-separable, the condition in (P2.2) cannot hold for every n. To deal with this problem, violation $\xi_n \geq 0$ may be introduced to capture the amount of violation for example $\boldsymbol{x}_n$. In this way, (P2.2) is modified to

$$\begin{aligned} y_n(\boldsymbol{w}^T\boldsymbol{x}_n) &\geq 1 - \xi_n \\ \xi_n &\geq 0 \end{aligned} \tag{P2.3}$$

Evidently, we want to minimize the total violation $\sum_{n=1}^{N} \xi_n$, thus we seek to find a weight vector w that solve the constrained problem

$$\begin{aligned} \underset{\boldsymbol{w},\, \xi_1,\ldots,\, \xi_N}{\text{minimize}} \quad & \sum_{n=1}^{N} \xi_n \\ \text{subject to:} \quad & y_n(\boldsymbol{w}^T\boldsymbol{x}_n) \geq 1 - \xi_n \\ & \xi_n \geq 0 \end{aligned} \tag{P2.4}$$

Re-formulate problem (P2.4) into a "standard" LP problem as seen in (P2.1) with parameters $\boldsymbol{A}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ specified.

2.5 [8] For linear regression, the out-of-sample error is given by

$$E_{\text{out}}(h) = E\left[\left(h(\boldsymbol{x}) - y\right)^2\right]$$

Show that among all hypotheses, the one that minimizes E_{out} is given by

$$h^*(\boldsymbol{x}) = E\left[y \mid \boldsymbol{x}\right]$$

The function h^* can be treated as a deterministic target function, in which case we can write $y = h^*(\boldsymbol{x}) + \varepsilon(\boldsymbol{x})$ where $\varepsilon(\boldsymbol{x})$ is an input-dependent noise variable. Show that the expected value of $\varepsilon(\boldsymbol{x})$ is zero.

2.6 Apply linear regression to the classification problem whose training data is generated part (i) in Problem 2.1. Note that you are applying a machine learning algorithm designed for regression problems to solve a classification problem. Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Also comment on the pros and cons of the linear regression algorithm relative to PLA and PCA. To earn credit, include your MATLAB code that implements the linear regression method as applied to the present training data.

2.7 Apply linear regression to the classification problem whose training data is generated part (ii) in Problem 2.1. Note that you are applying a machine learning algorithm designed for regression problems to solve a classification problem. Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include

a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Also comment on the pros and cons of the linear regression algorithm relative to PLA and PCA that were applied to the same data set.

To earn credit, include your MATLAB code that implements the linear regression method as applied to the present training data.

2.8 Apply logistic regression to the classification problem whose training data is generated part (ii) in Problem 2.1. Note that you are applying a machine learning algorithm designed for probability prediction to solve a classification problem. Use SDM with backtracking (BT) line search to minimize the logistic loss unction in (2.17). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Also comment on the pros and cons of the logistic regression algorithm relative to PLA, PCA, and linear regression that were applied to the same data set. To earn credit, include your MATLAB code that implements the logistic regression algorithm as applied to the present training data.

2.9 Apply logistic regression to the classification problem whose training data is generated part (ii) in Problem 2.1. Note that you are applying a machine learning algorithm designed for probability prediction to solve a classification problem. Use BFGS-BT to minimize the logistic loss unction in (2.17). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Also comment on the pros and cons of the present version of the logistic regression algorithm relative to that using SDM-BT. To earn credit, include your MATLAB code that implements the logistic regression algorithm as applied to the present training data.

2.10 Apply logistic regression to the classification problem whose training data is generated part (ii) in Problem 2.1. Note that you are applying a machine learning algorithm designed for probability prediction to solve a classification problem. Use SGD to minimize the logistic loss unction in (2.17). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Also comment on the pros and cons of the present version of the logistic regression algorithm relative to those using SDM-BT and BFGS-BT. To earn credit, include your MATLAB code that implements the logistic regression algorithm as applied to the present training data.

2.11 Apply linear regression with 2nd-order nonlinear transform (see (2.25)) to the classification problem whose training data is generated part (ii) in Problem 2.1. Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Compare the performance of this version of linear regression algorithm with that studied in Prob. 2.7. To earn credit, include your MATLAB code that implements the linear regression with 2nd-order nonlinear transform as applied to the present training data.

2.12 Apply linear regression with 3rd-order nonlinear transform (see (2.26)) to the classification problem whose training data is generated part (ii) in Problem 2.1. Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Compare the performance of this version of linear regression algorithm with that studied in Prob. 2.10. To earn credit, include your MATLAB code that implements the linear regression with 3rd-order nonlinear transform as applied to the present training data.

2.13 Apply logistic regression with 2nd-order nonlinear transform (see (2.25)) to the classification problem whose training data is generated part (ii) in Problem 2.1. Note that you are applying a machine learning algorithm designed for probability prediction to solve a classification problem. Use BFGS-BT to minimize the logistic loss unction in (2.17). Report the numerical results in terms of the optimized weight $\boldsymbol{w}^*$ and in-sample error E_{in} (percentage of misclassified data pairs); include a plot showing the training data as points with binary labels and the decision boundary generated by $\boldsymbol{w}^*$. Compare the performance of this version of logistic regression algorithm with that studied in Prob. 2.11. To earn credit, include your MATLAB code that implements the logistic regression algorithm with nonlinear transform as applied to the present training data.

2.14 Prove (2.37). (Hint: use Eq. (2.34))

2.15 The Frobenius norm of a real-valued matrix $\boldsymbol{A} = \{a_{i,j}\}$ is equal to the square root of the sum of all $a_{i,j}^2$.

(i) Prove that $\|\boldsymbol{A}\|_F^2 = \text{trace}(\boldsymbol{A}\boldsymbol{A}^T)$.

(ii) Use the result from part (i) to show Eq. (2.36).

2.16 The computational complexity of performing the eigen-decomposition of $\boldsymbol{C}$ in (2.41) becomes intractably high when m is very large. In the case of $n << m$, the computational burden may be greatly reduced and this exercise problem introduces such a technique.

(i) Define $\hat{\boldsymbol{X}} = \frac{1}{\sqrt{n-1}}\left[\boldsymbol{x}_1 - \bar{\boldsymbol{x}} \quad \boldsymbol{x}_2 - \bar{\boldsymbol{x}} \quad \cdots \quad \boldsymbol{x}_n - \bar{\boldsymbol{x}}\right]$. Verify that $\boldsymbol{C} = \hat{\boldsymbol{X}}\hat{\boldsymbol{X}}^T$.

(ii) Show that the nonzero eigenvalues of $\boldsymbol{C}$ are the same as the nonzero eigenvalues of $\Phi = \hat{\boldsymbol{X}}^T\hat{\boldsymbol{X}}$.

(iii) Let $\boldsymbol{v}$ be an eigenvector of Φ, show that $\hat{\boldsymbol{X}}\boldsymbol{v}$ is an eigenvector of $\boldsymbol{C}$.

(iv) Based on the results from parts (i) to (iii), develop an efficient method to compute $\boldsymbol{U}$ and $\boldsymbol{S}$ in (2.41) without computing the eigenvalues and eigenvectors of $\boldsymbol{C}$.

Chapter 3 Overfitting, Regularization and Validation

Overfitting is an important issue to understand and address in machine learning: it is a phenomenon where good fitting of training data does not necessarily imply satisfactory generalization. In this chapter, we address the question of when overfitting occurs and present two approaches to dealing with the overfitting problem in supervised learning. The materials presented here are largely based on [8].

3.1 What Is Overfitting?

Roughly speaking, overfitting refers to *fitting the data more than is warranted* [8]. The main case of overfitting is when the learning model yields a hypothesis with a low E_{in} but a high E_{out}. As will become transparent shortly, there are three things involved in overfitting data: target complexity, noise, and data size. Below we illustrate the interplay between them in relation to the overfitting phenomenon through two examples.

Example 3.1 *Overfitting with polynomials when data are noisy*

Consider the problem of fitting a polynomial $p(x)$ into15 points $\mathcal{D} = \{(x_n, y_n), n = 1, \ldots, 15\}$ in least-squares (LS) sense, where the data points are produced by a target function which is a polynomial of order 10 plus Gaussian noise with variance $\sigma^2 = 0.09$ (i.e. $\sigma = 0.3$). The MATLAB code below generates a target function and data set $\mathcal{D}$ as shown in Fig. 3.1.

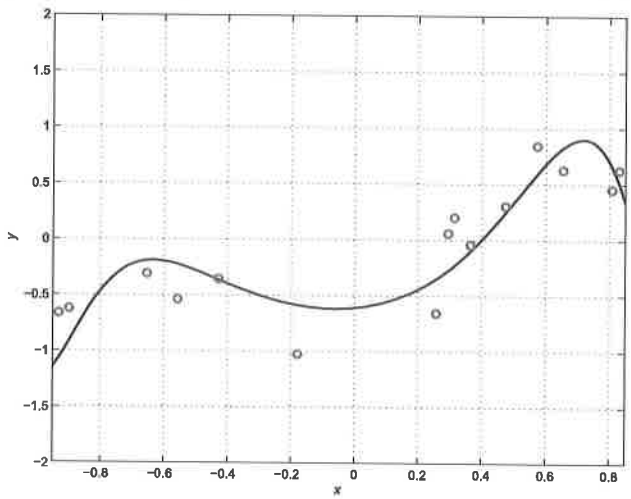


Fig. 3.1 10th-order target function and 15 noisy data points with noise variance $\sigma^2 = 0.09$.

```
rand('state',148),
x = sort(2*(rand(1,15)-0.5));
randn('state',119)
p = 5*randn(1,11);
y = polyval(p,x);
randn('state',18)
w = 0.3*randn(1,15);
yn = y + w;
```

```
xt = -0.95:1.8/1000:0.85;
yt = polyval(p,xt);
figure(1)
plot(xt,yt,'k-','linewidth',1.5)
hold on
plot(x,yn,'bo','linewidth',1.5)
grid
axis([-0.95 0.85 -2 2])
xlabel('\itx')
ylabel('\ity')
hold off
```

Next, we apply 10th order and 2nd order LS fits to 15 noisy data points :

```
pn = polyfit(x,yn,10);
zn = polyval(pn,x);
zt = polyval(pn,xt);
```

where order_m was set to 10 and 2 to carrying the 10th order and 2nd order LS fits, respectively. Figures 3.2 and 3.3 depict these two cases.

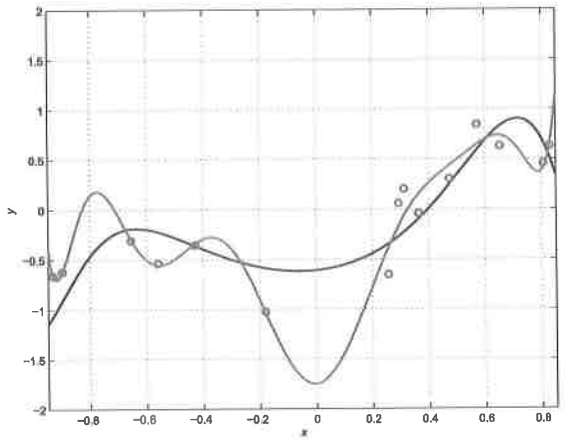


Fig. 3.2 10th order LS fit (red curve).

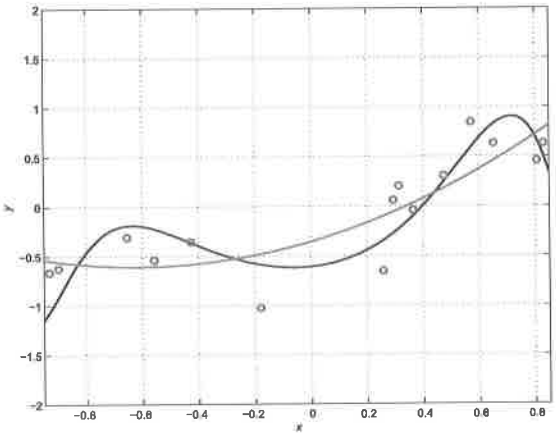


Fig. 3.3 2nd order LS fit (red curve).

The performance of the two LS fits can also be evaluated in terms of in-sample and, more importantly, out-of-sample errors. The in-sample error is defined by the average squared 2-norm of the difference between the noisy sample point and the corresponding point generated by the fitting polynomial $p(x)$, namely,

$$E_{\text{in}} = \frac{1}{N}\sum_{n=1}^{N} | p(x_n) - y_n |^2 \tag{3.1}$$

The out-of-sample error is also of average squared 2-norm type, but over the region outside the data set $\mathcal{D}$. In this case study the $\{x_n\}$ in data $\mathcal{D}$ are chosen at random over interval [–1, 1], while the $\{x_t, t = 1, \ldots, T\}$ are dense grids that are placed uniformly over [–1, 1]. Thus

$$E_{\text{out}} = \frac{1}{T}\sum_{t=1}^{T} | p(x_t) - y_t |^2 \tag{3.2}$$

where y_t was generated by the target function at x_t plus the noise. In our simulation $N = 15$ and T was set to 1001. The MATLAB code associated this part of simulation is given below:

```
ein = (norm(yn-zn)^2)/15;
randn('state',18)
wn = sig*randn(1,1001);
ytn = yt + wn;
eout = (norm(ytn-zt)^2)/1001;
```

Based on the definitions from (3.1) and (3.2), the outcome of numerical evaluations are as follows. For the 10th order fit, $E_{\text{in}} = 0.0201$ and $E_{\text{out}} = 0.3250$, while for the 2nd order fit, $E_{\text{in}} = 0.0834$ and $E_{\text{out}} = 0.1544$. The numerical results indicate a clear case of overfitting. Here we make two observations: (i) The target function is also a 10th order polynomial. Therefore, choosing a 10th order polynomial to fit the data seems reasonable and justified, so why overfitting has occurred? One explanation is that it has to do with the presence of noise in the data– overfitting occurs because the 10th order polynomial tries to fit the noise! Note that the 2nd order fit is considerably less capable of fitting the noisy data which, on one hand increases E_{in} relative to that achieved by the 10th order fit, but on the other hand tends to be less sensitive to the noise, hence a reduced E_{out}. (ii) Put it in another perspective, overfitting may occur when the *quality* of the data does not match the target complexity. ■

A question that naturally arises is whether the observations made above are merely a coincidence. Addressing the question requires simulation studies where a large number of experiments at a range of noise levels are conducted to see how overfitting in average is related to the noise level in the data given the target complexity and a certain order of polynomial fit. To prepare the simulations, we introduce a class of polynomials known as *Legendre polynomials* whose orders provide a better indication of their complexity than the monomials $\{x^k, k = 0, 1, 2 \ldots\}$ do.

Legendre Polynomials

The Legendre polynomials $\{L_k(x), k = 0, 1, \ldots\}$ are defined over interval $[-1, 1]$ by

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}\left[\left(x^2 - 1\right)^k\right] \quad \text{for } k = 0, 1, \ldots \tag{3.3}$$

It follows that $L_0(x) = 1$, $L_1(x) = x$. Several properties of the Legendre polynomials are listed below.

(i) $L_k(-x) = (-1)^k L_k(x)$

(ii) $L_k(-1) = (-1)^k$, $L_k(1) = 1$

(iii) Orthogonality of Legendre polynomials:

$$\int_{-1}^{1} L_k(x) L_l(x) dx = \frac{2}{2k+1}\delta(k-l)$$

where $\delta(i)$ equals to one when $i = 0$ and zero elsewhere.

(iv) With $L_0(x) = 1$ and $L_1(x) = x$, $L_k(x)$ for $k > 1$ can be found by recursion as

$$L_{k+1}(x) = \frac{(2k+1)}{k+1} x \cdot L_k(x) - \frac{k}{k+1} L_{k-1}(x) \quad \text{for } k = 1, 2, \ldots \tag{3.6}$$

The MATLAB function that computes the coefficients of $L_k(x)$ for $k = 0, 1, \ldots n$ is coeff_legendre.m which is enclosed at the end of this chapter.

Fig. 3.4 shows the first six Legendre polynomials from which we can sense how the "complexity" of the polynomials grows with the order (degree).

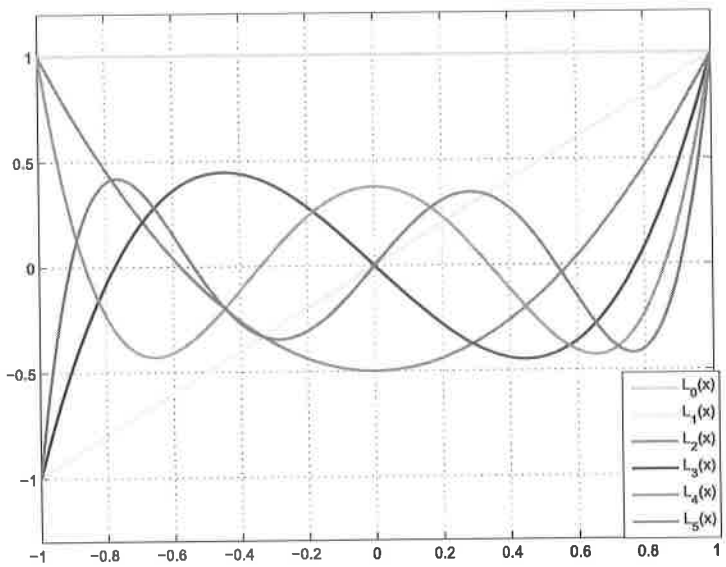


Fig. 3.4 Legendre polynomials $L_0(x), \ldots, L_5(x)$.

We now consider the case with a 10th order target polynomial of the form

$$f(x) = \sum_{k=0}^{10} \alpha_k L_k(x)$$

where $\{\alpha_k\}$ are chosen at random and normalized so that $E_x[f(x)^2] = 1$. We let standard deviation σ of the noise in the data vary from $\sigma = 0$ to $\sigma = 0.5$, each time with an increment of 0.025. At each noise level, we carry out 1000 runs for 10th order and 2nd order fits, compute the average E_{out} for each fit, and use the difference $E_{out}[g_{10}(x)] - E_{out}[g_2(x)]$ as an overfit measure. The results are depicted in Fig. 3.5. The figure shows how overfitting gets worse as data gets noisier.

The MATLAB function that implements the simulation and generates Fig. 3.5 are given is ex3_1b.m which is enclosed at the end of this chapter.

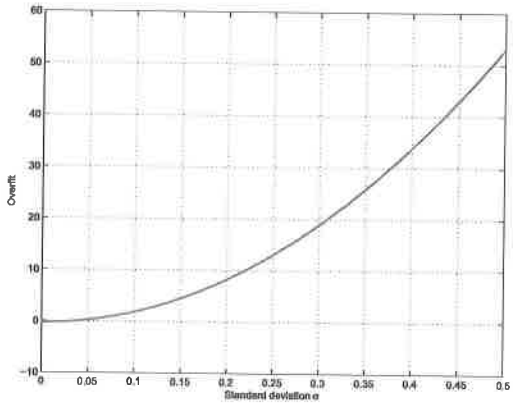


Fig. 3.5 Overfit as a function of standard deviation σ of the noise in data set. The order of the target function was set to 10, with its coefficients chosen at random and normalized. The overfit shown here was the averaged $E_{out}[g_{10}(x)] - E_{out}[g_2(x)]$ over 1000 runs. See the MATLAB code below for implementation details.

Next, we fixed a noise level to $\sigma = 0.4$ and order of target polynomial to $Q = 10$, and evaluated averaged overfit as the data size N varies in the range from 15 to 120, each time by an increment of 5. The results are depicted in Fig. 3.6. The figure shows how overfitting vanishes as data size gets larger.

Example 3.2 *Overfitting with polynomials for a highly complex target*

In this example, we look at how overfitting is related to target complexity. To focus on this aspect of overfitting, the data sets in our simulations were free of noise contamination. In the first set of simulations, 10th and 2nd order LS fits were applied to 15 data points that were selected at random over [–1, 1] for a target polynomial of order 50 that was generated by random combination of 51 Legendre polynomials of orders varying from 0 to 50. The results of the two fits are depicted in Figs. 3.7 and 3.8.

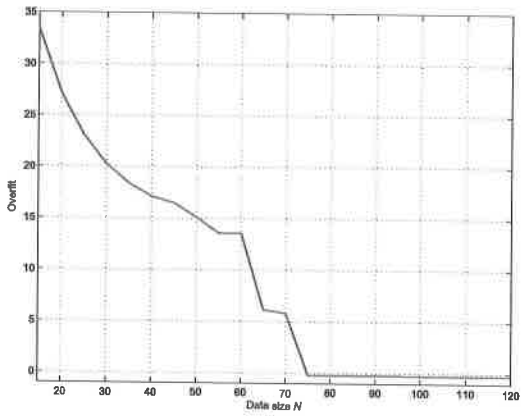


Fig. 3.6 Overfit as a function of data size N. The standard deviation σ of the noise in data set was set to 0.4; the order of the target function was set to $Q = 10$ with its coefficients chosen at random and normalized. The overfit shown here was the averaged $E_{out}[g_{10}(x)] - E_{out}[g_2(x)]$ over 2000 runs.

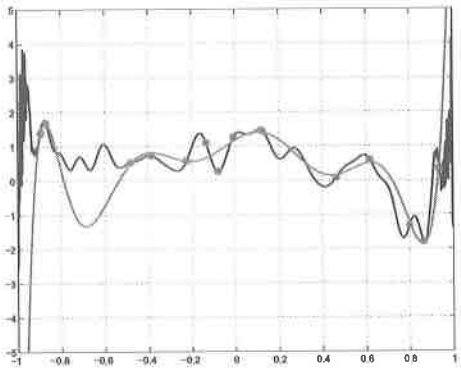


Fig. 3.7 10th order LS fit of a 50th order target.

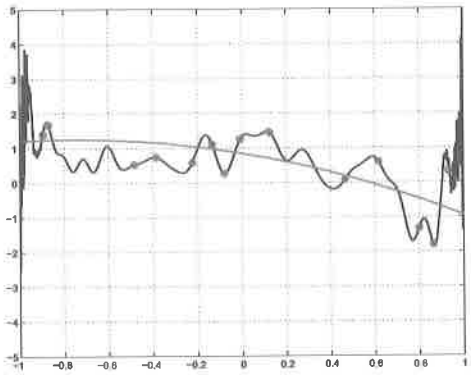


Fig. 3.8 2nd order LS fit of a 50th order target.

For the 10th order fit, $E_{in} = 0.0385$ and $E_{out} = 5.2772$, while for the 2nd order fit, $E_{in} = 0.4419$ and $E_{out} = 0.5655$. Thus with hypothesis $\mathcal{H}_{10}$ overfitting has occurred over hypothesis $\mathcal{H}_2$, where $\mathcal{H}_r$ denotes the set of polynomials of order r. We see that, between $\mathcal{H}_{10}$ and $\mathcal{H}_2$, a learning model with hypothesis set $\mathcal{H}_2$ provides better generalization performance for a given 50th order target function. On the other hand, the optimal hypothesis from $\mathcal{H}_2$ (the red curve shown in Fig. 3.8) differs considerably from the target function (the black curve in Fig. 3.8). The difference between these curves is called *deterministic noise*. The term comes from the fact that it is not stochastic because the data was assumed to be perfectly accurate, however it cannot be modeled by an $\mathcal{H}_2$ hypothesis just like the stochastic noise in the data that cannot be modeled by $\mathcal{H}_2$.

Next, we looked at three cases, each with a fixed data size N and 10th and 2nd order fits are applied to approximate a target polynomial whose order varied from $Q = 10$ to $Q = 46$ with an increment of 2. The averaged overfit over 2000 runs of the three cases with $N = 15$, 20, and 25 are shown in Fig. 3.9. It is observed that (i) overfitting grows with target complexity Q; (ii) overfitting reduces as data size N increases. ■

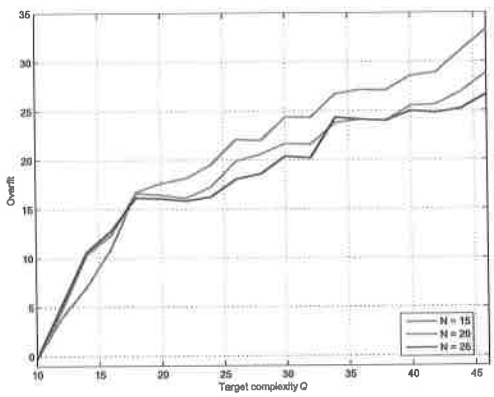


Fig. 3.9 Overfit as a function of target complexity Q. The noise level σ was set to 0; the order of the target function varied from $Q = 10$ and $Q = 46$. The overfit shown here was the averaged $E_{out}[g_{10}(x)] - E_{out}[g_2(x)]$ over 2000 runs.

Summarizing the observations made through the two examples, the primary causes of overfitting are stochastic noise that degrades the *quality* of training data and deterministic noise due to higher target complexity. The size of the data (i.e., *quantity*) has proven to play an important role in the overfitting phenomenon – increasing the data size in general reduces or even eliminates overfitting. In other words, "what matters is how the model complexity matches the *quantity and quality* of the data we have, not how it matches the target function [8]."

3.2 Regularization Techniques

We have seen that overfitting is closely related to size of training data as well as target complexity. Regularization is a general technique to address overfitting problems that has found effective regardless of whether the overftting is due to insufficient data size or high target complexity. One way to understand the machinery of regularization is through the VC-dimension analysis. Recall Eq. (1.30), namely,

$$E_{\text{out}}(h) \le E_{\text{in}}(h) + \Omega(N, \mathcal{H}, \delta) \tag{3.7}$$

where

$$\Omega(N, \mathcal{H}, \delta) = \sqrt{\frac{8}{N} \ln \frac{4m_{\mathcal{H}}(2N)}{\delta}} \le \sqrt{\frac{8}{N} \ln \frac{4(2N)^{d_{\text{vc}}} + 4}{\delta}}$$

is controlled by data size N, confidence parameter δ, and VC-dimension d_{vc}. In a typical learning problem N and δ are fixed, hence $\Omega(N, \mathcal{H}, \delta)$ depends only on d_{vc} which indicates the complexity of hypothesis $\mathcal{H}$. Thus (3.7) becomes

$$E_{\text{out}}(g) \le E_{\text{in}}(g) + \Omega(\mathcal{H}) \tag{3.8}$$

Roughly speaking, *regularized learning* minimizes a combination of $E_{\text{in}}(h)$ with $\Omega(h)$ over $h \in \mathcal{H}$ instead of minimizing the in-sample error $E_{\text{in}}(h)$ alone, that is

$$\underset{h}{\text{minimize}} \;\; E_{\text{in}}(h) + \lambda \Omega(h) \tag{3.9}$$

where $\Omega(h)$ is a penalty term for the complexity of hypothesis h, and $\lambda \ge 0$ is a parameter that controls the amount of regularization. From (3.8), it is quite clear that regularization is justified as it effectively prevents $E_{\text{out}}(h)$ from being too large.

A. Regularization with L_2-Penalty

Example 3.3 *Use regularization to handle insufficient data size*

Again we consider Example 3.1 illustrated in Figs. 3.1 to 3.3, where the target function was a 10th order polynomial and a 10th order LS was found to overfit when it applied to a set of 15 noisy data points. Here we re-formulate the problem as a regularized LS problem. Let $f(x)$ be the target function and $h(x, \boldsymbol{w})$ be the 10th order hypothesis polynomial with coefficient vector $\boldsymbol{w} = [w_0 \; w_1 \cdots w_{10}]^T$, i.e.,

$$h(x, \boldsymbol{w}) = w_0 + w_1 x + \cdots + w_{10} x^{10}$$

We employ an L_2-loss function to measure the in-sample error

$$E_{\text{in}}(\boldsymbol{w}) = \sum_{n=1}^{N} | h(x_n, \boldsymbol{w}) - f(x_n) |^2 = \sum_{n=1}^{N} | h(x_n, \boldsymbol{w}) - y_n |^2 \tag{3.10}$$

and an L_2 penalty term $\Omega(h) = \| \boldsymbol{w} \|_2^2$. In this way, scheme (3.10) is realized by a regularized least-squares formulation as

$$\underset{\boldsymbol{w}}{\text{minimize}} \;\; E_{\text{in}}(\boldsymbol{w}) + \lambda \| \boldsymbol{w} \|_2^2 \tag{3.11}$$

Taking a perspective from VC-analysis, the solution weight $\boldsymbol{w}^*$ of problem (3.11) shall keep $E_{\text{in}}(\boldsymbol{w})$ sufficiently small subject to $\boldsymbol{w}$ being in a limited vicinity of the origin in the $\boldsymbol{w}$ space thereby reducing the *effective* VC dimension [8] of $\mathcal{H}$. It follows from (3.8) that such a $\boldsymbol{w}^*$ is expected to offer a reduced E_{out} relative to that achieved by minimizing $E_{\text{in}}(\boldsymbol{w})$ alone.

With an L_2-loss $E_{\text{in}}(\boldsymbol{w})$ in (3.10), there is a closed-form solution for problem (3.11) which we derive as follows. Let $\boldsymbol{x}_n = [1 \;\; x_n \;\; x_n^2 \cdots x_n^{10}]^T$, $\boldsymbol{y} = [y_1 \;\; y_2 \;\; \cdots \;\; y_N]^T$, and write $h(x_n, \boldsymbol{w}) = \boldsymbol{x}_n^T \boldsymbol{w}$, the in-sample error in (3.10) becomes $\| \boldsymbol{X}\boldsymbol{w} - \boldsymbol{y} \|_2^2$ with $\boldsymbol{X} \in R^{N \times 11}$ whose nth row is $\boldsymbol{x}_n^T$, the objective function in (3.11) is given by

$$\boldsymbol{w}^T \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}_{11} \right) \boldsymbol{w} - 2\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{y} + \text{const}$$

whose global minimizer is given by

$$\boldsymbol{w}^* = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}_{11} \right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{3.12}$$

For the example at hand, we applied (3.11) with $\lambda = 0.015$ for a regularized 10th order LS fit. The performance is shown in Fig. 3.10 which on comparing with the 10th order LS fit without regularization (see Fig. 3.2) demonstrates considerable improvement. Actually, the overfit has been eliminated as can be seen from the table below.

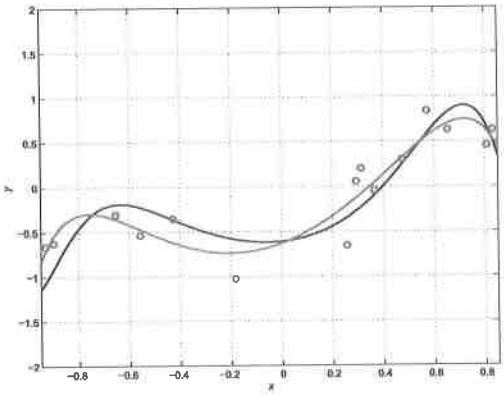
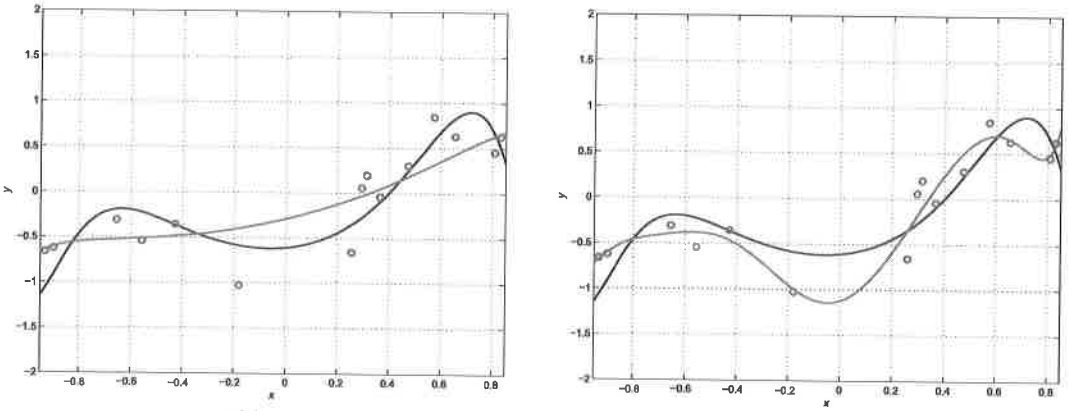


Fig. 3.10 L_2-regularized 10th order LS fit (red curve).

	E_{in}	E_{out}
2nd order LS fit without regularization	0.0834	0.1544
10th order LS fit without regularization	0.0201	0.3250
10th order LS fit with L_2-regularization ($\lambda = 0.015$)	0.0408	0.1218

The value of regularization parameter λ greatly affects solution's performance. As shown in the figures below, a λ too large (*over-regularization*, see Fig. 3.11a) or a λ too small (*under-regularization*, see Fig. 3.11b) leads to an increased out-of-sample error E_{out} relative to that with an optimal λ. The table enclosed below lists the numerical values of E_{in} and E_{out} for the three

regularized solutions. It follows that using an appropriate value of λ is of practical importance. We shall come back to this issue in the next section where an approach based on cross validation is studied. ■



(a) (b)

Figure 3.11 (a) A over-regularized 10th order LS fit with $\lambda = 0.3$ and (b) a under-regularized 10th order LS fit with $\lambda = 10^{-6}$.

	E_{in}	E_{out}
10th order LS fit with L_2-regularization ($\lambda = 0.3$)	0.0820	0.1515
10th order LS fit with L_2-regularization ($\lambda = 0.015$)	0.0408	0.1218
10th order LS fit with L_2-regularization ($\lambda = 10^{-6}$)	0.0257	0.1728

The MATLAB function used in this example is ex3_1a_re.m which is enclosed at the end of this chapter.

Example 3.4 *Use regularization to handle high target complexity*

We now re-examine Example 3.2 illustrated in Fig. 3.7 where the target function was a 50th order polynomial and overfitting occurred when a 10th order LS fir was applied to a set of 15 data points that are selected from [–1, 1] at random and free of noise contamination. Again we re-formulate the problem as a regularized LS problem as we did in the first example. Because the LS fit also involves a 10th order polynomial, the regularized solution assumes the same form as in (3.12), namely,

$$\boldsymbol{w}^* = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_{11}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

With $\lambda = 0.01$, a regularized 10th order LS fit was obtained, whose performance is shown in Fig. 3.12. On comparing with the 10th order LS fit without regularization (see Fig. 3.2), the regularized solution demonstrates considerable improvement. The table below gives numerical results that compares the regularized 10th order LS fit with those obtained from 2nd order and 10th order LS fits without regularization. ■

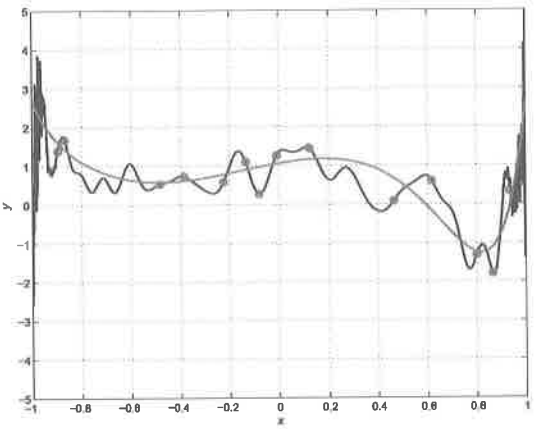


Fig. 3.12 L_2-regularized 10th order LS fit for a 50th order target function using 15 noise-free random data points.

	E_{in}	E_{out}
2nd order LS fit without regularization	0.4419	0.5655
10th order LS fit without regularization	0.0385	5.2772
10th order LS fit with L_2-regularization ($\lambda = 0.01$)	0.1644	0.3287

B. *Regularization with L_1-Penalty for Sparse Solutions*

B.1 *L_2-L_1 Minimization*

This part requires some background knowledge on convex functions with Lipschitz continuous gradients. A continuously differentiable function $u(\boldsymbol{x})$ with $\boldsymbol{x} \in R^n$ is said to have Lipschitz continuous gradient if

$$\|\nabla u(\boldsymbol{x}) - \nabla u(\boldsymbol{y})\|_2 \le L\|\boldsymbol{x} - \boldsymbol{y}\|_2 \tag{3.13}$$

holds for any $\boldsymbol{x}$ and $\boldsymbol{y}$, where $L > 0$ is a (Lipschitz) constant. An example of such function is $u(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$ whose gradient is given by $\nabla u(\boldsymbol{x}) = \boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^T\boldsymbol{b}$, hence

$$\|\nabla u(\boldsymbol{x}) - \nabla u(\boldsymbol{y})\|_2 = \|\boldsymbol{A}^T\boldsymbol{A}(\boldsymbol{x} - \boldsymbol{y})\|_2 \le \lambda_{\max}(\boldsymbol{A}^T\boldsymbol{A})\|\boldsymbol{x} - \boldsymbol{y}\|_2$$

Therefore $u(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$ has Lipschitz continuous gradient with $L = \lambda_{\max}(\boldsymbol{A}^T\boldsymbol{A})$.

Recall that a smooth function $u(\boldsymbol{x})$ is convex if and only if all tangent lines of the function lie underneath the function's graph. An analytical form of the above statement is

$$u(\boldsymbol{x}) \ge u(\boldsymbol{y}) + \nabla^T u(\boldsymbol{y})(\boldsymbol{x} - \boldsymbol{y})$$

In other words, *a convex function is globally bounded from below by a linear function*. Now if in addition $u(\boldsymbol{x})$ is convex with a Lipschitz gradient, then $u(\boldsymbol{x})$ is also *globally bounded from above* by a convex quadratic function with Hessian $L \cdot \boldsymbol{I}$, namely,

$$u(\boldsymbol{y}) + \nabla^T u(\boldsymbol{y})(\boldsymbol{x} - \boldsymbol{y}) \le u(\boldsymbol{x}) \le u(\boldsymbol{y}) + \nabla^T u(\boldsymbol{y})(\boldsymbol{x} - \boldsymbol{y}) + \frac{L}{2}\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 \tag{3.14}$$

Proof: We write

$$u(\boldsymbol{x}) = u(\boldsymbol{y}) + \int_0^1 \nabla^T u(\boldsymbol{y} + \tau(\boldsymbol{x} - \boldsymbol{y}))\,(\boldsymbol{x} - \boldsymbol{y})\,d\tau$$

$$= u(\boldsymbol{y}) + \nabla^T u(\boldsymbol{y})\,(\boldsymbol{x} - \boldsymbol{y}) + \int_0^1 [\nabla u(\boldsymbol{y} + \tau(\boldsymbol{x} - \boldsymbol{y})) - \nabla u(\boldsymbol{y})]^T (\boldsymbol{x} - \boldsymbol{y})\,d\tau$$

Hence

$$\left| u(\boldsymbol{x}) - u(\boldsymbol{y}) - \nabla^T u(\boldsymbol{y})\,(\boldsymbol{x} - \boldsymbol{y}) \right|$$

$$= \left| \int_0^1 [\nabla u(\boldsymbol{y} + \tau(\boldsymbol{x} - \boldsymbol{y})) - \nabla u(\boldsymbol{y})]^T (\boldsymbol{x} - \boldsymbol{y})\,d\tau \right|$$

$$\le \int_0^1 \left| [\nabla u(\boldsymbol{y} + \tau(\boldsymbol{x} - \boldsymbol{y})) - \nabla u(\boldsymbol{y})]^T (\boldsymbol{x} - \boldsymbol{y}) \right| d\tau$$

$$\le \int_0^1 \left\| \nabla u(\boldsymbol{y} + \tau(\boldsymbol{x} - \boldsymbol{y})) - \nabla u(\boldsymbol{y}) \right\|_2 \cdot \left\| \boldsymbol{x} - \boldsymbol{y} \right\|_2 d\tau$$

$$\le \int_0^1 \tau L \left\| \boldsymbol{x} - \boldsymbol{y} \right\|_2^2 d\tau = \frac{L}{2} \left\| \boldsymbol{x} - \boldsymbol{y} \right\|_2^2$$

The proof is complete. ∎

If we denote the upper bound in (3.14) by $q(\boldsymbol{x}, \boldsymbol{y})$, we can write $q(\boldsymbol{x}, \boldsymbol{y})$ as a complete square plus a constant:

$$q(\boldsymbol{x}, \boldsymbol{y}) = u(\boldsymbol{y}) + \nabla^T u(\boldsymbol{y})(\boldsymbol{x} - \boldsymbol{y}) + \frac{L}{2} \left\| \boldsymbol{x} - \boldsymbol{y} \right\|_2^2 = \frac{L}{2} \left\| \boldsymbol{x} - \left(\boldsymbol{y} - \tfrac{1}{L} \nabla u(\boldsymbol{y}) \right) \right\|_2^2 + \text{const} \tag{3.15}$$

From (3.15) the minimizer of $q(\boldsymbol{x}, \boldsymbol{y})$ can readily be found as

$$\boldsymbol{x}^* = \boldsymbol{y} - \tfrac{1}{L} \nabla u(\boldsymbol{y}) \tag{3.16}$$

Fig. 3.13 illustrates the above analysis with an example of $u(x) = \frac{1}{2}(x-1)^2 - 4\log x + 3$ which is shown as a black solid curve. It is easy to verify that $u(x)$ is convex with a Lipschitz gradient (first-order derivative) with $L = 2$ over interval [2, 4]. At point $y = 3$, the dashed blue line was produced by the linear lower bound (see the left-hand side of (3.14)), namely,

$$u(y) + \nabla^T u(y)(x - y) = 0.6056 + 0.6667(x - 3) = 0.6667x - 1.3944$$

while the dashed red curve was generated by the convex quadratic upper bound (see the right-hand side of (3.14)), i.e.,

$$u(y) + \nabla^T u(y)(x - y) + \frac{L}{2} \left\| x - y \right\|_2^2 = 0.6056 + 0.6667(x - 3) + (x - 3)^2$$

Notice the two bounds are tightest at point $y = 3$. By (3.16), the convex quadratic upper bound function has a unique global minimizer at $x^* = 3 - 0.6667/2 = 2.6667$ which is much closer to the true minimizer of $u(x)$ relative to what the "preceding iterate" $y = 3$ is to the true minimizer. Together, Eqs. (3.15) and (3.16) play a fundamental role in the development of *proximal point algorithms* which we shall study below.

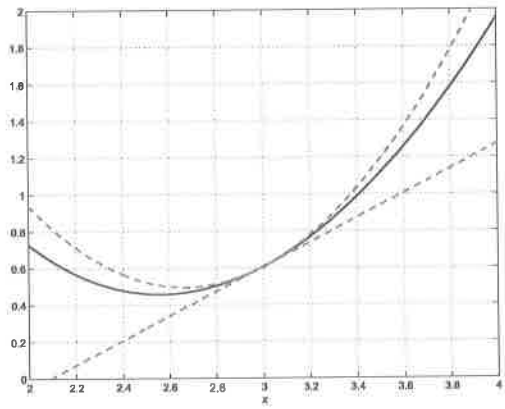


Fig. 3.13 An illustration of property (3.14).

We now develop a solution method for the L_2-L_1 problem

$$\text{minimize} \quad F(\boldsymbol{x}) = \tfrac{1}{2} \| \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} \|_2^2 + \lambda \| \boldsymbol{x} \|_1 \tag{3.17}$$

where $\boldsymbol{A}$ is an *arbitrary* matrix. As the first step of the solution method, we consider a simple special case of (3.17) with matrix $\boldsymbol{A}$ being the *identity matrix*, thus the problem at hand becomes

$$\text{minimize} \quad F(\boldsymbol{x}) = \tfrac{1}{2} \| \boldsymbol{x} - \boldsymbol{b} \|_2^2 + \lambda \| \boldsymbol{x} \|_1 \tag{3.18}$$

The objective function in (3.18) can be expressed as

$$F(\boldsymbol{x}) = \sum_{i=1}^{n} \left[\tfrac{1}{2}(x_i - b_i)^2 + \lambda \, | x_i | \right] \tag{3.19}$$

Therefore, minimizing $F(\boldsymbol{x})$ in (3.19) can be accomplished by minimizing n single-variable problems of the form

$$\underset{x}{\text{minimize}} \quad \tfrac{1}{2}(x - b)^2 + \lambda \, | x | \tag{3.20}$$

Fig. 3.14 illustrates the component functions in problem (3.20), assuming $b = 1$ and $\lambda = 0.6$. It is easy to verify that for arbitrary b and $\lambda > 0$, the global minimizer of (3.20) is given by

$$x^* = \begin{cases} \text{sign}(b)(|b| - \lambda) & \text{if } |b| \geq \lambda \\ 0 & \text{if } |b| < \lambda \end{cases} \tag{3.21}$$

Based on (3.21), the global minimizer of problem (4.18) is given by

$$\boldsymbol{x}^* = \text{sign}(\boldsymbol{b}) \cdot \max\{|\boldsymbol{b}| - \lambda, \, \boldsymbol{0}\} \tag{3.22a}$$

with the expression on the right-hand side performed *componentwisely*. From (3.22a) we see that the solution can be found by "shrinking" componentwisely data vector $\boldsymbol{b}$ by λ towards the origin for those b_i with magnitude larger than λ and setting other components to zero. Hence the method is known as a *soft-shrinkage* method. In what follows, we use the notation $S_\lambda(\boldsymbol{b})$ to denote the right-hand side of (3.22), i.e.,

$$S_\lambda(\boldsymbol{b}) \triangleq \text{sign}(\boldsymbol{b}) \cdot \max\{|\boldsymbol{b}| - \lambda, \, \boldsymbol{0}\} \tag{3.22b}$$

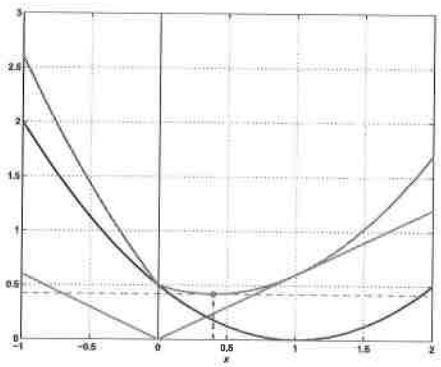


Fig. 3.14 The blue curve represents the objective function in (3.20) with $b = 1$ and $\lambda = 0.6$. Its 1st and 2nd terms are displayed in black and red, respectively. The function reaches its global minimum at $x = 0.4$, verifying the formula in (3.21).

We remark that the solution method described above is non-iterative and only requires a small amount of computations. This is made possible because the objective function in (3.18) is *separable* in its variables (as can be seen explicitly from (3.19)) so that one can treat the problem as a set of simple one-variable convex problems.

We are now ready to tackle the general L_2-L_1 problem in (3. 17). The objective functions in (3.17) is convex and of the form $F(\boldsymbol{x}) = u(\boldsymbol{x}) + \psi(\boldsymbol{x})$ where $u(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$ possesses Lipschitz continuous gradient because $\nabla u(\boldsymbol{x}) = \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$ with Lipschitz constant $L = \lambda_{\max}(\boldsymbol{A}^T\boldsymbol{A})$. Recall (3.14) that offers such a $u(\boldsymbol{x})$ with a global upper bound that is strongly convex and quadratic with *separate* variables and is tight at point $\boldsymbol{x}_k$. We take this as a motivation to consider sequentially minimizing the so-called *proximal-point function* for objective function $F(\boldsymbol{x})$. In the kth iteration with point $\boldsymbol{x}_k$ given, the proximal-point function is defined as

$$v(\boldsymbol{x}, \boldsymbol{x}_k) = u(\boldsymbol{x}_k) + \nabla^T u(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \tfrac{L}{2}\|\boldsymbol{x} - \boldsymbol{x}_k\|_2^2 + \lambda\|\boldsymbol{x}\|_1 \tag{3.23}$$

which is an upper bound of $F(\boldsymbol{x})$. To this end, we combine the first three terms of $v(\boldsymbol{x}, \boldsymbol{x}_k)$ as

$$u(\boldsymbol{x}_k) + \nabla^T u(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \tfrac{L}{2}\|\boldsymbol{x} - \boldsymbol{x}_k\|_2^2 = \tfrac{L}{2}\left\|\boldsymbol{x} - \left(\boldsymbol{x}_k - \tfrac{1}{L}\nabla u(\boldsymbol{x}_k)\right)\right\|_2^2 + \text{constant}$$

hence up to a constant the proximal-point function becomes

$$v(\boldsymbol{x}, \boldsymbol{x}_k) = \tfrac{L}{2}\left\|\boldsymbol{x} - \left(\boldsymbol{x}_k - \tfrac{1}{L}\nabla u(\boldsymbol{x}_k)\right)\right\|_2^2 + \lambda\|\boldsymbol{x}\|_1 \tag{3.24}$$

or simply

$$v(\boldsymbol{x}, \boldsymbol{x}_k) = \tfrac{L}{2}\left\|\boldsymbol{x} - \boldsymbol{b}_k\right\|_2^2 + \lambda\|\boldsymbol{x}\|_1 \tag{3.25a}$$

where

$$\boldsymbol{b}_k = \boldsymbol{x}_k - \tfrac{1}{L}\nabla u(\boldsymbol{x}_k) \tag{3.25b}$$

On comparing (3.25) with (3.17), we see that in (3.25) the term $\frac{L}{2}\left\|\boldsymbol{x} - \boldsymbol{b}_k\right\|_2^2$ replaces the term $\frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$, making (3.25) considerably easier to solve relative to that of (3.17), because the variables in (3.25a) are *separate*. The algorithmic steps of this method are sketched below.

Algorithm for (3.17) Using Sequential Minimization of Proximal-Point Functions

- Input: data A and $\boldsymbol{b}$, regularization parameter λ, initial point $\boldsymbol{x}_0$, Lipschitz constant L, and number of iterations K. Set $k = 0$.
- While $k < K$, do

  (1) Compute $\boldsymbol{x}_{k+1} = S_{\lambda/L}\left\{\frac{1}{L}A^T(\boldsymbol{b} - A\boldsymbol{x}_k) + \boldsymbol{x}_k\right\}$

  $\|w\|_\infty = \max_{1 \le i \le n} |w_i|$

  (2) Set $k = k + 1$;

  End

  $\|w\|_1 = \sum_{i=1}^{n} |w_i|$

  $\|w\|_2 = \sqrt{\sum_{i=1}^{n} |w_i|^2}$

- Output solution $\boldsymbol{x}^* = \boldsymbol{x}_K$.

norm(W, 1)

norm(W, 2) = norm(W)

norm(W, 'inf')

B.2 *Regularization with L_1 Penalty*

Again we employ an L_2-loss function to measure the in-sample error

$$E_{\text{in}}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} | h(x_n, \boldsymbol{w}) - f(x_n) |^2 = \frac{1}{2}\sum_{n=1}^{N} | h(x_n, \boldsymbol{w}) - y_n |^2 = \frac{1}{2} \| X\boldsymbol{w} - \boldsymbol{y} \|_2^2$$

CVX

but instead of using an L_2 penalty term, here we use an L_1 penalty $\Omega(h) = \| \boldsymbol{w} \|_1$ that leads to an L_1-regularized LS problem as

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad \tfrac{1}{2} \| X\boldsymbol{w} - \boldsymbol{y} \|_2^2 + \lambda \| \boldsymbol{w} \|_1 \tag{3.26}$$

Up to the difference in notation, the problems in (3.17) and (3.26) are identical, therefore the algorithm developed in Sec. 3.2.B.1 is directly applicable to (3.26).

Again we consider the first example studied in Sec. 3.2.A where the target function was a 10th order polynomial and the problem was formulated as an L_2-regularized 10th-order LS problem. Here we re-formulate the problem as an L_1-regularized LS problem instead, as seen in (3.26).

The data matrices X and $\boldsymbol{y}$ were identical to those from the example in Sec. 3.2.A. With $\lambda = 0.03$, the algorithm converged to the global minimizer of problem (3.26) after 10^4 iterations (soft-shrinkages). The numerical values of $\boldsymbol{w}^*$ are shown below together with the optimal solution obtained by solving the L_2-regularied LS problem for comparison purposes (in descending order). Performance of the L_1-regularized solution is depicted in Fig. 3.15. On comparing it with the L_2-regularied solution (see Fig. 3.10), the L_1 solution offers slightly reduced E_{out}. In addition, the sparse solution obtained by L_1 regularization helps identify a 6th-order (6th) polynomial model

L_1-regularied, $\lambda = 0.03$	L_2-regularied, $\lambda = 0.015$
0	-0.494990088844840
0	-0.173442588201798
0	-0.824953376793348
0	-0.262285486372259
-2.474821442899334	-1.121227504561928
-0.670748903773600	-0.352431834177799
0	-0.772568032119878
-0.034628447591051	-0.305879796539401
2.160179872543816	2.497167235882199

0.933464058245025	1.036020538249031
-0.585403475901743	-0.632437434927223

(with one zero coefficient), hence faster implementation.

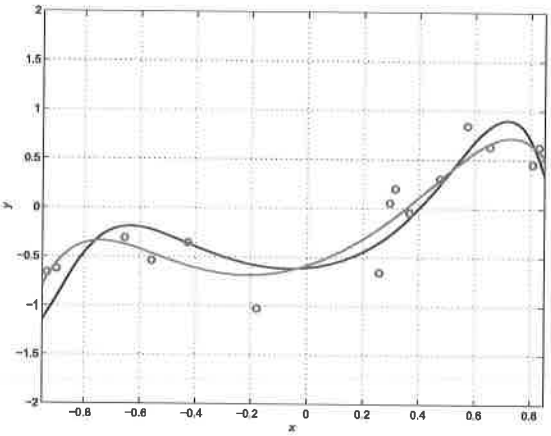


Figure 3.15 L_1-regularized 10th order LS fit (red curve) with $\lambda = 0.03$.

	E_{in}	E_{out}
10th order LS fit with L_2-regularization ($\lambda = 0.015$)	0.0408	0.1218
10th order LS fit with L_1-regularization ($\lambda = 0.03$)	0.0436	0.1178

The MATLAB function used for the above simulation is ex3_1a_re1a which is enclosed at the end of this chapter.

3.3 Cross Validation

Validation and cross validation are useful techniques for estimating E_{out} [3],[8]. These techniques can also be applied to model selection and estimation of regularization parameter λ.

A. *Validation Set*

Given a data set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, N\}$, we partition it into a training set $\mathcal{D}_{train}$ of size $N - K$ and a *validation set* $\mathcal{D}_{val}$ of size K. As expected, $\mathcal{D}_{train}$ is used in learning process like before, except that its size is reduced from N to $N - K$. We denote the final hypothesis identified based on $\mathcal{D}_{train}$ and a learning model of choice by $g^-(\boldsymbol{x})$. The role of the validation set is nearly the same as a test set in the sense that it is *independent* of the training data and it is used to compute a *validation error*, denoted by $E_{val}(g^-)$, which may be regarded as an estimate of the out-of-sample error of $g^-(\boldsymbol{x})$:

$$E_{val}(g^-) = \frac{1}{K} \sum_{\boldsymbol{x}_n \in \mathcal{D}_{val}} e\left(g^-(\boldsymbol{x}_n), y_n\right) \tag{3.27}$$

where is the pointwise error measure. In the case of classification, this error measure becomes $[\![g^-(\boldsymbol{x}_n) - y_n]\!]$, while it is $\left(g^-(\boldsymbol{x}_n) - y_n\right)^2$ for regression. The expected validation error over all possible $\mathcal{D}_{val}$ turns out to be equal to $E_{out}(g^-)$ because

$$E_{\mathcal{D}_{\text{val}}}\left[E_{\text{val}}(g^-)\right]=\frac{1}{K}\sum_{x_n\in\mathcal{D}_{\text{val}}}E_{\mathcal{D}_{\text{val}}}\left[e\left(g^-(x_n),y_n\right)\right]=\frac{1}{K}\sum_{x_n\in\mathcal{D}_{\text{val}}}E_{\text{out}}(g^-)=E_{\text{out}}(g^-) \tag{3.28}$$

Eq. (3.28) says that in average $E_{\text{val}}(g^-)$ is the same as the out-of-sample error at g^-. So how reliable is $E_{\text{val}}(g^-)$ in estimating $E_{\text{out}}(g^-)$ depends on the variance of $E_{\text{val}}(g^-)$. It can be shown that in a classification problem [8]

$$\text{var}\left[E_{\text{val}}(g^-)\right]\le\frac{1}{4K} \tag{3.29}$$

hence $E_{\text{val}}(g^-)$ as an estimate of $E_{\text{out}}(g^-)$ becomes more accurate as the size of the validation set gets larger. Another way to see the usefulness of $E_{\text{val}}(g^-)$ in estimating $E_{\text{out}}(g^-)$ is through the VC bound when it is applied to the scenario where $\mathcal{D}_{\text{val}}$ acts as the training set and the hypothesis set contains only a member g^-, thus $E_{\text{val}}(g^-)$ is equal to $E_{\text{in}}(g^-)$ and the VC bound gives

$$E_{\text{out}}(g^-)\le E_{\text{val}}(g^-)+O\left(\frac{1}{\sqrt{K}}\right) \tag{3.30}$$

From both (3.29) and (3.30), we see that the size K of the validation set has to be large for $E_{\text{val}}(g^-)$ to be a reliable estimate of $E_{\text{out}}(g^-)$. On the other hand, K has to be small such that the size $N-K$ of the training set is big enough for hypothesis g^- to offer a decent generalization performance. A rule of thumb is to set 20% of the data (i.e. $K=N/5$) for validation [8].

We now conclude this sub-section with a remark on "restoring" the data set $\mathcal{D}$. Here the restoration is the sense that once the validation set has been used in the estimation of E_{out} of hypothesis g^-, one does not have to take g^- as the final hypothesis. As long as $E_{\text{out}}(g^-)$ is considered acceptable, an improved (and final) hypothesis g can be found by learning that uses the entire data set $\mathcal{D}$ instead of using only a part of it with $N-K$ samples. From above analysis and (3.30), it follows that

$$E_{\text{out}}(g)\le E_{\text{out}}(g^-)\le E_{\text{val}}(g^-)+O\left(\frac{1}{\sqrt{K}}\right) \tag{3.31}$$

In summary, one purpose of utilizing a validation set is to guide the learning process and it is in this regard that a validation set differs from a test set [8].

B. ***Model Selection by Validation***

An important use of validation is for model selection. Here the term model selection refers to a variety of things including the selection of linearity/nonlinearity of the learning model, complexity of the model perhaps in terms of order of the polynomials involved, value of regularization parameter, etc. A point to stress is that it is validation's ability to estimate the out-of-sample error -- not only for one model but for a given number of models that makes validation a popular approach for model selection.

For description simplicity, suppose a certain learning algorithm has been chosen so that model difference can be characterized in terms of hypothesis set. Let us now consider the problem of selecting a model from M models represented by M hypothesis sets $\mathcal{H}_1$, $\mathcal{H}_2$, …, $\mathcal{H}_M$. In what

follows we use a common data set $\mathcal{D}$ that is partitioned into a training set $\mathcal{D}_{\text{train}}$ and a validation set $\mathcal{D}_{\text{val}}$. For each hypothesis set, say $\mathcal{H}_m$, training set $\mathcal{D}_{\text{train}}$ is used to learn a final hypothesis g_m^- for which validation set $\mathcal{D}_{\text{val}}$ is used to evaluate error $E_m = E_{\text{val}}(g_m^-)$, which estimates the out-of-sample error $E_{\text{out}}(g_m^-)$. If m^* is the index such that E_{m*} achieves the minimum of $\{E_m, m = 1, 2, \ldots, M\}$, then based on the validation errors model $\mathcal{H}_{m*}$ is selected. To estimate $E_{\text{out}}(g_{m*}^-)$, we consider a *new* learning problem with data set $\mathcal{D}_{\text{val}}$ and hypothesis set, $\mathcal{H}_{\text{val}}$, that contains M hypotheses: $\mathcal{H}_{\text{val}} = \{g_1^-, g_2^-, \cdots, g_M^-\}$ where each g_m^- was obtained above. Note that $E_{\text{val}}(g_{m*}^-)$ is within the hypothesis set $\mathcal{H}_{\text{val}}$ and $E_{\text{val}}(g_{m*}^-)$ can be viewed as "in-sample" error for $E_{\text{val}}(g_{m*}^-)$. Applying the VC bound in this case leads to

$$E_{\text{out}}(g_{m*}^-) \le E_{\text{val}}(g_{m*}^-) + O\left(\sqrt{\frac{\log M}{K}}\right) \tag{3.32}$$

Once the model $\mathcal{H}_{m*}$ is selected, as the last step of the learning process $\mathcal{H}_{m*}$ is utilized together with the *entire* data set $\mathcal{D}$ in the learning to identify a final hypothesis g_{m*}. From (3.32), it follows that

$$E_{\text{out}}(g_{m*}) \le E_{\text{out}}(g_{m*}^-) \le E_{\text{val}}(g_{m*}^-) + O\left(\sqrt{\frac{\log M}{K}}\right) \tag{3.33}$$

C. *Cross Validation*

From what described above we see that validation faces a dilemma when it comes to determine the size of validation set (K) versus the size of training set ($N - K$). In the extreme case of $K = 1$, the "final" hypothesis g^- obtained based on $N - 1$ data points and the final hypothesis g one would get based on the entire data set of N points are expected to be practically the same, consequently the difference between $E_{\text{out}}(g^-)$ and $E_{\text{out}}(g)$ is minimal. One the other hand, estimation of $E_{\text{out}}(g^-)$ as seen in (3.30) becomes unreliable as term $E_{\text{val}}(g^-)$ is obtained using only one data point and with $K = 1$ term $O\left(\frac{1}{\sqrt{K}}\right)$ becomes fairly large.

Cross validation (CV) is an effective approach to resolving the dilemma. In an M-fold CV [3], the data is split into M (approximately) equal-sized parts. One then carries out validation M times, each time using the mth data part as the validation set and the remaining data as the training set, for $m = 1, 2, \ldots, M$. This yields a total of M "final" hypotheses $\{g_1^-, g_2^-, \cdots, g_M^-\}$. Let $E_{\text{val}}(g_m^-)$ be the validation error obtained by evaluating the error of g_m^- over the mth data part (see (3.27)), the *CV estimate* is defined as the average value of $\{E_{\text{val}}(g_m^-), m = 1, 2, \ldots, M\}$, namely

$$E_{\text{cv}} = \frac{1}{M}\sum_{m=1}^{M} E_{\text{val}}(g_m^-) \tag{3.34}$$

In the particular case of $M = N$, a data set of size N is partitioned into N data parts, i.e. each part contains a single data point, the M-fold CV is reduced to the *leave-one-out* cross validation

(LOO-CV). Eq. (3.34) in this case becomes

$$E_{\text{cv}} = \frac{1}{N}\sum_{n=1}^{N} e_n, \quad e_n = e\left(g_n^-(\boldsymbol{x}_n), y_n\right) \tag{3.35}$$

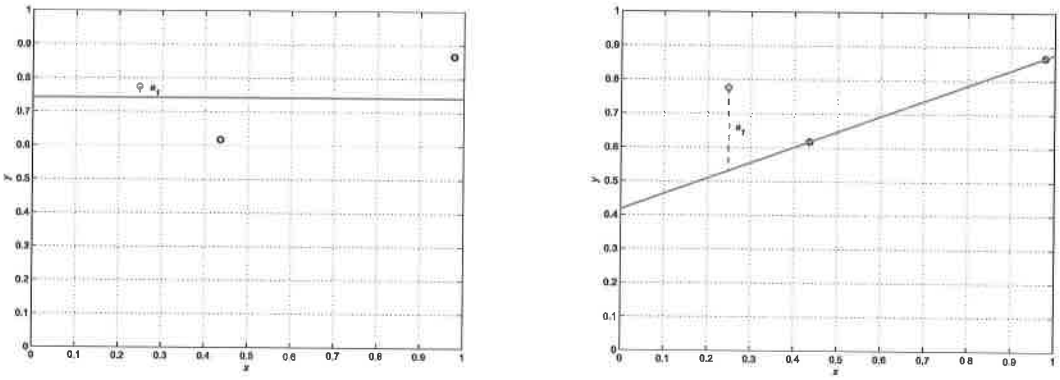
It can be shown [8] that for LOO-CV the expected E_{cv} with respect to all size-N data sets is the same as the expected $E_{\text{out}}(g)$ with respect to all size-(N–1) data sets, i.e., $\mathcal{E}_{\mathcal{D}_N}[E_{\text{cv}}] = \mathcal{E}_{\mathcal{D}_{N-1}}[E_{\text{out}}(g)]$. In other words, E_{cv} is an *unbiased* estimate of $\mathcal{E}_{\mathcal{D}_{N-1}}[E_{\text{out}}(g)]$. In practice the reliability of using E_{cv} to predict $E_{\text{out}}[g]$ has been found pretty high although little can be said theoretically.

Another use of CV is for model selection. Model selection was addressed in Sec. 3.3.B by validation. Here we approach the same problem via cross validation. Consider selecting a model from M models represented by M hypothesis sets $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_M$. For each hypothesis set, say $\mathcal{H}_m$, cross validation is carried out to yield $E_{\text{cv}}^{(m)}$. Let m^* be the index with $E_{\text{cv}}^{(m^*)}$ reaching the minimum of $\{E_{\text{cv}}^{(m)}, m = 1, 2, \ldots, M\}$, then model $\mathcal{H}_{m^*}$ is selected. A final hypothesis g will then be identified from $\mathcal{H}_{m^*}$ using the entire data set, whose out-of-sample error $E_{\text{out}}(g)$ may be estimated by $E_{\text{cv}}^{(m^*)}$.

Example 3.5 [8]

Suppose we have a data set of three points $\mathcal{D} = \{(x_i, y_i), i = 1, 2, 3\}$ that are produced from a target function over [0, 1] that equals a constant c plus Gaussian noise with zero mean and variance σ^2 with $\sigma = 0.1$. We consider two models: $\mathcal{H}_1$:$\{h(x) = ax + b\}$ and $\mathcal{H}_2$: $\{h(x) = b\}$ where a and b are real-valued parameters. We apply cross validation to both models, the CV errors for the linear and constant models are denoted by $E_{\text{cv}}^{(L)}$ and $E_{\text{cv}}^{(C)}$, respectively. An example situation is illustrated in Fig. 3.16. Fig. 3.16(a) shows error term e_1 when CV was applied to the constant model and the first data pair (x_1, y_1) was taken out. Fig. 3.16(b) shows error term e_1 when CV was applied to the linear model and the first data pair (x_1, y_1) was taken out. It was found that $E_{\text{cv}}^{(C)} = 0.0238$ and $E_{\text{cv}}^{(L)} = 0.3278$. Because $E_{\text{cv}}^{(C)} < E_{\text{cv}}^{(L)}$, the cross validation helps make the right decision of selecting the constant model.

Yet another use of cross validation is for selecting regularization parameter λ. This is not surprising because λ is considered as a part of a learning model as it is involved in both a regularized objective function in a way that effectively imposes a constraint on "how much" of the hypothesis set can actually be used in searching a final hypothesis $g(\boldsymbol{x})$. Let $\{\lambda_m, m = 1, 2, \ldots, M\}$ be M values for the regularization parameter over a range that contains the optimal λ^*. We consider selecting a model from M models represented by $(\mathcal{H}, \lambda_1)$, $(\mathcal{H}, \lambda_2)$, …, $(\mathcal{H}, \lambda_M)$. For each hypothesis set, say $(\mathcal{H}, \lambda_m)$, cross validation is carried out to yield $E_{\text{cv}}^{(m)}$. Let m^* be the index



(a) (b)

Fig. 3.16 Instants of leave-one-out CV errors for (a) constant and (b) linear models.

with $E_{\text{cv}}^{(m^*)}$ reaching the minimum of $\{E_{\text{cv}}^{(m)}, m = 1, 2, \ldots, M\}$, then model $(\mathcal{H}, \lambda_{m^*})$ is selected. A final hypothesis g will then be identified from $(\mathcal{H}, \lambda_{m^*})$ using the entire data set, whose out-of-sample error $E_{\text{out}}(g)$ may be estimated by $E_{\text{cv}}^{(m^*)}$.

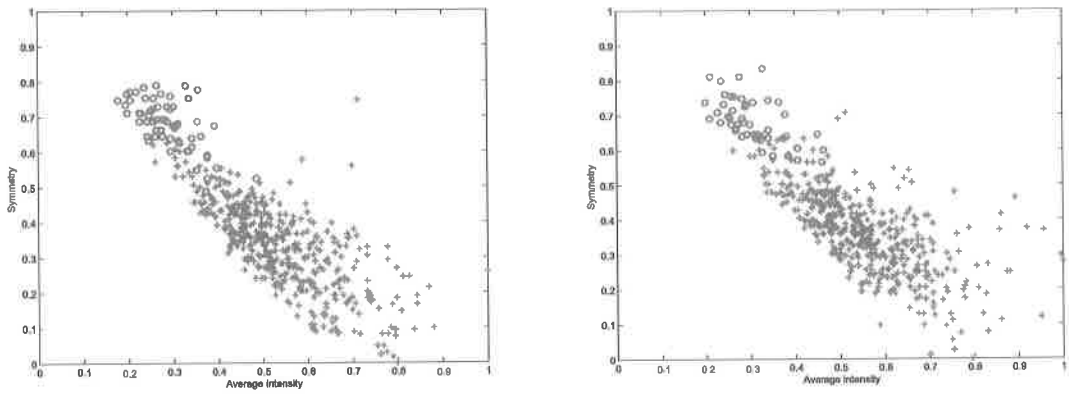
Example 3.6

This example illustrates the use of LOO-CV in estimating E_{out} in connection with model selection. Here we consider the problem of classifying a handwritten digit as either "1" or else. The *raw* training data consists of an array of size 784 by 500 representing 500 handwritten digits that were selected at random from MNIST database together with their labels. It has been noted that digit 1 has smaller average intensity relative to other digits and possesses symmetry measured by the difference between image "1" and the one by flipping it from left to right then from upside down will be small relative to the differences associated with other digits [8]. Based on these observations a 2-dimensional feature vector can be generated for each sample in the training set thus producing a training set of size 2 by 500. Fig. 3.17a shows the samples of digit 1 from the training data (marked as circles in blue) and the samples of all other digits marked as pluses in red in the input space with the average intensity as horizontal axes and "one-minus-(normalized) symmetry" as vertical axis.

It is clear that the problem at hand is a classification problem and, from Fig. 3.17 we see the classification can be done reasonably well by a linear method working with a nonlinear dilated feature space $\mathcal{Z}$. In this example we examine a variety of models by gradually increasing the number of features from the simplest case of one feature i.e. $\{1, x_1, x_2\}$ to twenty features, i.e.

$$\{1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, \cdots, x_1^5, x_1^4x_2, x_1^3x_2^2, x_1^2x_2^3, x_1x_2^4, x_2^5\}$$

The method of logistic regression with BFGS-BT was chosen for the classification problem. For each chosen number of features, classification was carried out and E_{in}, E_{cv} for LOO-CV, and E_{out} were evaluated, where E_{out} was calculated using a set testing data of 500 samples outside the training set, depicted in Fig. 3.17b, that were selected at random from MNIST database consisting 50 "1"s and 450 other digits.



(a) (b)

Figure 3.17 (a) training data and (b) testing data for Example 3.6.

The profiles of E_{in}, E_{cv}, and E_{out} versus the number of feature used are shown in Fig. 3.18. It is

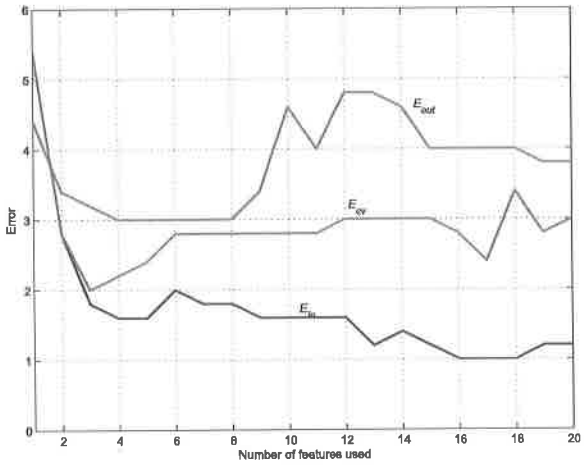


Figure 3.18 E_{in}, E_{cv}, and E_{out} versus the number of feature used for Example 3.6.

observed that the smallest E_{cv} values occurred when models using the first three or four features were employed and it was these models whose E_{out} were among the smallest. In this regard, a profile E_{cv}'s values versus model complexity is of help in selecting a right model for relative low E_{out}. It is also worth noticing that models with low E_{in} but high E_{cv} are not necessarily good ones as high E_{cv}'s usually indicate high E_{out}. ■

MATLAB Code Used in This Chapter

(1) coeff_legendre.m

```
% Generate the coefficients of Legendre polynomials pk(x) for k = 0, 1, ..., n. It is well known
% that p0(x) = 1 and p1(x) = x. The input of the MATLAB code is for n >= 2.
% The output is a matrix of size (n+1) by (n+1) whose (k+1)th row lists the coefficients of pk(x)
% in descending order.
% Written by W.-S. Lu, University of Victoria. Last modified: Feb. 11, 2015.
function P = coeff_legendre(n)
P = zeros(n+1,n+1);
P(1,n+1) = 1;
P(2,n) = 1;
for i = 1:(n-1),
   p1 = P(i+1,:);
   p1new = [p1(2:n+1) 0];
   p2 = P(i,:);
   r1 = (2*i+1)/(i+1);
   r2 = i/(i+1);
   P(i+2,:) = r1*p1new - r2*p2;
end
```

(2) ex3_1b.m

```
% To conduct the first set of simulations described in the case studies in Sec. 3.1.
% Input:
% N: number of sample points.
% st1: initial state for sample points.
% st2: initial state for weights that combine Q + 1 Legendre polynomials in
%      order to generate the target poly of order Q (which is set to 10 here).
% st3: initial state for noise.
% K: number of experiments for each case with fixed noise level.
% Output:
% E2: Average out-of-sample error for 2nd-order fit as function of noise level.
% E10: Average out-of-sample error for 10th-order fit as function of noise level.
% U: Overfit measure by E10 - E2.
% Written by W.-S. Lu, University of Victoria.  Last modified: Feb. 14, 2015.
% Example: [U,E2,E10] = ex3_1b(15,148,9,18,2000);
function [U,E2,E10] = ex3_1b(N,st1,st2,st3,K)
Q = 10;                    % Order of target polynomial.
P = coeff_legendre(Q);
xt = -1:2/1000:1;
sig = 0:0.025:0.5;
Ls = length(sig);
E2 = zeros(Ls,1);
E10 = E2;
rand('state',st1),
x = sort(2*(rand(1,N)-0.5));
for i = 1:Ls,
   sigi = sig(i);
   eg2 = 0;
```

```
  eg10 = 0;
  for k = 1:K,
     randn('state',st2+k-1)
     ck = randn(1,Q+1);
     pk = ck*P;
     ytk = polyval(pk,xt);
     mk = mean(ytk.^2);
     pk = pk/sqrt(mk);
     yk = polyval(pk,x);
     ytk = polyval(pk,xt);
     randn('state',st3+k-1)
     ykn = yk + sigi*randn(1,N);
     randn('state',st3+k-1)
     ytkn = ytk + sigi*randn(1,1001);
     p2 = polyfit(x,ykn,2);
     ztk2 = polyval(p2,xt);
     p10 = polyfit(x,ykn,10);
     ztk10 = polyval(p10,xt);
     eg2 = eg2 + (norm(ytkn-ztk2)^2)/1001;
     eg10 = eg10 + (norm(ytkn-ztk10)^2)/1001;
   end
   E2(i) = eg2/K;
   E10(i) = eg10/K;
end
U = E10 - E2;
figure(1)
plot(sig,U,'b-','linewidth',1.5)
xlabel('Standard deviation \sigma')
ylabel('Overfit')
grid
```

(3) ex3_1a_re.m

```
% Regularized least-squares polynomial fitting.
% Input:
% N: Data size.
% Q: Order of the target polynomial.
% M: Order of the fitting polynomial.
% lam: value of regularization parameter lambada.
% sig: standard deviation of the noise.
% st1: initial state for data samples.
% st2: initial state for target polynomial.
% st3: initial state for noise.
% Output:
% ws: Coefficients of f fitting polynomial.
% p: Coeff. of target polynomial.
% (x,y): Training data without noise.
% (x,yn): Training data with noise.
% Ein: In-sample error.
% Eout: Out-of-sample error.
% Written by W.-S. Lu, University of Victoria.
```

```
% Last modified: Feb. 20, 2015.
% Example:
% [ws,p,x,y,yn,Ein,Eout] = ex3_1a_re(15,10,10,0.015,0.3,148,119,18);
function [p,x,y,yn,Ein,Eout] = ex3_1a_re(N,Q,M,lam,sig,st1,st2,st3)
rand('state',st1)
x = sort(2*(rand(N,1)-0.5));
randn('state',st2)
p = 5*randn(1,Q+1);
y = polyval(p,x);
randn('state',st3)
w = sig*randn(N,1);
yn = y + w;
V = fliplr(vander(x));
X = V(:,1:(M+1));
I = eye(M+1,M+1);
w = (inv(X'*X + lam*I))*(X'*yn);
ws = flipud(w);
zn = polyval(ws,x);
xt = -0.95:1.8/1000:0.85;
yt = polyval(p,xt);
zt = polyval(ws,xt);
figure(1)
plot(xt,yt,'k-','linewidth',1.5)
hold on
plot(x,yn,'bo','linewidth',1.5)
plot(xt,zt,'r-','linewidth',1.5')
grid
axis([-0.95 0.85 -2 2])
xlabel('\itx')
ylabel('\ity')
hold off
Ein = (norm(yn-zn)^2)/N;
randn('state',st3)
wn = sig*randn(1,1001);
ytn = yt + wn;
Eout = (norm(ytn-zt)^2)/1001;
disp('in-sample error and out-of-sample error:')
[Ein Eout]
```

(4) ex3_1a_re1a

```
% L1-regularized least-squares polynomial fitting.
% Input:
% N: Data size.
% Q: Order of the target polynomial.
% M: Order of the fitting polynomial.
% lam: value of regularization parameter lambada.
% sig: standard deviation of the noise.
% st1: initial state for data samples.
% st2: initial state for target polynomial.
% st3: initial state for noise.
% K: number of iterations (soft shrinkage).
```

```
% Output:
% ws: Coeff. of fitting polynomial.
% Ein: In-sample error.
% Eout: Out-of-sample error.
% Written by W.-S. Lu, University of Victoria.  Last modified: March 16, 2015.
% Example:
% [ws1,Ein1,Eout1] = ex3_1a_re1a(15,10,10,0.03,0.3,148,119,18,10000);
function [ws,Ein,Eout] = ex3_1a_re1a(N,Q,M,lam,sig,st1,st2,st3,K)
rand('state',st1),
x = sort(2*(rand(N,1)-0.5));
randn('state',st2)
p = 5*randn(1,Q+1);
y = polyval(p,x);
randn('state',st3)
w = sig*randn(N,1);
yn = y + w;
V = fliplr(vander(x));
X = V(:,1:(M+1));
xx = X'*X;
L = max(eig(xx));
yh = (1/L)*X'*yn;
Xh = (1/L)*xx;
Lh = lam/L;
wk = zeros(M+1,1);
k = 0;
while k < K,
   bk = yh - Xh*wk + wk;
   abk = abs(bk) - Lh;
   wk = sign(bk).*(max(abk,0));
   k = k + 1;
end
ws = flipud(wk);
zn = polyval(ws,x);
xt = -0.95:1.8/1000:0.85;
yt = polyval(p,xt);
zt = polyval(ws,xt);
figure(1)
plot(xt,yt,'k-','linewidth',1.5)
hold on
plot(x,yn,'bo','linewidth',1.5)
plot(xt,zt,'r-','linewidth',1.5')
grid
axis([-0.95 0.85 -2 2])
xlabel('\itx')
ylabel('\ity')
hold off
Ein = (norm(yn-zn)^2)/N;
randn('state',st3)
wn = sig*randn(1,1001);
ytn = yt + wn;
Eout = (norm(ytn-zt)^2)/1001;
```

```
disp('in-sample error and out-of-sample error:')
[Ein Eout]
```

Problems

3.1 Polynomial models may be considered as a special family of linear models in dilated feature space $\mathcal{Z}$ under nonlinear transformation $z = \Phi(x)$. So far conventional polynomials have been utilized in nonlinear transformations. Here we consider a linear model in space $\mathcal{Z}$ where Legendre polynomials up to order Q are utilized instead of a collection of monomials. Assume that there is only one basic feature, x, in the input space $\mathcal{X}$. Let

$$z = \begin{bmatrix} 1 \\ L_1(x) \\ \vdots \\ L_Q(x) \end{bmatrix}$$

The hypothesis set $\mathcal{H}$ is given by

$$\mathcal{H} = \left\{ h : h(x) = \boldsymbol{w}^T z = \sum_{q=0}^{Q} w_q L_q(x) \right\}$$

where $L_0(x) = 1$. For linear regression, one seeks to find a weight vector $\boldsymbol{w}^*$ that minimizes the L_2 in-sample error

$$E_{\text{in}}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} \left(\boldsymbol{w}^T z_n - y_n \right)^2$$

(i) Derive a closed-form formula for the global minimizer $\boldsymbol{w}^*$ of $E_{\text{in}}(\boldsymbol{w})$ where the Legendre polynomials involved are explicitly shown.
(ii) Now consider the case where the input space includes *two* basic features x_1 and x_2. To simplify the matter, you are asked to develop a closed-form formula for the minimizer $\boldsymbol{w}^*$ for $Q = 3$ by carrying out the following two steps.
<u>Step 1</u>: Recall in the single variable case the Legendre polynomials up to order Q are $L_0(x)$, $L_1(x)$, …, $L_Q(x)$. Write down all two-variable Legendre polynomials (in terms of single-variable Legendre polynomials) up to order $Q = 3$.
<u>Step 2</u>: Based on the outcome from Step 1, develop a closed-form formula for $\boldsymbol{w}^*$ where the single-variable Legendre polynomials are explicitly shown.

3.2 Recall that in Sec. 3.2.B.2 the modeling problem involving noisy data was addressed using an algorithm based on sequential minimization of proximal functions. Here you are asked to repeat the example (see pages 130 – 132) by using CVX instead of sequential minimization. This can be done by simply replacing the relevant part of the code ex3_1a_re1a.m with a much shorter block of CVX lines. Compare the two solutions in terms of E_{in} and E_{out} as well as solutions' sparsity.

3.3 Suppose we have a data set of three points $\mathcal{D} = \{(x_i, y_i), i = 1, 2, 3\}$ that are produced from a target function over [0, 1] that equals a linear function $f(x) = -0.6x + 0.8$ plus Gaussian noise with zero mean and variance σ^2 with $\sigma = 0.1$. The data set $\mathcal{D} = \{(x_i, y_i), i = 1, 2, 3\}$ is generated as follows:

```
rand('state',10),
```

```
x = rand(3,1)
randn('state',16)
w = 0.1*randn(3,1)
y0 = -0.6x + 0.8;
y = y0 + w;
```

We consider two models: $\mathcal{H}_0$: $\{h(x) = c\}$ and $\mathcal{H}_1$: $\{h(x) = ax + b\}$ where a, b, and c are real-valued parameters.

(i) Apply cross validation to both models and compute E_{cv} for the constant and linear models.

(ii) Based on the results from part (i), select a hypothesis.

Chapter 4 Support Vector Machines

The present form of support vector machines (SVMs) was largely developed by V. Vapnik and co-workers at AT&T Bell Labs in 1990's ([1], [2]) and has since been a subject of intensive studies. As a result, numerous variants of SVM are now available, with many of them demonstrating competitive performance in real-world applications. The material presented below is largely based on [18] and [11].

4.1 Support Vector Machines

A. Support Vector Machines for Classification

Given linearly separable training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)$ for $n = 1, \ldots, N\}$ with $\boldsymbol{x}_n$ belonging to an input space $\mathcal{X}$ of dimension d, there exists a decision boundary $\boldsymbol{w}^T\boldsymbol{x} = 0$ (where $\boldsymbol{x}$ assumes the form of $[1 \;\; x_1 \ldots x_d]^T$) that separates the points in $\mathcal{D}$ with $y_n = 1$ from those points with $y_n = -1$. Typically such separating weight vector is *not* unique, thus one may define a *margin* as the smallest distance between the decision boundary and the data points on both sides of the boundary. Among other things, a separating $\boldsymbol{w}$ with bigger margin offers more robust performance against data's uncertainty. A question naturally arises is how to find a separating $\boldsymbol{w}$ with largest margin? The SVM exactly addresses this question.

For clarity of description, in what follows we decide to denote the weight vector as

$$\begin{bmatrix} b \\ \boldsymbol{w} \end{bmatrix}$$

and $\boldsymbol{x} = [x_1 \ldots x_d]^T$ so that the decision boundary assumes the form

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0 \tag{4.1}$$

Let $\boldsymbol{x}_n$ be a sample point from $\mathcal{D}$ that is nearest to the decision boundary. Depending on which side of the boundary $\boldsymbol{x}_n$ resides, we have $\boldsymbol{w}^T\boldsymbol{x}_n + b = \pm\delta$ with $\delta > 0$, hence $\left|\boldsymbol{w}^T\boldsymbol{x}_n + b\right| = \delta$. By dividing both sides of the above by δ and renaming $\boldsymbol{w}/\delta$ and b/δ to $\boldsymbol{w}$ and $\boldsymbol{b}$ respectively, we have

$$\left|\boldsymbol{w}^T\boldsymbol{x}_n + b\right| = 1 \tag{4.2}$$

Since the same scaling can be applied to (4.1), the $\boldsymbol{w}$'s and b's in (4.1) and (4.2) are now considered to be the same.

To compute the distance between point $\boldsymbol{x}_n$ and the decision boundary, note that a separating w is orthogonal to the boundary. This is because for any two points $\boldsymbol{x}$ and $\boldsymbol{x}'$ on the boundary, (4.1) implies that $\boldsymbol{w}^T(\boldsymbol{x}' - \boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}' + b - (\boldsymbol{w}^T\boldsymbol{x} + b) = 0$. Therefore the distance we look for can be found by projecting vector $\boldsymbol{x}_n - \boldsymbol{x}$ on $\boldsymbol{w}$, namely,

$$\|\hat{\boldsymbol{w}}\| = \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}, \text{ the distance} = \left|\hat{\boldsymbol{w}}^T(\boldsymbol{x}_n - \boldsymbol{x})\right| = \frac{1}{\|\boldsymbol{w}\|}\left|\boldsymbol{w}^T\boldsymbol{x}_n - \boldsymbol{w}^T\boldsymbol{x}\right| = \frac{1}{\|\boldsymbol{w}\|}\left|\boldsymbol{w}^T\boldsymbol{x}_n + b - (\boldsymbol{w}^T\boldsymbol{x} + b)\right| = \frac{1}{\|\boldsymbol{w}\|}$$

Therefore, the separating $\boldsymbol{w}$ with largest margin can be obtained by solving the constrained problem

$$\begin{aligned} &\text{maximize} && \frac{1}{\|\boldsymbol{w}\|} \\ &\text{subject to:} && \min_{1\le n\le N} |\boldsymbol{w}^T\boldsymbol{x}_n + b| = 1 \end{aligned}$$

which is equivalent to

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \\ &\text{subject to:} && y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \ge 1 \ \text{ for } n = 1,\ldots,N \end{aligned} \tag{4.3}$$

Problem (4.3) is a convex quadratic programming (QP) problem. By defining

$$\hat{\boldsymbol{w}} = \begin{bmatrix} b \\ \boldsymbol{w} \end{bmatrix}, \quad \boldsymbol{H} = \begin{bmatrix} 0 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_N \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} -y_1 & -y_1\boldsymbol{x}_1^T \\ \vdots & \vdots \\ -y_N & -y_N\boldsymbol{x}_N^T \end{bmatrix}, \quad \boldsymbol{e} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Problem (4.3) can be expressed in a standard form

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\hat{\boldsymbol{w}}^T\boldsymbol{H}\,\hat{\boldsymbol{w}} \\ &\text{subject to:} && \boldsymbol{A}\hat{\boldsymbol{w}} \le -\boldsymbol{e} \end{aligned} \tag{4.4}$$

The Lagrangian of (4.3) (see Sec. B.1.2 of Appendix B) is given by

$$L(\boldsymbol{w}, b, \boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \sum_{n=1}^{N}\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)) \tag{4.5}$$

where vector $\boldsymbol{\mu} = [\mu_1 \quad \mu_2 \quad \cdots \quad \mu_N]^T$ collects N Lagrange multipliers. The KKT conditions (see Sec. B.1.2.A of Appendix B) for the solution of (4.3) are the N constraints in (4.3) plus

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{n=1}^{N}\mu_n y_n \boldsymbol{x}_n = \boldsymbol{0} \tag{4.6}$$

$$\frac{\partial L}{\partial b} = -\sum_{n=1}^{N}\mu_n y_n = 0 \tag{4.7}$$

$$\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)) = 0 \quad \text{for } n = 1, \ldots, N \tag{4.8}$$

$$\mu_n \ge 0 \quad \text{for } n = 1, \ldots, N \tag{4.9}$$

Recall the Lagrange dual (see Sec. B.1.2.D of Appendix B) of problem (4.3) is given by

$$\begin{aligned} &\text{maximize} && q(\boldsymbol{\mu}) = \inf_{\boldsymbol{w},b} L(\boldsymbol{w}, b, \boldsymbol{\mu}) \\ &\text{subject to:} && \boldsymbol{\mu} \ge 0 \end{aligned} \tag{4.10}$$

It follows from (4.5) – (4.7) that $q(\boldsymbol{\mu}) = -\frac{1}{2}\boldsymbol{\mu}^T(\boldsymbol{d}\cdot\boldsymbol{d}^T)\boldsymbol{\mu} + \boldsymbol{e}^T\boldsymbol{\mu}$ subject to $\sum_{n=1}^{N}\mu_n y_n = 0$, where $\boldsymbol{d} = [y_1\boldsymbol{x}_1 \quad y_2\boldsymbol{x}_2 \quad \cdots \quad y_N\boldsymbol{x}_N]^T$. This in conjunction with the positivity constraint on $\boldsymbol{\mu}$ leads (4.10) to an explicit convex QP dual problem as

$$\underset{\boldsymbol{\mu}}{\text{minimize}} \quad \frac{1}{2}\boldsymbol{\mu}^T(\boldsymbol{d}\cdot\boldsymbol{d}^T)\boldsymbol{\mu} - \boldsymbol{e}^T\boldsymbol{\mu} \tag{4.11}$$
$$\text{subject to:} \quad \boldsymbol{\mu}^T\boldsymbol{y} = 0, \quad \boldsymbol{\mu} \geq \boldsymbol{0}$$

where $\boldsymbol{y} = [y_1\ y_2\ \ldots\ y_N]^T$. The solution $\boldsymbol{\mu}$ of the dual problem (4.11) is connected to the solution $(\boldsymbol{w}, b)$ of the primal problem (4.3) via (4.6) as

$$\boldsymbol{w} = \sum_{n=1}^{N} \mu_n y_n \boldsymbol{x}_n \tag{4.12}$$

then via any equation in (4.8) with $\mu_m \neq 0$ as

$$b = y_m - \boldsymbol{w}^T\boldsymbol{x}_m \tag{4.13}$$

Support Vectors

If we let $S \subseteq \{1,2,\ldots,N\}$ be the set of indices with which $\mu_n \neq 0$, (4.12) can be reduced to

$$\boldsymbol{w} = \sum_{n\in S} \mu_n y_n \boldsymbol{x}_n \tag{4.14}$$

From the complementarity conditions in (4.8), we see that $\mu_n \neq 0$ implies $|\boldsymbol{w}^T\boldsymbol{x}_n + b| = 1$ which means that the distance from point $\boldsymbol{x}_n$ to the decision boundary is exactly equal to the margin. Such a vector $\boldsymbol{x}_n$ bears the name of *support vector* (SV). From (4.14) it follows that the solution $\boldsymbol{w}$ of problem (4.3) is merely a linear combination of SVs, hence the name of support vector machine (SVM). For a given data set of size N, the number of SVs is usually much smaller than N, formula (4.14) expresses the optimal separating weight $\boldsymbol{w}$ as a *sparse* linear representation of the data set. It is worthwhile to mention [18] that there exists an "in-sample" check of the expected out-of-sample error as

$$E[E_{\text{out}}] \leq \frac{E[\text{number of SV's}]}{N-1} \tag{4.15}$$

Example 4.1 Apply linear SVM to classify the same data as that given in Example 1.3.

Solution

The linear SVM was applied to the data set by solving the dual problem (4.11) via CVX. Among 15 Lagrange multipliers, there are three nonzero ones. The three support vectors that are associated with these nonzero Lagrange multipliers are found to be

3.1000 8.4000 10.5000
4.9000 2.6000 0.8000

Using (4.14), the optimal $\boldsymbol{w}$ was found as a linear combination of these SV to be

w = $[1.740968869612935\ \ 3.142234305706753]^T$

and using (4.12) b was obtained as

b = -21.793951593763186

Each SV is marginal with a distance $1/\|\boldsymbol{w}\| = 0.2784$ to the decision boundary, see Fig. 4.1 where the SVs are marked by blue and red dots. The MATLAB function implementing the SVM for Example 4.1 is svm_linear.m which is enclosed at the end of this chapter.

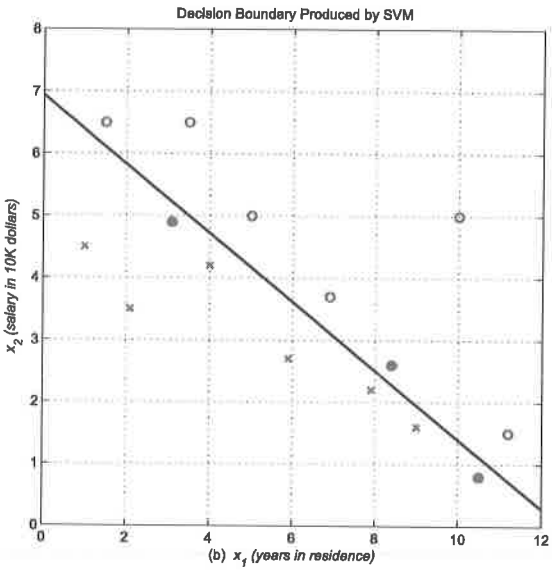


Figure 4.1 Decision boundary and three support vectors for Example 4.1.

B. Support Vector Machine with Nonlinear Transformation

In this section, we describe SVMs as applied to a dilated feature space Z using *nonlinear* combinations of the individual features x_i's. In spirit, this bears an analogy to what we did in Sec. 2.4 where linear models are extended from the basic feature space X to nonlinearly dilated feature space Z.

We consider a transform that maps a feature vector $\boldsymbol{x} \in R^d$ to a nonlinearly dilated feature vector $\boldsymbol{z} \in R^{\tilde{d}}$ by

$$\boldsymbol{z} = \Phi(\boldsymbol{x}) \tag{4.16}$$

For a 2-dimensional input space R^2 for example, the general 2nd-order, 3rd-order, and 5th-order dilations assume the form

$$\Phi_2(\boldsymbol{x}) = \begin{bmatrix} x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \end{bmatrix}^T$$

$$\Phi_3(\boldsymbol{x}) = \begin{bmatrix} x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & x_1^3 & x_1^2x_2 & x_1x_2^2 & x_2^3 \end{bmatrix}^T$$

and

$$\Phi_5(\boldsymbol{x}) = [x_1\ x_2\ x_1^2\ x_1x_2\ x_2^2\ x_1^3\ x_1^2x_2\ \cdots\ x_1^5\ x_1^4x_2\ x_1^3x_2^2\ x_1^2x_2^3\ x_1x_2^4\ x_2^5]^T$$

respectively. Notice the subtle difference between what we did here and in Sec. 1.4 – the component "1" in $\Phi(\boldsymbol{x})$ has been removed because the weight vector is now denoted to separate the "intersect" b from the rest part of the weight.

Working in space Z, the decision boundary as a counterpart of (4.1) becomes

$$\boldsymbol{w}^T\boldsymbol{z} + b = 0 \tag{4.17}$$

where $\boldsymbol{w} \in R^{\tilde{d}}$. Since the decision boundary in space Z remains linear as seen in (4.16), the SVM

can be obtained by solving the convex QP problem

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \\ &\text{subject to:} && y_n(\boldsymbol{w}^T\boldsymbol{z}_n + b) \geq 1 \quad \text{for} \quad n = 1,\ldots,N \end{aligned} \tag{4.18}$$

where $\boldsymbol{z}_n$ are generated by $\boldsymbol{z}_n = \Phi(\boldsymbol{x}_n)$ for n = 1, …, N. Like the case in Sec. 4.1.A, the primal problem (4.18) may be tackled by solving its Lagrange dual problem

$$\begin{aligned} &\underset{\boldsymbol{\mu}}{\text{minimize}} && \frac{1}{2}\boldsymbol{\mu}^T(\tilde{\boldsymbol{d}}\cdot\tilde{\boldsymbol{d}}^T)\boldsymbol{\mu} - \boldsymbol{e}^T\boldsymbol{\mu} \\ &\text{subject to:} && \boldsymbol{\mu}^T\boldsymbol{y} = 0, \quad \boldsymbol{\mu} \geq \boldsymbol{0} \end{aligned} \tag{4.19}$$

where $\tilde{\boldsymbol{d}} = [y_1\boldsymbol{z}_1 \quad y_2\boldsymbol{z}_2 \quad \cdots \quad y_N\boldsymbol{z}_N]^T$. The KKT conditions for the solution of (4.18) are the N constraints in (4.18) plus

$$\boldsymbol{w} - \sum_{n=1}^{N}\mu_n y_n \boldsymbol{z}_n = \boldsymbol{0} \tag{4.20}$$

$$-\sum_{n=1}^{N}\mu_n y_n = 0 \tag{4.21}$$

$$\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{z}_n + b)) = 0 \quad \text{for } n = 1, \ldots, N \tag{4.22}$$

$$\mu_n \geq 0 \quad \text{for } n = 1, \ldots, N \tag{4.23}$$

The solution $\boldsymbol{\mu}$ of the dual problem (4.19) is connected to the solution ($\boldsymbol{w}$, b) of the primal problem (4.18) via (4.20) as

$$\boldsymbol{w} = \sum_{n=1}^{N}\mu_n y_n \boldsymbol{z}_n \tag{4.24}$$

then via any of the complementarity conditions in (4.22) with $\mu_m \neq 0$ as

$$b = y_m - \boldsymbol{w}^T\boldsymbol{z}_m \tag{4.25}$$

If we let $S \subseteq \{1,2,\ldots,N\}$ be the set of indices with which $\mu_n \neq 0$, (4.24) can be reduced to

$$\boldsymbol{w} = \sum_{n\in S}\mu_n y_n \boldsymbol{z}_n \tag{4.26}$$

From the complementarity conditions in (4.22), we see that the vectors $\boldsymbol{z}_n$ that present in (4.26) satisfy $|\boldsymbol{w}^T\boldsymbol{z}_n + b| = 1$ which means that the distance from $\boldsymbol{z}_n$ to the decision boundary is exactly equal to the margin, hence the $\boldsymbol{z}_n$'s in (4.26) are *support vectors*. In words, for an SVM with a certain nonlinear transformation $\boldsymbol{z} = \Phi(\boldsymbol{x})$, the data set $\mathcal{D}$ = {($\boldsymbol{x}_n$, y_n) for n = 1, …, N} is transformed into space $\mathcal{Z}$ as $\tilde{\mathcal{D}}$ = {($\boldsymbol{z}_n$, y_n) for n = 1, …, N}, and the nonzero (optimized) Lagrange multipliers {μ_n, for $n \in S$} in that space help identify support vectors {$\boldsymbol{z}_n$, for $n \in S$}. And it is in space $\mathcal{Z}$ that formula (4.26) gives the optimal separating weight $\boldsymbol{w}$ as a *sparse* linear representation of $\tilde{\mathcal{D}}$ = {($\boldsymbol{z}_n$, y_n) for n = 1, …, N}.

Once an optimal weight $\boldsymbol{w}$ is found (in space $\mathcal{Z}$), a *nonlinear* decision boundary in space $\mathcal{X}$ can be obtained by combining (4.17) with (4.16) as

$$w^T\Phi(x)+b=0 \tag{4.27}$$

Example 4.2 Apply SVM with nonlinear transformation to classify the linearly non-separable "double semi-circle", a data set utilized in Chapter 2. Here the training data is the same as that generated in Example 3.1 with $N = 2000$.

Solution

For SVM with nonlinear transformation, the dimension of dual variable μ in dual problem (4.19) is N while the dimension of (w, b) in primal problem (4.18) is $\tilde{d}+1$ which typically is much less than N = 2000. The number of constraints in the primal and dual problems are practically the same (being N + 1 and N, respectively). Since The SVM with third-order nonlinear transform $z=\Phi_3(x)$ was applied to the data set by solving the primal problem (4.18).

Optimal w was found to be

-0.006919943508105
0.000017123243147
-0.037769011917928
-0.003294822314068
0.000891025633946
0.001678288960348
-0.000156410899954
0.000379761282649
-0.003379784198872

and optimal b was obtained as b = 1.232073071853626, which defines the decision boundary

$$w^T\Phi_3(x)+b=0$$

The support vectors in this case reside in 9-dimensional space Z and can be identified as those zn that satisfy $y_n(w^T z_n+b)=1$. In this way, seven SVs were detected whose counterparts in the basic feature space X were found to be

10.2288	2.4884	3.3034	21.2379	9.6739	-2.4578	22.5842
0.1156	0.7152	-2.9492	-3.8696	2.6056	0.8448	0.9720

The decision boundary together with the training data is depicted in Fig. 4.2. We stress that by definition the vectors given above are *not* SV's, however inspecting the figure indicates that the distance between each of these points to the decision boundary remains small although not guaranteed to be the smallest.

The MATLAB function implementing the SVM for Example 4.2 is svm_NL_semi_circle.m which is enclosed at the end of this chapter.

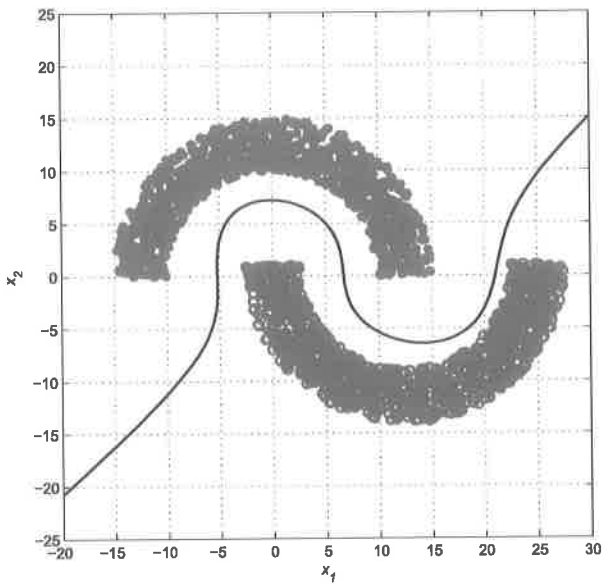


Figure 4.2 Decision boundary and training data for Example 4.2.

4.2 Kernel Trick and Soft-Margin Support Vector Machines

A. *The Kernel Trick*

Definition $K(\boldsymbol{x}, \boldsymbol{x}')$ is said to be a valid *kernel* function if it is symmetric and positive definite, i.e. $K(\boldsymbol{x}, \boldsymbol{x}') = K(\boldsymbol{x}', \boldsymbol{x})$, and the matrix

$$\begin{bmatrix} K(\boldsymbol{x}_1,\boldsymbol{x}_1) & K(\boldsymbol{x}_1,\boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_1,\boldsymbol{x}_N) \\ K(\boldsymbol{x}_2,\boldsymbol{x}_1) & K(\boldsymbol{x}_2,\boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_2,\boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ K(\boldsymbol{x}_N,\boldsymbol{x}_1) & K(\boldsymbol{x}_N,\boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_N,\boldsymbol{x}_N) \end{bmatrix} \tag{4.28}$$

is positive semidefinite for any $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, which is known as *Mercer's condition*.

Commonly used kernels include polynomial kernels of the form

$$K_Q(\boldsymbol{x},\boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}' + c)^Q \text{ with } c > 0 \text{ and } Q \text{ a positive integer} \tag{4.29}$$

and 'Gaussian' kernel

$$K_G(\boldsymbol{x},\boldsymbol{x}') = e^{-\|\boldsymbol{x}-\boldsymbol{x}'\|^2/2\sigma^2} \tag{4.30}$$

There are two reasons that motivate introduction and study of kernel functions. First, given a basic feature space $\mathcal{X}$, a kernel function induces a dilated feature space $\mathcal{Z}$ such that *computing an inner product in space $\mathcal{Z}$ becomes straightforward*. Consider a kernel $K(\boldsymbol{x},\boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}'+1)^2$ for a two-dimensional feature space $\mathcal{X} = R^2$ with $\boldsymbol{x} = [x_1\ x_2]^T$ and $\boldsymbol{x}' = [x_1'\ \ x_2']^T$. We can write

$$K_2(\boldsymbol{x},\boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}'+1)^2 = 1 + 2x_1x_1' + 2x_2x_2' + x_1^2x_1'^2 + 2x_1x_1'x_2x_2' + x_2^2x_2'^2 \tag{4.31}$$

If we consider a dilated feature space $\mathcal{Z}$ with

$$\boldsymbol{z} = \hat{\Phi}_2(\boldsymbol{x}) = \begin{bmatrix} 1 & \sqrt{2}x_1 & \sqrt{2}x_2 & x_1^2 & \sqrt{2}x_1x_2 & x_2^2 \end{bmatrix}$$

then we have

$$z^T z' = (x^T x' + 1)^2 = K_2(x, x')$$

Similarly, a Qth-order polynomial kernel $K_Q(x, x') = (x^T x' + 1)^Q$ induces a Qth-order feature space Z such that $z^T z' = (x^T x' + 1)^Q = K_Q(x, x')$. Furthermore, note that for a scalar feature space X we can write the Gaussian kernel with $\sigma = 1/\sqrt{2}$ as

$$K_G(x, x') = e^{-(x-x')^2} = e^{-x^2} e^{-x'^2} \sum_{k=0}^{\infty} \frac{2^k x^k x'^k}{k!} \tag{4.32}$$

hence a Gaussian kernel induces an infinite-dimensional feature space Z such that

$$z^T z' = e^{-\|x-x'\|^2/2\sigma^2} = K_G(x, x')$$

In summary, a valid kernel $K(x, x')$ induces a dilated feature space Z in which

$$z^T z' = K(x, x') \tag{4.33}$$

Note that using (4.33) to compute the inner product in space Z can be quite economical, especially when both the dimension of the basic feature space X and order of the nonlinear transformation Q are not too small.

The second reason we are interested in kernel functions is the role they play in many ML algorithms including SVMs. To be specific, we reexamine the dual problem for SVM with nonlinear transformation that is addressed in Sec. 4.1.B. In the objective function in (4.19), its quadratic term is given by $\frac{1}{2}\mu^T(\tilde{d} \cdot \tilde{d}^T)\mu$ with $\tilde{d} = [y_1 z_1 \quad y_2 z_2 \quad \cdots \quad y_N z_N]^T$, hence

$$\tilde{d} \cdot \tilde{d}^T = \begin{bmatrix} y_1 y_1 z_1^T z_1 & y_1 y_2 z_1^T z_2 & \cdots & y_1 y_N z_1^T z_N \\ y_2 y_1 z_2^T z_1 & y_2 y_2 z_2^T z_2 & \cdots & y_2 y_N z_2^T z_N \\ \vdots & \vdots & \vdots & \vdots \\ y_N y_1 z_N^T z_1 & y_N y_2 z_N^T z_2 & \cdots & y_N y_N z_N^T z_N \end{bmatrix}$$

Following (4.33), we have

$$\tilde{d} \cdot \tilde{d}^T = \begin{bmatrix} y_1 y_1 K(x_1, x_1) & y_1 y_2 K(x_1, x_2) & \cdots & y_1 y_N K(x_1, x_N) \\ y_2 y_1 K(x_2, x_1) & y_2 y_2 K(x_2, x_2) & \cdots & y_2 y_N K(x_2, x_N) \\ \vdots & \vdots & \vdots & \vdots \\ y_N y_1 K(x_N, x_1) & y_N y_2 K(x_N, x_2) & \cdots & y_N y_N K(x_N, x_N) \end{bmatrix} \tag{4.34}$$

Moreover, using (4.26), the decision boundary can be expressed as

$$w^T z + b = \sum_{n \in S} \mu_n y_n z_n^T z + b = \sum_{n \in S} \mu_n y_n K(x_n, x) + b = 0 \tag{4.35}$$

where b can be written as

$$b = y_m - w^T z_m = y_m - \sum_{n \in S} \mu_n y_n z_n^T z_m = y_m - \sum_{n \in S} \mu_n y_n K(x_n, x_m) \tag{4.36}$$

By means by (4.34)-(4.36), training SVMs in dual space becomes computationally more attractive.

Example 4.3 Apply SVM with a Gaussian kernel to classify the linearly non-separable "double semi-circle", a data set utilized in Chapter 3. The training data is the same as that generated in Example 2.1 with $N = 2000$.

Solution

As can be seen from (4.34), a kernel is inherently associated with the dual problem which in the present case involves $N = 2000$ variables as the components of $\boldsymbol{\mu}$. The dual problem in (4.19), where matrix $\tilde{\boldsymbol{d}} \cdot \tilde{\boldsymbol{d}}^T$ is given by (4.33) and $K(\boldsymbol{x}, \boldsymbol{x}') = K_G(\boldsymbol{x}, \boldsymbol{x}') = e^{-\|\boldsymbol{x}-\boldsymbol{x}'\|^2/2\sigma^2}$, was solved using CVX. The CPU time consumed by CVX was 158.15 seconds on a 64-bit desktop PC with Intel i7-3770 CPU@3.4GHz with 32GB RAM. It is of importance to note the role parameter σ in the Gaussian kernel (see (4.30)) plays – it controls the degree of closeness between the decision boundary and the "support vectors". Here the parameter was set to $\sigma = 0.8$ as it yielded a satisfactory decision boundary as seen in Fig. 4.3. Another thing to note is that the optimal weight vector $\boldsymbol{w}$ itself cannot be made available because $\boldsymbol{w}$ resides in space Z which in the case of Gaussian kernel is an infinite-dimensional space. Nevertheless, the decision boundary is explicitly defined in input space X, see (4.35), where b is computed by (4.36). The MATLAB function for Example 4.3 is svm_gaussian_semi_circle.m which is enclosed at the end of this chapter.

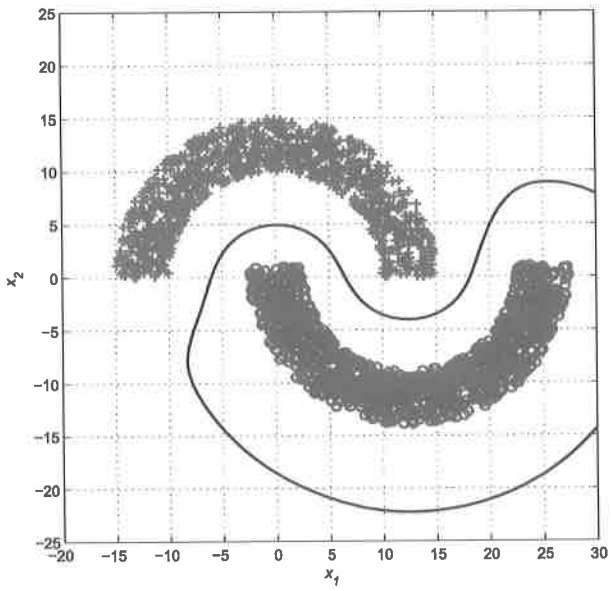


Figure 4.3 Decision boundary and training data for Example 4.3.

B. Soft-Margin Support Vector Machines

The standard SVM works for linearly separable data, but is not suited for non-separable data. A variant of SVM known as *soft-margin* SVM deal was developed to deal with non-separable data.

The soft-margin SVM modifies the standard SVM formulation (see (4.3)) in constraints as well as objective function, and the modifications are accomplished by introducing N nonnegative *slack variables* where N is the size of the training data. Specifically, the constraints in (4.3) are

relaxed to

$$y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - s_n \text{ with } s_n \geq 0 \text{ for } n = 1,\ldots,N$$

where slack variable s_n may be regarded as an amount of violation for data pair $(\boldsymbol{x}_n, y_n)$. Therefore, the "total violation" by all constraints can be measured by $\sum_{n=1}^{N} s_n$. To control the amount of total violation while maximizing the margin, the objective in (4.3) is modified by including a term $\tau\sum_{n=1}^{N} s_n$ with parameter $\tau > 0$ making a tradeoff between maximizing the margin and minimizing the total violation. Summarizing, the soft-margin SVM solves the convex QP problem

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \tau\sum_{n=1}^{N} s_n \\ &\text{subject to:} && y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - s_n \quad \text{for } n = 1,\ldots,N \\ & && s_n \geq 0 \quad \text{for } n = 1,\ldots,N \end{aligned} \tag{4.37}$$

The Lagrangian of (4.37) is given by

$$L(\boldsymbol{w}, b, \boldsymbol{s}, \boldsymbol{\mu}, \boldsymbol{\nu}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \tau\sum_{n=1}^{N} s_n + \sum_{n=1}^{N}\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - s_n) - \sum_{n=1}^{N}\nu_n s_n$$

and the KKT conditions constitute the constraints in (4.37) plus

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{n=1}^{N}\mu_n y_n \boldsymbol{x}_n = \boldsymbol{0} \tag{4.38}$$

$$\frac{\partial L}{\partial b} = -\sum_{n=1}^{N}\mu_n y_n = 0 \tag{4.39}$$

$$\frac{\partial L}{\partial s_n} = \tau - \mu_n - \nu_n = 0 \quad \text{for } n = 1, \ldots, N \tag{4.40}$$

$$\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - s_n) = 0 \quad \text{for } n = 1, \ldots, N \tag{4.41}$$

$$\nu_n \cdot s_n = 0 \quad \text{for } n = 1, \ldots, N \tag{4.42}$$

$$\mu_n \geq 0 \quad \text{for } n = 1, \ldots, N \tag{4.43}$$

$$\nu_n \geq 0 \quad \text{for } n = 1, \ldots, N \tag{4.44}$$

Using (4.38), (4.39), (4.40), and (4.44), the Lagrangian is reduced to a function of $\boldsymbol{\mu}$ as

$$L(\boldsymbol{\mu}) = -\frac{1}{2}\boldsymbol{\mu}^T(\boldsymbol{d}\cdot\boldsymbol{d}^T)\boldsymbol{\mu} + \boldsymbol{e}^T\boldsymbol{\mu} \tag{4.45}$$

where $\boldsymbol{d} = [y_1\boldsymbol{x}_1 \quad y_2\boldsymbol{x}_2 \quad \cdots \quad y_N\boldsymbol{x}_N]^T$ hence the dual problem is formulated as

$$\begin{aligned} &\underset{\boldsymbol{\mu}}{\text{minimize}} && \frac{1}{2}\boldsymbol{\mu}^T(\boldsymbol{d}\cdot\boldsymbol{d}^T)\boldsymbol{\mu} - \boldsymbol{e}^T\boldsymbol{\mu} \\ &\text{subject to:} && \boldsymbol{\mu}^T\boldsymbol{y} = 0, \quad \boldsymbol{0} \leq \boldsymbol{\mu} \leq \tau\boldsymbol{e} \end{aligned} \tag{4.46}$$

On comparing (4.46) with its counterpart (4.11) for the standard SVM, we see that the only difference between them is that an upper bound τ is imposed for soft-margin SVM.

Having solved the dual problem (4.46) for $\boldsymbol{\mu}$, the optimal $\boldsymbol{w}$ is obtained from (4.38) as

$$\boldsymbol{w} = \sum_{n=1}^{N} \mu_n y_n \boldsymbol{x}_n$$

which can be reduced to only include the terms with nonzero μ_n, namely

$$\boldsymbol{w} = \sum_{n \in S} \mu_n y_n \boldsymbol{x}_n \tag{4.47}$$

The points $\{\boldsymbol{x}_n,\ n \in S\}$ are called support vectors. For soft-margin SVM, however, it is important to realize there are two types of support vectors:

- ***Margin support vectors*** These vectors $\boldsymbol{x}_n$ are associated μ_n with $0 < \mu_n < \tau$. It follows (4.41) that $1 - y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) = s_n$. Because $\mu_n < \tau$, (4.40) implies that ν_n is nonzero, hence (4.42) implies that $s_n = 0$. Therefore we have

$$y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) = 1 \tag{4.48}$$

- ***Non-margin support vectors*** These vectors $\boldsymbol{x}_n$ are associated μ_n with $\mu_n = \tau$. Again, it follows (4.41) that $1 - y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) = s_n$. Now, because $\mu_n = \tau$, (4.40) leads to $\nu_n = 0$, hence (4.42) and (4.44) imply that

$$y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) < 1$$

To determine optimal intersect b, let $\boldsymbol{x}_m$ be a margin support vector. From (4.48) it follows that

$$b = y_m - \boldsymbol{w}^T \boldsymbol{x}_m \tag{4.49}$$

where $\boldsymbol{w}$ is given by (4.47).

<u>Soft-Margin Support Vector Machines with Nonlinear Transformation</u>

It is rather straightforward to extend the soft-margin SVM from feature space $\mathcal{X}$ to a dilated space $\mathcal{Z}$ with nonlinear transformation $\boldsymbol{z} = \Phi(\boldsymbol{x})$. The primal and dual problems for space $\mathcal{Z}$ are given by

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\boldsymbol{w}^T \boldsymbol{w} + \tau \sum_{n=1}^{N} s_n \\ &\text{subject to:} && y_n(\boldsymbol{w}^T \boldsymbol{z}_n + b) \geq 1 - s_n \quad \text{for } n = 1, \ldots, N \\ & && s_n \geq 0 \quad \text{for } n = 1, \ldots, N \end{aligned} \tag{4.50}$$

and

$$\begin{aligned} &\underset{\boldsymbol{\mu}}{\text{minimize}} && \frac{1}{2}\boldsymbol{\mu}^T (\tilde{\boldsymbol{d}} \cdot \tilde{\boldsymbol{d}}^T)\boldsymbol{\mu} - \boldsymbol{e}^T \boldsymbol{\mu} \\ &\text{subject to:} && \boldsymbol{\mu}^T \boldsymbol{y} = 0, \quad \boldsymbol{0} \leq \boldsymbol{\mu} \leq \tau \boldsymbol{e} \end{aligned} \tag{4.51}$$

respectively, where $\tilde{\boldsymbol{d}} = [y_1 z_1 \quad y_2 z_2 \quad \cdots \quad y_N z_N]^T$.

<u>Example 4.4</u> Apply soft-margin SVM with a second-order nonlinear transformation to the data set utilized in Example 3.6 to classify handwritten digit "1" from other 9 digits. The size of the training data is $N = 500$ in a 2-dimensional feature space.

Solution

With $z = \Phi_2(x)$ and $\tau = 0.7$, soft-margin SVM was applied by solving the primal problem (4.50). The optimal separating weight w was found to be

-1.723630104245624
2.833732449476559
-1.273873229835718
-0.130448829121387
3.332236270017219

and b = -2.171268426130970 which defines a decision boundary shown in Fig. 4.4. Out of 500 digits, a total of 488 digits were classified correctly, representing an error rate of 2.4%. The MATLAB function implementing the SVM for Example 4.4 is svm_soft_NL.m which is enclosed at the end of this chapter.

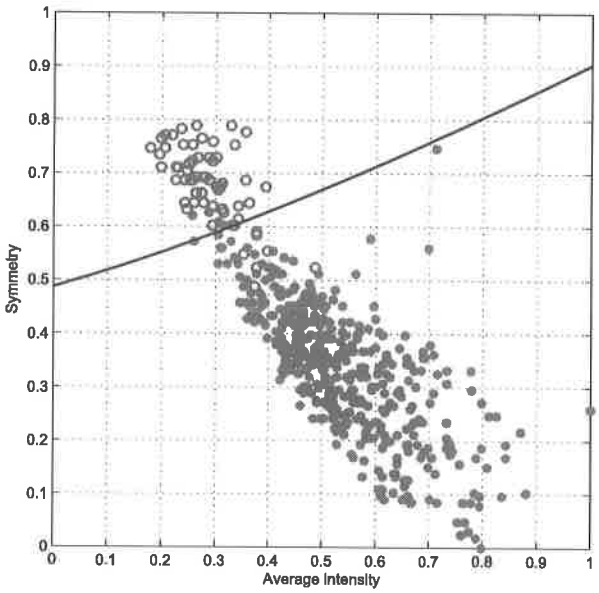


Figure 4.4 Decision boundary and training data for Example 4.4.

4.3 Radial Basis Functions for Classification and Regression

The radius of a diverging circle centered at point x_n is measure by the 2- norm $||x - x_n||$, and a "local" function associated with a data point x_n is said to be a *radial basis function* (RBF) if it is a function of $||x - x_n||$ and can act as a member of basis functions for functional representation and approximation purposes. A class of commonly used radial basis functions (RBFs) are *Gaussian* RBFs defined by

$$K(x, x_n) = e^{-\gamma ||x - x_n||^2} \tag{4.52}$$

where $\gamma > 0$ is a user-defined parameter to control the decay rate of the RBF. It turns out that the value of γ effects the performance of a RBF-based model in a significant way, hence should be chosen with care. On comparing (4.52) with (4.30), we see that Gaussian RBFs are essentially Gaussian kernels. Recall a kernel $K(\boldsymbol{x}, \boldsymbol{x}')$ is said to be positive definite if matrix $\left(K(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)$ is positive semidefinite for *any* $\{\boldsymbol{x}_i, i = 1, \ldots, N\}$ (see Sec. 4.2.A). It can be shown that Gaussian RBFs are positive definite.

In a RBF model with training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, \ldots, N\}$, the hypothesis function assumes the form

$$h(\boldsymbol{x}) = \sum_{n=1}^{N} w_n K(\boldsymbol{x}, \boldsymbol{x}_n) \tag{4.53}$$

for regression problems, and

$$h(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{n=1}^{N} w_n K(\boldsymbol{x}, \boldsymbol{x}_n)\right) \tag{4.54}$$

for classification problems, where coefficients $\{w_n, n = 1, \ldots, N\}$ are determined by minimizing an error measure. Because of the structure of function $h(\boldsymbol{x})$ (having exactly N terms) and the particular choice of RBF in (4.52) which is known to be *positive define* (i.e., it obeys Mercer's condition), alternatively coefficients $\{w_n, n = 1, \ldots, N\}$ may be determined by solving the *exact interpolation* problem

$$h(\boldsymbol{x}_m) = \sum_{n=1}^{N} w_n K(\boldsymbol{x}_m, \boldsymbol{x}_n) = y_m \quad \text{for } m = 1, \ldots, N \tag{4.55}$$

Define

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \quad \boldsymbol{K} = \begin{bmatrix} K(\boldsymbol{x}_i, \boldsymbol{x}_j) & K(\boldsymbol{x}_i, \boldsymbol{x}_j) & \cdots & K(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ K(\boldsymbol{x}_i, \boldsymbol{x}_j) & K(\boldsymbol{x}_i, \boldsymbol{x}_j) & \cdots & K(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ \vdots & \vdots & \ddots & \vdots \\ K(\boldsymbol{x}_i, \boldsymbol{x}_j) & K(\boldsymbol{x}_i, \boldsymbol{x}_j) & \cdots & K(\boldsymbol{x}_i, \boldsymbol{x}_j) \end{bmatrix} \tag{4.56}$$

the linear system of equations in (4.55) can be expressed as

$$\boldsymbol{K}\boldsymbol{w} = \boldsymbol{y} \tag{4.57}$$

where $\boldsymbol{K}$ is positive semidefinite. If $\boldsymbol{K}$ is positive definite, the exact interpolation problem in (4.55) has a unique solution which can be found efficiently by solving (4.57) using for example conjugate-gradient method [5]. If $\boldsymbol{K}$ is only positive semidefinite, system (4.57) is modified to

$$(\boldsymbol{K} + \lambda \boldsymbol{I})\boldsymbol{w} = \boldsymbol{y} \tag{4.58}$$

where parameter $\lambda > 0$ regularizes the linear system to deal with the singularity of $\boldsymbol{K}$, data noise as well as possible overfitting issues. Since system (4.58) is always positive definite, conjugate-gradient method finds its solution efficiently. In the literature, the classification hypothesis function in (4.54) with $\boldsymbol{w}$ given by the solution of (4.58) is also known as *regularized least-squares classification* (RLSC) [11].

Example 4.5 Apply RLSC to the data set utilized in Example 4.4 to classify handwritten digit

"1" from other 9 digits. The size of the training data is $N = 500$ in a 2-dimensional feature space.

Solution

With $\gamma = 90$ and $\lambda = 7$, RLSC was the data set utilized in Example 4.4. The optimal $\boldsymbol{w}$ was found by solving (4.58) using conjugate-gradient method [5]. The decision boundary obtained is shown in Fig. 4.5. Out of 500 digits, a total of 491 digits were classified correctly, representing an error rate of 1.8%. The MATLAB function implementing RLSC for Example 5.5 is rlsc_gaussian_digits.m which is enclosed at the end of this chapter.

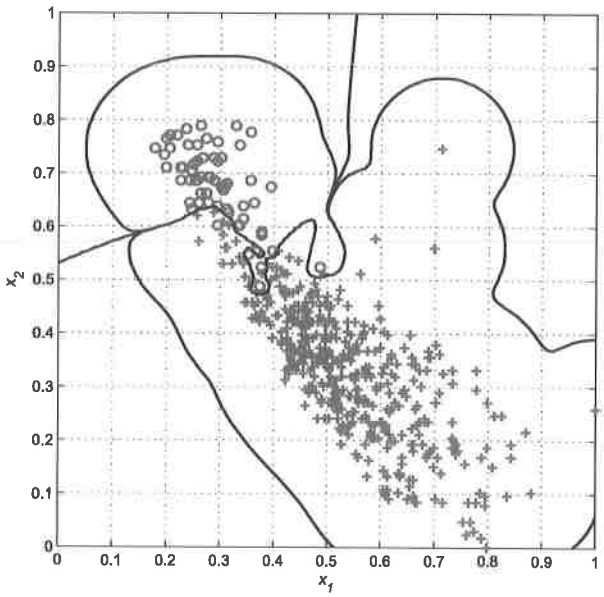


Figure 4.5 Decision boundary and training data for Example 4.5.

MATLAB Code Used in This Chapter

(1) svm_linear.m

```
% For Example 4.1 where linear SVM was applied to the same data as that
% used in Example 1.3 for PLA.
% Written by W.-S. Lu, University of Victoria. Last modified: March 23, 2015.
% Output:
% (w,b): optimal separating weight.
% sv: support vectors.
% marg: optimal margin of decision boundary.
% Example:
% load data_ex1_3
% [w,b,sv,marg] = svm_linear(x,y,xp,xn);
function [w,b,sv,marg] = svm_linear(x,y,xp,xn)
N = length(y);
ind = 1:1:N;
y = y(:);
d = diag(y)*x';
e = ones(N,1);
cvx_begin quiet
  variable u(N,1);
  dw = d'*u;
  minimize(0.5*(dw'*dw) - e'*u);
  subject to
  y'*u == 0;
  u >= 0;
cvx_end
mu = u;
ind1 = find(mu < 1e-6);
ind2 = setdiff(ind,ind1);
sv = x(:,ind2);
mu1 = mu(ind2);
y1 = y(ind2);
c = mu1(:).*y1;
w = sv*c;
marg = 1/norm(w);
b = y1(1) - w'*sv(:,1);
wt = [b; w];
Dt = [ones(N,1) x'];
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp('optimal weights:')
wt
Ein = L/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
disp(sprintf('optimal margin of decision boundary: %d.',marg));
figure(1)
```

```
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
nt = size(sv,2);
for i = 1:nt,
   if y1(i) == 1,
      plot(sv(1,i),sv(2,i),'b.','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'b+','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'bx','linewidth',1.5)
   else
      plot(sv(1,i),sv(2,i),'r.','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'r+','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'ro','linewidth',1.5)
   end
end
grid
xlabel('(b)  \itx_1 (years in residence)')
ylabel('\itx_2 (salary in 10K dollars)')
p1 = 0;
p2 = (-wt(2)*p1-wt(1))/wt(3);
q1 = 12;
q2 = (-wt(2)*q1-wt(1))/wt(3);
plot([p1 q1],[p2 q2],'k-','linewidth',1.5)
axis([0 12 0 8])
axis square
```

(2) svm_NL_semi_circle.m

```
% For Example 4.2 where SVM with 3rd-order nonlinearity was
% applied to the same data as that produced in Example 2.1.
% Written by W.-S. Lu, University of Victoria. Last modified: March 23, 2015.
% Output:
% (w,b): optimal separating weight.
% sv: "support vectors" in basic feature space.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [w,b,sv] = svm_NL_semi_circle(x,y,xp,xn,17);
function [w,b,sv] = svm_NL_semi_circle(x,y,xp,xn,st)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
y = y(:);
z1 = zeros(7,N);
for i = 1:N,
   z1(:,i) = [x(1,i)^2; x(1,i)*x(2,i); x(2,i)^2; x(1,i)^3; x(1,i)^2*x(2,i); x(1,i)*x(2,i)^2; x(2,i)^3];
end
z = [x; z1];
```

```
D = diag(y)*z';
e = ones(N,1);
cvx_begin quiet
  variable w(9,1);
  variable b(1)
  minimize(w'*w);
  subject to
  D*w + y*b >= e;
cvx_end
c = D*w + y*b;
ind1 = find(abs(c - e) < (1e-5)*e);
sv = x(:,ind1);
y1 = y(ind1);
wt = [b; w];
Dt = [ones(N,1) z'];
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp('optimal weights:')
wt
Ein = L/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
nt = size(sv,2);
for i = 1:nt,
   if y1(i) == 1,
      plot(sv(1,i),sv(2,i),'b.','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'b+','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'bx','linewidth',1.5)
   else
      plot(sv(1,i),sv(2,i),'r.','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'r+','linewidth',1.5)
      plot(sv(1,i),sv(2,i),'ro','linewidth',1.5)
   end
end
[x1,x2] = meshgrid(-20:50/100:30,-25:50/100:25);
w0 = wt(1); w1 = wt(2); w2 = wt(3); w3 = wt(4); w4 = wt(5); w5 = wt(6);
w6 = wt(7); w7 = wt(8); w8 = wt(9); w9 = wt(10);
h1 = w0 + w1*x1 + w2*x2 + w3*(x1.^2) + w4*(x1.*x2) + w5*(x2.^2);
h = h1 + w6*(x1.^3) + w7*((x1.^2).*x2) + w8*(x1.*(x2.^2)) + w9*(x2.^3);
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
contour(x1,x2,h,v,'k-','linewidth',1.5);
```

```
grid
xlabel('\itx_1')
ylabel('\itx_2')
axis square
axis([-20 30 -25 25])
hold off
```

(3) svm_gaussian_semi_circle

```
% For Example 4.3 where SVM with a Gaussian kernel was applied to the same
% data as that produced in Example 2.1.
% Written by W.-S. Lu, University of Victoria. Last modified: March 26, 2015.
% Input:
% (x,y); training data.
% xp: training data with positive labels.
% xn: training data with negative labels.
% st: initial random state for mixing training data.
% sig: \sigma for Gaussian kernel.
% Output:
% mu: optimal solution of the dual problem.
% b: optimal intersect.
% sv: "support vectors" in basic feature space.
% cpt: CPU time consumed by CVX.
% Example:
% [x,y,xp,xn] = data_semi_circle(10,5,-1,1000,9,7);
% [mu,b,sv] = svm_gaussian_semi_circle(x,y,xp,xn,17,0.8);
function [mu,b,sv,cpt] = svm_gaussian_semi_circle(x,y,xp,xn,st,sig)
N = length(y);
rand('state',st)
N1 = randperm(N);
x0 = x;
y0 = y;
x = x0(:,N1);
y = y0(N1);
y = y(:);
D1 = eye(N);
for i = 1:(N-1),
   for j = (i+1):N,
      nij = (norm(x(:,i)-x(:,j)))^2;
      D1(i,j) = exp(-nij/(2*sig^2));
      D1(j,i) = D1(i,j);
   end
end
Y = y*y';
D = Y.*D1;
clear Y
e = ones(N,1);
cvx_begin quiet
  variable u(N,1);
  minimize(0.5*u'*D*u - e'*u);
  subject to
  u'*y == 0;
```

```
  u >= 0;
cvx_end
cpt = cvx_cputime
mu = u;
ind = 1:1:N;
ind1 = find(mu < 1e-5);
ind2 = setdiff(ind,ind1);
sv = x(:,ind2);
mu1 = mu(ind2);
y1 = y(ind2);
c = mu1(:).*y1;
di = D1(ind2(1),ind2);
b = y1(1) - di*c;
nt = length(ind2);
dw = b*ones(N,1);
for i = 1:N,
   xi = x(:,i);
   dwi = dw(i);
   for j = 1:nt,
      nij = (norm(sv(:,j)-xi))^2;
      dij = exp(-nij/(2*sig^2));
      dwi = dwi + c(j)*dij;
   end
   dw(i) = dwi;
end
dwt = (dw >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
Ein = L/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
N2 = 100;
[x1,x2] = meshgrid(-20:50/N2:30,-25:50/N2:25);
h = b*ones(N2+1,N2+1);
for i = 1:nt,
   x1w = sv(1,i);
   x2w = sv(2,i);
   m1w = (x1w - x1).^2;
   m2w = (x2w - x2).^2;
   m2 = sqrt(m1w + m2w)/(2*sig^2);
   Kw = exp(-m2);
   h = h + c(i)*Kw;
end
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
contour(x1,x2,h,v,'k-','linewidth',1.5);
grid
```

```
xlabel('\itx_1')
ylabel('\itx_2')
axis square
axis([-20 30 -25 25])
hold off
```

(4) svm_soft_NL.m

```
% For Example 4.4 where soft-margin SVM with 2rd-order nonlinearity
% was applied to the same data as that produced in Example 3.6.
% Written by W.-S. Lu, University of Victoria. Last modified: March 27, 2015.
% Input:
% (x,y): training data.
% tau: parameter \tau for the total violation term.
% Output:
% (w,b): optimal separating weight.
% Example:
% load data_ex4_4
% [w,b] = svm_soft_NL(x,y,xp,xn,0.7);
function [w,b] = svm_soft_NL(x,y,xp,xn,tau)
N = length(y);
y = y(:);
z1 = zeros(3,N);
for i = 1:N,
   z1(:,i) = [x(1,i)^2; x(1,i)*x(2,i); x(2,i)^2];
end
z = [x; z1];
D = diag(y)*z';
e = ones(N,1);
cvx_begin quiet
  variable w(5,1);
  variable b(1);
  variable s(N,1);
  minimize(0.5*w'*w + tau*sum(s));
  subject to
  D*w + y*b + s >= e;
  s >= 0;
cvx_end
wt = [b; w];
Dt = [ones(N,1) z'];
dwt = (Dt*wt >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
disp('optimal weights:')
wt
Ein = L/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
```

```
plot(xn(1,:),xn(2,:),'rx','linewidth',1.5)
[x1,x2] = meshgrid(0:1/100:1,0:1/100:1);
w0 = wt(1); w1 = wt(2); w2 = wt(3); w3 = wt(4); w4 = wt(5); w5 = wt(6);
h = w0 + w1*x1 + w2*x2 + w3*(x1.^2) + w4*(x1.*x2) + w5*(x2.^2);
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
contour(x1,x2,h,v,'k-','linewidth',1.5);
grid
xlabel('Average intensity')
ylabel('Symmetry')
axis square
axis([0 1 0 1])
hold off
```

(5) rlsc_gaussian_digits.m

```
% For Example 4.5 where a Gaussian RBF was applied to the same data
% in Example 4.4 for classify digit 1 from other digits..
% Written by W.-S. Lu, University of Victoria. Last modified: March 28, 2015.
% Input:
% (x,y); training data.
% xp: training data with positive labels.
% xn: training data with negative labels.
% gam: parameter \gamma for Gaussian RBF.
% lam: parameter \lambda for regularized LS classification (RLSC).
% Output:
% mu: optimal solution of the dual problem.
% b: optimal intersect.
% sv: "support vectors" in basic feature space.
% cpt: CPU time consumed by CVX.
% Example:
% load data_ex4_4
% w = rlsc_gaussian_digits(x,y,xp,xn,90,7);
function w = rlsc_gaussian_digits(x,y,xp,xn,gam,lam)
N = length(y);
y = y(:);
I = eye(N,N);
K = I;
for i = 1:(N-1),
   for j = (i+1):N,
      nij = (norm(x(:,i)-x(:,j)))^2;
      K(i,j) = exp(-gam*nij);
      K(j,i) = K(i,j);
   end
end
w = conj_gradient_2015(lam*I+K,y,1e-6);
% w = inv(lam*I+K)*y;
dw = zeros(N,1);
for i = 1:N,
```

```
    xi = x(:,i);
    dwi = dw(i);
    for j = 1:N,
        nij = (norm(x(:,j)-xi))^2;
        dij = exp(-gam*nij);
        dwi = dwi + w(j)*dij;
    end
    dw(i) = dwi;
end
dwt = (dw >= 0);
z = dwt + dwt - y - 1;
L = sum(abs(z))/2;
Ein = L/N;
disp(sprintf('In-sample error was found to be %d.', Ein));
disp(sprintf('Out of N = %d sample points,',N));
disp(sprintf('%d points were classified correctly.',N-L));
N2 = 100;
[x1,x2] = meshgrid(0:1/N2:1,0:1/N2:1);
h = zeros(N2+1,N2+1);
for i = 1:N,
    x1w = x(1,i);
    x2w = x(2,i);
    m1w = (x1w - x1).^2;
    m2w = (x2w - x2).^2;
    m2 = gam*sqrt(m1w + m2w);
    Kw = exp(-m2);
    h = h + w(i)*Kw;
end
figure(1)
plot(xp(1,:),xp(2,:),'bo','linewidth',1.5)
hold on
plot(xn(1,:),xn(2,:),'r+','linewidth',1.5)
v = -1e-6:1e-6:1e-6;
contour(x1,x2,h,v,'k-','linewidth',1.5);
grid
xlabel('\itx_1')
ylabel('\itx_2')
axis square
axis([0 1 0 1])
hold off
```

```
% Conjugate-gradient algorithm for Ax = b with A positive definite.
% Theory: See Practical Optimization Sec. 6.9.
% Input:
% {A,b} -- data for linear equation A and b.
% epsi: termination tolerance.
% Output:
% xs: solution point.
% Written by W.-S. Lu, University of Victoria. Last modified: March 28, 2015.
function xs = conj_gradient_2015(A,b,epsi)
```

```
n = length(b);
x0 = zeros(n,1);
xk = x0;
rk = b - A*xk;
dk = rk;
rk2_old = rk'*rk;
err = norm(rk);
while  err >= epsi,
   adk = A*dk;
   ak = rk2_old/(dk'*adk);
   xk = xk + ak*dk;
   rk = rk - ak*adk;
   rk2 = rk'*rk;
   bk = rk2/rk2_old;
   rk2_old = rk2;
   dk = rk + bk*dk;
   err = norm(rk);
end
xs = xk;
```

Appendix A: Elements of Probability Theory

A.1 Probability Theory

We begin by presenting a brief review of the probability theory. The very reason of doing so is that contemporary machine learning methodologies are developed in a probabilistic framework [1] - [3]. In other words, most part of machine leaning makes sense only when you look at it from a statistical perspective.

A.1.1 Definition of Probability

A. ***Classical Probability***

There are different kinds of probability: probability as intuition, probability as the ratio of favorable to total outcomes (also known as classical theory of probability), probability as a measure of frequency of occurrence, and probability based on axiomatic theory [4].

Definition A.1 *Random experiment* and *basic event*

An experiment E is said to be a *random experiment* if it can be performed repeatedly and its outcomes are *not* deterministic but probabilistic. An outcome of a random experiment, denoted by ω, is called a *basic event.* ■

Definition A.2 *Sample space*

The set of all basic events, denoted by Ω, is called *sample space.* ■

Example A.1

Let E be the experiment of picking a ball at random from a box containing N identical balls, where the balls have been marked as #1, #2, ..., to #N. Obviously E is random experience because its outcome has N possibilities. If the number on the ball that has been picked is i, then the outcome is a basic event and is denoted by ω_i. Obviously, the sample space is $\Omega = \{\omega_1, \cdots, \omega_N\}$. ■

Example A.2

Let E be that of observing the average daytime temperature in a city in October. Obviously E is a random experiment. If we denote the outcome of experiment E by ω_a, then the sample space in this case may be described as $\Omega = \{\omega_a, -\infty < \omega_a < \infty\}$. Note that Ω contains infinite number of basic events. ■

Definition A.3 *General event*

A general event (or simply called it event), say A, consists of several basic events, hence a general event is a subset of sample space Ω. ■

Example A.3

Continue from Example A.1. Suppose one randomly picks a ball from the box, one can state that

• Event A of getting a ball whose number is no greater than 3 is a general event because it consists of three basic events: $A = \{\omega_1, \omega_2, \omega_3\}$ which is a proper subset of $\Omega = \{\omega_1, \cdots, \omega_N\}$.

• Event B of getting a ball with number even numbers is a general event because $B = \{\omega_2, \omega_4, \ldots, \omega_{100}\} \subset \Omega$. ■

The classical probability theory applies to the cases where the random experiment E obeys two basic assumptions: (i) the total number of outcomes, N, is finite; and (ii) all outcomes (i.e. the results of basic events) are equally likely. Consequently, the probability of a (general) *event A*, denoted by $P(A)$, is obtained *a priori* (i.e., relating to reasoning that proceeds from theoretical deduction rather than from observation or experience) by counting the number of ways N_e that event A can occur, then computing the ratio N_e/N as the probability. This is,

$$P(A) = \frac{N_e}{N} \tag{A.1}$$

Example A.4

Continue from Example A.3.

Let A be the event of obtaining a ball whose number is no greater than 3. The probability $P(A) = 3/100 = 0.03$.

Let B be the event of obtaining a ball that is even-numbered. The probability $P(B) = 50/100 = 0.5$.

Let C be the event of obtaining a ball whose number is a multiple of 3. The probability $P(C) = 33/100 = 0.33$.

■

The major problems with the classical theory of probability are that it cannot handle outcomes that are not equally likely; and it cannot deal with infinite number of outcomes.

B. Probability Based On Axiomatic Theory

Developed by A. N. Kolmogorov in 1930's, the probability based on axiomatic theory is the one that is followed by most modern texts on the subject. Kolmogorov's probability theory is set-theoretic in that a random event is considered as a subset of a sample space in an abstract setting where the two basic assumptions made in the classical probability theory are no longer necessary.

In order to present Kolmogorov's axiomatic definition of probability, basic elements of *set algebra* and the sigma algebra (σ – algebra) are sketched first.

Set Algebra

• A *set* is a collection of objects (or elements).

• A set B is said to be a *subset* of set A, and this relation is denoted by $B \subseteq A$, if B is contained within set A. Let for example set A denote all Victoria residents and set B be all Victoria residents whose height is between 5.5 and 6.5 feet. Obviously set B is a (proper) subset of set A.

• Let A and B be two sets in space Ω. The *union* (sum) of A and B, denoted by $A \bigcup B$ or $A + B$, is the set of elements that are in at least one of the sets A or B.

• The *intersection* (or set product) of A and B, denoted by $A \bigcap B$ or AB, is the set of elements that are in both A and B.

• The *empty* set, denoted by ϕ, is a set that contains no objects.

• Let A be a set in space Ω, the complement of A, denoted by A^c, is a set of all elements that are

not in A. Obviously, $A \bigcup A^c = \Omega$ and $A \bigcap A^c = \phi$.

• Two sets A and B are said to be equal if both $B \subseteq A$ and $A \subseteq B$ hold.

• The difference of A and B, denoted by $A - B$, is the set of elements that are in A but not in B. It follows that

$$A - B = AB^c \quad \text{and} \quad B - A = BA^c$$

• The *exclusive-or* of sets A and B, denoted by $A \oplus B$, is the set of elements that are in A or B, but not in both. Obviously, we can write

$$A \oplus B = (A - B) \bigcup (B - A)$$

• Sets A and B are said to be disjoint if they have no elements in common, thus $AB = \phi$.

• It can readily be verified that

$$(A \bigcup B)^c = A^c \bigcap B^c \quad \text{and} \quad (A \bigcap B)^c = A^c \bigcup B^c$$

which can be extended by induction to the general case of n sets as

$$\left(\bigcup_{i=1}^{n} A_i\right)^c = \bigcap_{i=1}^{n} A_i^c \quad \text{and} \quad \left(\bigcap_{i=1}^{n} A_i\right)^c = \bigcup_{i=1}^{n} A_i^c$$

<u>Definition A.4</u> *σ-Algebra*

Let Ω be a set and we consider a collection of subsets of Ω, denoted by $\mathcal{F}$. $\mathcal{F}$ is said to be a *σ-Algebra* if

1. $\phi \in \mathcal{F}$ and $\Omega \in \mathcal{F}$,

2. If $A \in \mathcal{F}$ and $B \in \mathcal{F}$, then $A \bigcup B \in \mathcal{F}$ and $A \bigcap B \in \mathcal{F}$.

3. If $A \in \mathcal{F}$ then $A^c \in \mathcal{F}$.

4. $\mathcal{F}$ is closed under countable set of unions, intersections, and combinations. Hence if $A_1, \ldots, A_n, \ldots$ belong to $\mathcal{F}$, then

$$\bigcup_{i=1}^{\infty} A_i \in \mathcal{F} \text{ and } \bigcap_{i=1}^{\infty} A_i \in \mathcal{F} \qquad \blacksquare$$

<u>Why is σ-Algebra relevant?</u>

Recall in the classical theory of probability the number of basic events must be finite and all outcomes are equally likely, i.e. the probability of every basic event is the same. The axiomatic probability theory built on σ-algebra allows us to remove these two fundamental limitations. Consider a random experiment and the associated sample space Ω. In the axiomatic probability theory we only consider the collection $\mathcal{F}$ of those subsets of Ω that form a σ-algebra. These subsets are called events. It turns out that $\mathcal{F}$ usually includes all subsets of engineering and science interest.

<u>Definition A.5</u> *Axiomatic probability*

Given a sample space Ω and a σ-algebra $\mathcal{F}$ of Ω, probability $P[\cdot]$ is a set function that assigns to every event $A \in \mathcal{F}$ that obeys the three axioms:

(1) $P[A] \geq 0$.

(2) $P[\Omega] = 1$.

(3) $P[A \cup B] = p[A] + P[B]$ if $A \cap B = \phi$.

From these axioms, one can establish the following important properties:

(4) $P[\phi] = 0$.

(5) $P[A \cap B^c] = P[A] - P[A \cap B]$.

(6) $P[A] = 1 - P[A^c]$.

(7) $P[A \cup B] = P[A] + P[B] - P[A \cap B]$.

(8) $P\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} P[A_i]$ if $A_i \cap A_j = \phi$ for all $i \neq j$. ■

Definition A.6 *Probability space*

The triple of a sample space Ω, a collection of (general) events $\mathcal{F}$ that form a σ-algebra, and a probability measure P, namely $(\Omega, \mathcal{F}, P)$, is called a probability space. ■

A.1.2 Joint Probability, Conditional Probability, and Independence

A. Joint Probability

The *joint probability* of two events A and B is defined as the probability of "events A and B both occur", namely $P[A \cap B]$ or P[AB].

B. Conditional Probability

Often times people want to know the probability of "event A occurs" given that event B has occurred. This is called *conditional probability* and is denoted by $P[A|B]$. It is intuitively clear that because of the presence of the condition that "event B has occurred", $P[A|B]$ is in general different from $P[A]$.

Example A.5

Suppose there are four identical balls in a box, which are marked as #1, #2, #3, and #4. The random experiment is to pick up a ball from the box. Now consider two events: Event A is that of obtaining ball #4; and event B is that of obtaining an even-numbered ball. Obviously, we have $P[A] = ¼$ and $P[A|B] = ½$. ■

A question that naturally arises is how to define and compute conditional probability in general circumstances? Let A and B are two events in a random experiment. Each outcome of the random

experiment must fall into one of the four cases: (1) A occurs and B does not; (2) B occurs and A does not; (3) A and B both occur; and (4) A and B both do not occur.

Suppose one applies above analysis to Example A.5, repeats the experiment n times, and denotes the times that each of the four cases occurs by n_1, n_2, n_3, and n_4, respectively. We can state that

- $n_1 + n_2 + n_3 + n_4 = n$
- the frequency of event $B = F_n(B) = \dfrac{n_2 + n_3}{n}$.
- the frequency of event $AB = F_n(AB) = \dfrac{n_3}{n}$.
- given that event B has occurred, the frequency of event $A = F_n(A|B) = \dfrac{n_3}{n_2 + n_3}$.

It follows that

$$F_n(A \mid B) = \frac{F_n(AB)}{F_n(B)} \quad \text{provided that} \quad F_n(B) > 0 \tag{A.2}$$

Based on (A.2), conditional probability is defined as follows.

Definition A.7 *Conditional probability*

Let $(\Omega, \mathcal{F}, P)$ be a probability space, $A \in \mathcal{F}, B \in \mathcal{F}$ with $P[B] > 0$. The conditional probability of event A given that event B has occurred is defined by

$$P[A \mid B] = \frac{P[AB]}{P[B]} \tag{A.3}$$

Similarly, the conditional probability of event B given that event A has occurred is defined by

$$P[B \mid A] = \frac{P[AB]}{P[A]} \tag{A.4}$$

provided that $P[A] > 0$. From (A.3) and (A.4) it follows that

$$P[AB] = P[A]P[B \mid A] = P[B]P[A \mid B] \qquad \blacksquare \tag{A.5}$$

Example A.6

Consider a binary communication system with a two-symbol alphabet, i.e., 0 and 1. Let X and Y be the transmitted and received symbols, respectively. Here the sample space is given by $\Omega = \{(X,Y) : X = 0 \text{ or } 1, Y = 0 \text{ or } 1\} = \{(0,0), (0,1), (1,0), (1,1)\}$. Suppose that the communication system is slightly corrupted by noise such that

$$P[Y = 1 \mid X = 1] = 0.92, \quad P[Y = 0 \mid X = 1] = 0.08$$
$$P[Y = 0 \mid X = 0] = 0.92, \quad P[Y = 1 \mid X = 0] = 0.08$$

And by design, $P[X=0] = P[X=1] = 0.5$. Under these circumstances, we have

$$P[X=0,Y=0] = P[X=0]\cdot P[Y=0\mid X=0] = 0.5\times 0.92 = 0.46$$
$$P[X=0,Y=1] = P[X=0]\cdot P[Y=1\mid X=0] = 0.5\times 0.08 = 0.04$$
$$P[X=1,Y=0] = P[X=1]\cdot P[Y=0\mid X=1] = 0.5\times 0.08 = 0.04$$
$$P[X=1,Y=1] = P[X=1]\cdot P[Y=1\mid X=1] = 0.5\times 0.92 = 0.46$$

■

Properties of Conditional Probability

Property 1 (Multiplication formula)

Let $A_1, A_2, \ldots, A_n$ be n events with $n \geq 2$ and $P[A_1A_2\cdots A_{n-1}] > 0$, then

$$P[A_1A_2\cdots A_n] = P[A_1]\cdot P[A_2\mid A_1]\cdot P[A_3\mid A_1A_2]\cdots P[A_n\mid A_1A_2\cdots A_{n-1}] \qquad \text{(A.6)}$$

Proof:

Using (A.3), we see that the right-hand side of (1.6) is equal to

$$P[A_1]\cdot\frac{P[A_1A_2]}{P[A_1]}\cdot\frac{P[A_1A_2A_3]}{P[A_1A_2]}\cdots\frac{P[A_1A_2\cdots A_n]}{P[A_1A_2\cdots A_{n-1}]} = P[A_1A_2\cdots A_n] \qquad ■$$

Property 2 (Unconditional probability)

Let $A_1, A_2, \ldots, A_n$ be mutually exclusive events such that $\bigcup_{i=1}^{n} A_i = \Omega$ with $P[A_i] > 0$ for all i. Let B be any event defined over the probability space of A_i's. Then

$$P[B] = \sum_{i=1}^{n} P[A_i]P[B\mid A_i] \qquad \text{(A.7)}$$

Proof:

From

$$B = B\Omega = B\bigcup_{i=1}^{n} A_i = \bigcup_{i=1}^{n} BA_i$$

where $\{BA_i\}$ are mutually exclusive, it follows that

$$P[B] = P\left[\bigcup_{i=1}^{n} BA_i\right] = \sum_{i=1}^{n} P[BA_i] = \sum_{i=1}^{n} P[A_i]\cdot P[B\mid A_i] \qquad ■$$

Property 3 (Bayes formula)

Let $A_1, A_2, \ldots, A_n$ be mutually exclusive events such that $\bigcup_{i=1}^{n} A_i = \Omega$ with $P[A_i] > 0$ for all i.

Let B be any event with $P[B] > 0$. Then

$$P[A_j \mid B] = \frac{P[B \mid A_j] \cdot P[A_j]}{\sum_{i=1}^{n} P[B \mid A_i] \cdot P[A_i]} \tag{A.8}$$

Proof:

By following the definition of conditional probability and the formula of unconditional probability in (A.7), we obtain

$$P[A_j \mid B] = \frac{P[BA_j]}{P[B]} = \frac{P[B \mid A_j] \cdot P[A_j]}{\sum_{i=1}^{n} P[B \mid A_i] \cdot P[A_i]} \qquad \blacksquare$$

The Bayes formula finds many applications in science and engineering. The terms in (A.8) bear various names: $P[A_j|B]$ is often called *a posteriori* probability of A_j given B; $P[B|A_j]$ is called the *a priori* probability of B given A_j; and $P[A_j]$ is called the *causal* or *a priori* probability of A_j. Typically *a priori* probabilities are estimated from past measurements or presupposed by experience, while *a posteriori* probabilities are measured or computed from observations [4].

Example A.7

Suppose there are three boxes that look identical, each contains certain number of red and blue balls that are identical except the color. The number of red and blue balls in box i are r_i and b_i, respectively, for $i = 1, 2, 3$. The experiment in question is to randomly pick a box, then randomly pick a ball. The outcome was a red ball. Given that it is a red ball, compute the probability of the ball belongs to box 1.

Solution

Let A_i be the event of the ball in question belongs to box i for $i = 1, 2, 3$. Obviously, $\{A_i$, for $i = 1, 2, 3\}$ are mutually exclusive, $\bigcup_{i=1}^{3} A_i = \Omega$, and $P[A_1] = P[A_2] = P[A_3] = 1/3$. We also define event B as "it is a red ball". With these definitions, the problem we need to address is to compute the conditional probability $P[A_1|B]$.

Clearly, formula (1.8) is applicable if we are able to compute conditional probabilities $P[B|A_i]$ for $i = 1, 2, 3$. These conditional probabilities are found to be

$$P[B \mid A_i] = \frac{r_i}{r_i + b_i} \quad \text{for} \quad i = 1, 2, 3.$$

Hence (A.8) yields

$$P[A_1 \mid B] = \frac{\frac{1}{3}\frac{r_1}{r_1 + b_1}}{\frac{1}{3}\frac{r_1}{r_1 + b_1} + \frac{1}{3}\frac{r_2}{r_2 + b_2} + \frac{1}{3}\frac{r_3}{r_3 + b_3}} = \frac{1}{1 + \frac{r_2(r_1 + b_1)}{r_1(r_2 + b_2)} + \frac{r_3(r_1 + b_1)}{r_1(r_3 + b_3)}} \qquad \blacksquare$$

C. *Independence*

Consider two events A and B. We have seen that in general probability $P[A]$ differs from the conditional probability $P[A|B]$ as long as the occurrence of event B has an impact on occurrence of event A. In other words, if $P[A] = P[A|B]$ happens, then the occurrence of B has no impact on

the occurrence of A, and we will say events A and B are mutually independent. Note that $P[A] = P[A|B]$ in conjunction with (A.3) leads to

$$P[AB] = P[A] \cdot P[B] \tag{A.9}$$

This motivates the following definition.

Definition A.8 *Independence*

- Events A and B are said to be independent from each other if (1.9) holds.
- n events $A_1, A_2, \ldots, A_n$ are said to be mutually independent if

$$P[A_1 A_2 \cdots A_n] = P[A_1] \cdot P[A_2] \cdots P[A_n] \quad \blacksquare \tag{A.10}$$

A.1.3 Random Variables, Distribution Function, and Probability Density

A. Random Variables

Let E be a random experiment associated with sample space Ω. Corresponding to each outcome ω, suppose there is a real-valued function $\xi(\omega)$. In probability theory, one is concerned not only with the value of $\xi(\omega)$, but also with the probability of $\xi(\omega)$ taking certain values.

Example A.8

In a junior-high school a student (that is an ω) is randomly selected, whose height is recorded as $\xi(\omega)$. One of the valid questions in this case would be "what is the probability of the height no greater than x cm?", namely, what is $P[\xi(\omega) \le x]$? ■

Evidently, to address the question in the associated probability space $(\Omega, \mathcal{F}, P)$, it is necessary to assure that $\xi(\omega) \le x$ belongs to $\sigma-$ algebra $\mathcal{F}$ so that $P[\xi(\omega) \le x]$ is well defined. In probability theory functions of this kind are called *random variables*.

Definition A.9 *Random variables*

Let $\xi(\omega)$ be a real-valued function defined over sample space Ω that is associated with a probability space $(\Omega, \mathcal{F}, P)$. If, for any real x, $(\xi(\omega) \le x)$ is an event, i.e., $(\xi(\omega) \le x) \in \mathcal{F}$, then $\xi(\omega)$ is called a random variable. ■

The introduction of random variables was a major event in the development of modern probability theory as the concept made it possible to extend the probability theory to include studies of random variables and a variety of related issues. One of the issues is *distribution function* of a random variable.

B. Distribution Function and Probability Density

Definition A.10 *Distribution function of a random variable*

Let $\xi(\omega)$ be a random variable, the distribution function of ξ, denoted by $\Phi_\xi(x)$, is defined as the probability of event $(\xi(\omega) \le x)$. Namely,

$$\Phi_\xi(x) = P[\xi(\omega) \le x] \qquad \blacksquare \qquad \text{(A.11)}$$

Several basic properties of distribution functions follow:

- $\Phi_\xi(x)$ is monotonically non-decreasing, i.e.,

$$x_2 \ge x_1 \text{ implies that } \Phi_\xi(x_2) \ge \Phi_\xi(x_1) \qquad \text{(A.12)}$$

- $$\Phi_\xi(-\infty) = 0, \ \Phi_\xi(\infty) = 1 \qquad \text{(A.13)}$$

- $$P[x_1 < \xi(\omega) \le x_2] = \Phi_\xi(x_2) - \Phi_\xi(x_1) \qquad \text{(A.14)}$$

- $$P[\xi(\omega) > x] = 1 - \Phi_\xi(x) \qquad \text{(A.15)}$$

- $$P[\xi(\omega) = x] = \Phi_\xi(x) - \Phi_\xi(x-0) \qquad \text{(A.16)}$$

where $\Phi_\xi(x-0) = \lim_{a \uparrow x} \Phi_\xi(a)$.

Types of distribution functions

There are several types of distribution functions, with the most important being *discrete type* and *continuous type*.

Discrete type distribution

A discrete random variable ξ takes on discrete values $x_0, x_1, x_2, \ldots$ (finite or countable infinite), with probabilities $p_0, p_1, p_2, \ldots$, respectively. Thus a discrete distribution is characterized by a two-row matrix, known as *density matrix*, with finite or countable infinite columns as

$$\xi \Leftrightarrow \begin{bmatrix} x_0 & x_1 & x_2 & \cdots \\ p_0 & p_1 & p_2 & \cdots \end{bmatrix} \qquad \text{(A.17)}$$

Obviously we have $p_i \ge 0$ and $\sum_i p_i = 1$. The distribution function in this case is given by

$$F(x) = \sum_{i: x_i \le x} p_i \qquad \text{(A.18)}$$

which is a staircase function that is continuous from right, see Fig. A.1.

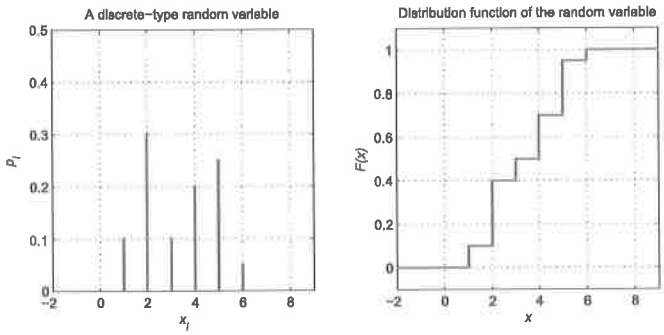


Fig. A.1 Left: A discrete-type random variable with $x_0 = 1, x_1 = 2, \ldots, x_5 = 6$ and $p_0 = 0.1, p_1 = 0.3, p_2 = 0.1, p_3 = 0.2, p_4 = 0.25$, and $p_5 = 0.05$; Right: The distribution function of the random variable.

Example A.9 *Bernoulli distribution*

Also known as *binomial distribution*, the Bernoulli distribution is discrete and assumes the form of

$$\begin{bmatrix} 0 & 1 & \cdots & n \\ p_0 & p_1 & \cdots & p_n \end{bmatrix}$$

where

$$p_k = \binom{n}{k} p^k q^{n-k} \quad p \geq 0,\ q \geq 0,\ p + q = 1 \tag{A.19}$$

with

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

being the number of combinations of taking k elements out of n elements. Note that p_k in (A.19) are exactly the coefficient of polynomial $(p+q)^n$, hence the name of binomial distribution. The reason it also bears the name of Bernoulli distribution has to do with the well-known *Bernoulli trial* which is a random experiment with exactly two possible outcomes: "success" and "failure". Denote the probability of success and failure by p and q, respectively, and suppose the experiment is repeated n time independently. It is obvious that there are 2^n possible outcomes; the number of outcomes with k successes and $(n - k)$ failures is equal to $\binom{n}{k}$; and the probability of "one such event occurs" is $p^k q^{n-k}$. Therefore, the probability of having k successes and $(n - k)$ failures is precisely given by p_k as seen in (A.19). ■

Continuous type distribution

A distribution of random variable ξ is said to be of continuous type if its distribution function assumes the form

$$\Phi_\xi(x) = \int_{-\infty}^{x} \varphi_\xi(y)\,dy \tag{A.20}$$

where $\varphi_\xi(x) \geq 0$ is called *density function* or *density*. Since $\Phi_\xi(\infty) = 1$, we have

$$\int_{-\infty}^{\infty} \varphi_\xi(y)\,dy = 1$$

Example A.10 *Uniform distribution*

Let $a < b$. Consider the random variable ξ that takes a value over the interval $[a, b]$ uniformly randomly. The associated distribution, called uniform distribution, is of continuous type with the density function given by

$$\varphi_\xi(x) = \begin{cases} \dfrac{1}{b-a} & \text{if } x \in [a,b] \\ 0 & \text{if } x \notin [a,b] \end{cases} \tag{A.21}$$

Given the density in (A.21), the distribution function is found to be

$$\Phi_\xi(x) = \int_{-\infty}^{x} \varphi_\xi(y)\,dy = \begin{cases} 0 & \text{if } x < a \\ \dfrac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases} \tag{A.22}$$

See Figs. A.2 and A.3 for plots of the density $\varphi_\xi(x)$ and distribution function $\Phi_\xi(x)$. ■

C. *Expectation, Variance and Covariance*

An important operation involving random variables is that of finding weighted average of a random variable ξ or a (measurable) function of ξ.

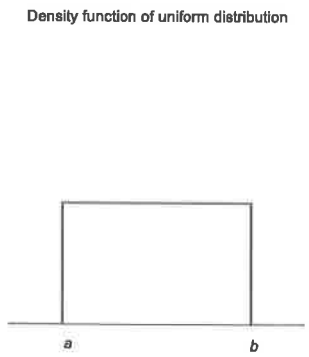


Figure A.2

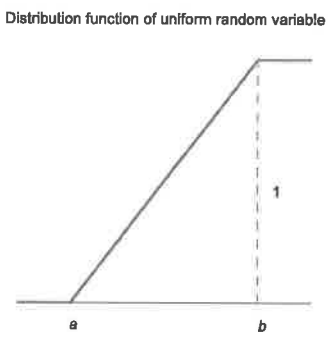


Figure A.3

Definition A.11 *Expectation*

The average of a random variable ξ under its probability distribution is called the expectation of

ξ, denoted by $E[\xi]$.

For a discrete type random variable ξ that takes values x_0, x_1, x_2, ... with probability p_0, p_1, p_2, ... (recall the density matrix given by (A.17), namely

$$\begin{bmatrix} x_0 & x_1 & x_2 & \cdots \\ p_0 & p_1 & p_2 & \cdots \end{bmatrix}$$

), the expectation of ξ, also known as mean value of ξ, is defined as

$$E[\xi] = \sum_i x_i p_i$$

Since the random variable ξ here takes values x_i, for the sake of notation convenience we denote the random variable by x, and write the above expression as

$$E[x] = \sum_i x_i p_i \tag{A.23}$$

For a continuous type random variable ξ with density $\varphi_\xi(x)$, its expectation is defined as

$$E[x] = \int_{-\infty}^{\infty} x\varphi_\xi(x)dx \tag{A.24}$$

The concept of expectation can be extended to expectation of a (measurable) function, say $g(\xi)$, of random variable ξ as

$$E[g(x)] = \sum_i g(x_i)p_i \tag{A.25}$$

for discrete-type case, or

$$E[g(x)] = \int_{-\infty}^{\infty} g(x)\varphi_\xi(x)dx \tag{A.26}$$

for continuous-type case. ■

<u>Definition A.12</u> *Variance*

The variance of a function g of random variable ξ provides a measure of how much there is in g around its mean value $E[g(x)]$:

$$\mathrm{var}[g(x)] = E\left[\left(g(x) - E[g(x)]\right)^2\right] = E\left[g(x)^2\right] - \left(E[g(x)]\right)^2 \tag{A.27}$$

In particular, with $g(x) = x$ (A.27) yields the variance of a random variable ξ itself as

$$\mathrm{var}[x] = E\left[x^2\right] - \left(E[x]\right)^2 \quad ■ \tag{A.28}$$

Definition A.13 *Covariance*

Covariance is concerned with two random variables x and y and quantifies the extent to which x and y vary together, namely,

$$\operatorname{cov}[x,y] = E_{x,y}\left[(x-E[x])(y-E[y])\right] = E_{x,y}[xy] - E[x]E[y] \tag{A.29}$$

where $E_{x,y}$ denotes expectation with respect to joint density $\varphi(x,y)$, namely,

$$E_{x,y}[xy] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} xy\,\varphi(x,y)\,dx\,dy \qquad \blacksquare$$

D. Independence of Random Variables and Conditional Distribution

Definition A.14 *Independence of random variables*

Random variables $\xi_1(\omega),\ldots,\xi_n(\omega)$ with distribution functions $\Phi_1(x_1),\ldots,\ \Phi_n(x_n)$ are said to be *mutually independent* if

$$\Phi(x_1,\ldots,x_n) = \Phi_1(x_1)\cdots\Phi_n(x_n) \tag{A.30}$$

holds, where $\Phi(x_1,\ldots,x_n)$ is the joint distribution of $\xi_1(\omega),\cdots,\xi_n(\omega)$, namely,

$$\Phi(x_1,\ldots,x_n) = P[\xi_1 \le x_1,\cdots,\xi_n \le x_n] \qquad \blacksquare$$

From (A.30), it follows that

$$P[\xi_1 \le x_1,\cdots,\xi_n \le x_n] = P[\xi_1 \le x_1]\cdots P[\xi_n \le x_n] = \prod_{i=1}^{n} P[\xi_i \le x_i] \tag{A.31}$$

In words, if random variables $\xi_1,\ldots,\xi_n$ are mutually independent, the joint probability of the event $(\xi_1 \le x_1,\cdots,\xi_n \le x_n)$ is equal to product of the probability of individual event $(\xi_i \le x_i)$.

For continuous type of random variables, (A.30) implies that $\xi_1(\omega),\cdots,\xi_n(\omega)$ are mutually independent if the joint probability density equals the product of the individual probability density almost everywhere (a.e.), i.e.,

$$\varphi(x_1,\ldots,x_n) = \varphi_1(x_1)\cdots\varphi_n(x_n) \qquad \text{(a.e.)} \tag{A.32}$$

Definition A.15 *Conditional distribution*

For two discrete-type random variables ξ and η with

$$\xi \Leftrightarrow \begin{bmatrix} x_0 & x_1 & x_2 & \cdots \\ p_0 & p_1 & p_2 & \cdots \end{bmatrix} \text{ and } \eta \Leftrightarrow \begin{bmatrix} y_0 & y_1 & y_2 & \cdots \\ q_0 & q_1 & q_2 & \cdots \end{bmatrix}$$

where $p_i > 0$ and $q_i > 0$ are assumed, denote the joint probability

$$P[\xi = x_i, \eta = y_j] = p_{i,j}$$

hence we can write

$$p_i = \sum_j p_{i,j} \text{ and } q_j = \sum_i p_{i,j}$$

which in turn implies that

$$P[\xi = x_i \mid \eta = y_j] = \frac{P[\xi = x_i, \eta = y_j]}{P[\eta = y_j]} = \frac{p_{i,j}}{q_j} = \frac{p_{i,j}}{\sum_i p_{i,j}} \tag{A.33}$$

and

$$P[\eta = y_j \mid \xi = x_i] = \frac{p_{i,j}}{p_i} = \frac{p_{i,j}}{\sum_j p_{i,j}} \tag{A.34}$$

The conditional distribution refers to the probability of event $(\xi \le x \text{ given } \eta = y_j)$ or that of $(\eta \le y \text{ given } \xi = x_i)$. By using (A.33) and (A.34), we obtain

$$P[(\xi \le x \mid \eta = y_j)] = \frac{\sum_{i:x_i \le x} p_{i,j}}{\sum_i p_{i,j}} \tag{A.35}$$

and

$$P[\eta \le y \mid \xi = x_i] = \frac{\sum_{j:y_j \le y} p_{i,j}}{\sum_j p_{i,j}} \tag{A.36}$$

From (A.35) and (A.36), we see that conditional distributions involving two discrete-type random variables can be expressed (and evaluated) using their joint probability distribution $p_{i,j}$.

For two continuous-type random variables ξ and η with joint probability density $\varphi(x, y)$, the conditional distributions are given by

$$P[\xi \le x \mid \eta = y] = \frac{\int_{-\infty}^{x} \varphi(z, y)\, dz}{\int_{-\infty}^{\infty} \varphi(z, y)\, dz} \tag{A.37}$$

and

$$P[\eta \le y \mid \xi = x] = \frac{\int_{-\infty}^{y} \varphi(x, z)\, dz}{\int_{-\infty}^{\infty} \varphi(x, z)\, dz} \tag{A.38}$$

By writing (A.37) as

$$P[\xi \le x \mid \eta = y] = \int_{-\infty}^{x} \left(\frac{\varphi(z,y)}{\int_{-\infty}^{\infty} \varphi(z,y)\,dz} \right) dz$$

one may interpolate the integrand in the above expression as the *conditional distribution density* and denote it as $\varphi(x \mid y)$:

$$\varphi(x \mid y) = \frac{\varphi(x,y)}{\int_{-\infty}^{\infty} \varphi(z,y)\,dz} \tag{A.39}$$

In this way, (A.37) becomes

$$P[\xi \le x \mid \eta = y] = \int_{-\infty}^{x} \varphi(z \mid y)\,dz$$

Similarly, by defining the conditional distribution density

$$\varphi(y \mid x) = \frac{\varphi(x,y)}{\int_{-\infty}^{\infty} \varphi(x,z)\,dz} \tag{A.40}$$

(A.38) can be expressed as

$$P[\eta \le y \mid \xi = x] = \int_{-\infty}^{y} \varphi(z \mid x)\,dz \qquad \blacksquare$$

A.1.4 Gaussian Distribution

Also known as the *normal distribution*, the Gaussian distribution played an extremely important role in the development of probability theory, and is arguably the most useful among all probabilistic distributions. The life span of bulbs produced under practically same manufacturing conditions, for example, obeys a Gaussian distribution. This is also true for several other measures of products that are manufactured in quantity under the same conditions. Gaussian distribution is also encountered in many natural, biological, and social events/phenomena: velocities of gas molecules; errors in measuring a physical object; heights/weights of biological species, yearly precipitations of a certain city, etc. A common thread of these random variables is that they are cumulative synthesis of many small (i.e. of minor importance), independent random components.

A. <u>One-Dimensional Gaussian Distribution</u>

<u>Definition A.16</u> *Gaussian distribution*

A one-dimensional Gaussian distribution is a continuous-type distribution whose density is given by

$$\varphi_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{A.41}$$

where μ and σ are two real-valued parameters. Hence the Gaussian distribution function is

given by

$$\Phi_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(y-\mu)^2/2\sigma^2} dy \tag{A.42}$$

A common notation for the one-dimensional normal distribution is $\mathcal{N}(x, \mu, \sigma^2)$. ■

By (A.24) and (A.28), the expectation and variance of a Gaussian random variable are found to be

$$E[x] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} x e^{-(x-\mu)^2/2\sigma^2} dx = \mu \tag{A.43}$$

and

$$\text{var}[x] = E[x^2] - E[x]^2 = \sigma^2 \tag{A.44}$$

respectively. In other words, we see that the Gaussian distribution is characterized by its expectation and variance. Fig. A.4 illustrates the meaning of μ as the mean value of x using two Gaussian density functions with different μ's, while Fig. A.5 illustrates the meaning of σ^2 as the variance of x by several Gaussian density functions with different $\sigma's$.

Example A.11

Let x be a Gaussian random variable with $\mu = 2$ and $\sigma = 1$. Compute the probability of x being in between $\mu - 3$ and $\mu + 3$ (i.e. x falls into the interval [–1, 5]).

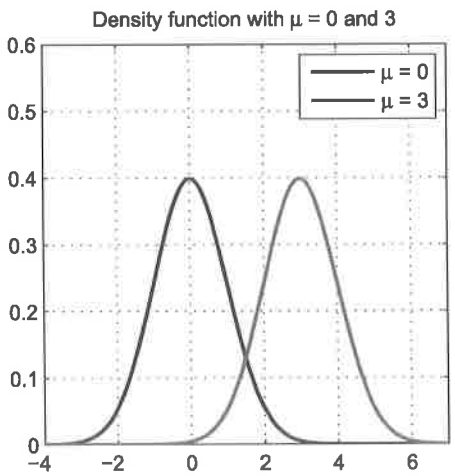


Figure A.4

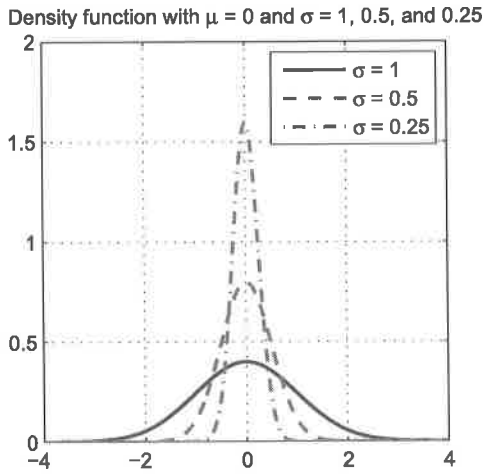


Figure A.5

Solution

From (A.14) and (A.42) it follows that

$$P[-1 \le \xi \le 5] = \frac{1}{\sqrt{2\pi}} \int_{-1}^{5} e^{-(y-2)^2/2} dy = \frac{1}{\sqrt{2\pi}} \int_{-3}^{3} e^{-z^2/2} dz \approx 99.7\% \qquad \blacksquare$$

B. Multidimensional Gaussian Distribution

Let us consider n random variables $\xi_1(\omega), \cdots, \xi_n(\omega)$. The joint distribution of these random variables is defined by

$$\Phi(x_1, x_2, \cdots, x_n) = P[\xi_1(\omega) \le x_1, \xi_2(\omega) \le x_2, \cdots, \xi_n(\omega) \le x_n] \tag{A. 45}$$

If $\xi_1(\omega), \cdots, \xi_n(\omega)$ are of continuous type, then $\Phi(x_1, x_2, \cdots, x_n)$ can be expressed as

$$\Phi(\boldsymbol{x}) = \int_{-\infty}^{x_1} \cdots \int_{-\infty}^{x_n} \varphi(y_1, \cdots, y_n) dy_n \cdots dy_1 \tag{A.46}$$

where $\boldsymbol{x} = [x_1 \quad x_2 \quad \cdots \quad x_n]^T$ and $\varphi(x_1, \cdots, x_n)$ is the density function.

As expected, the most important continuous-type multidimensional distribution is Gaussian distribution whose density in matrix notation is given by

$$\varphi(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{ -\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \right\} \tag{A.47}$$

where $\boldsymbol{\Sigma}$ is a symmetric and positive definite matrix of size n by n, called *covariance matrix*, and $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$. Oftentimes the density of normal distribution is denoted by $\mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$. It can be verified that the expectation of Gaussian $\boldsymbol{\xi}$ is $\boldsymbol{\mu}$, and the covariance of between the individual components (as random variables themselves) of $\boldsymbol{\xi}$ is given by $\boldsymbol{\Sigma}$.

An important special case of multidimensional Gaussian distribution is when the individual random variables are mutually independent, the density function in this case becomes separable as

$$\varphi(x_1, \cdots, x_n) = \varphi_1(x_1) \cdots \varphi(x_n)$$

with each $\varphi_i(x_i)$ being a one-dimensional Gaussian density, i.e.,

$$\varphi_i(x_i) = \frac{1}{\sigma_i \sqrt{2\pi}} e^{-(x_i - \mu_i)^2 / 2\sigma_i^2}$$

Therefore the probability density of n mutually independent Gaussian distribution is given by

$$\varphi(x_1, \cdots, x_n) = \prod_{i=1}^{n} \frac{1}{\sigma_i \sqrt{2\pi}} e^{-(x_i - \mu_i)^2 / 2\sigma_i^2} = \frac{1}{(2\pi)^{n/2}} \frac{1}{\sigma_1 \cdots \sigma_n} \exp\left(-\frac{1}{2} \sum_{i=1}^{n} \frac{(x_i - \mu_i)^2}{\sigma_i^2} \right) \tag{A.48}$$

It can be readily verified that (A.47) coincides with (A.48) by assigning $\boldsymbol{\mu} = [\mu_1 \quad \cdots \quad \mu_n]$ and

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{bmatrix}$$

In other words, (A.47) becomes joint density of n independent Gaussian distributions if and only if Σ is *diagonal*.

A.1.5 Likelihood Function and Log-Likelihood

Consider a continuous-type distribution with density $\varphi(x, \boldsymbol{w})$ where vector $\boldsymbol{w}$ collects the parameters involved. Let $\mathcal{D}$ be a data set $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ that are drawn independently from the same distribution. In literature, a data set of this type is said to be *independently and identically distributed* (i.i.d.). Because of its probabilistic independence, given parameter $\boldsymbol{w}$ the probability density of an i.i.d. data set $\mathcal{D}$ assumes the form

$$p(\mathcal{D} \mid \boldsymbol{w}) = \prod_{i=1}^{N} \varphi(x_i, \boldsymbol{w}) \tag{A.49}$$

which is called *likelihood function* of the probability distribution. A popular approach for determining the parameters of a probability distribution using an observed data is to maximize the likelihood function with respect to the parameters. Because logarithm is a monotonically increasing function and logarithm of the likelihood function simplifies subsequent mathematical analysis, one maximizes the log-likelihood instead:

$$\log p(\mathcal{D} \mid \boldsymbol{w}) = \sum_{i=1}^{n} \log \varphi(x_i, \boldsymbol{w}) \tag{A.50}$$

Example A.12

Consider the 1-D Gaussian distribution in (A.41), i.e.

$$\varphi_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

Following (A.50), its log-likelihood function for an observed i.i.d. data set, denoted by $L(\mu, \sigma^2, \mathcal{D})$, is given by

$$L(\mu, \sigma^2, \mathcal{D}) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - \mu)^2 - \frac{N}{2}\log\sigma^2 - \frac{N}{2}\log(2\pi)$$

Maximizing $L(\mu, \sigma^2, \mathcal{D})$ with respect to $\{\mu, \sigma^2\}$ yields *maximum likelihood* (ML) estimates

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{A.51}$$

and

$$\sigma_{ML}^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_{ML})^2 \qquad \blacksquare \qquad \text{(A.52)}$$

Appendix B Optimization Methods and Software

B.1.1 Unconstrained Optimization [5]

Unconstrained optimization studies problems of the form

$$\underset{x \in R^n}{\text{minimize}} \quad f(\boldsymbol{x}) \tag{B.1}$$

where $f(\boldsymbol{x})$ is a read-valued twice continuously differentiable function.

- Gradient of $f(\boldsymbol{x})$ is defined by

$$\nabla f(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix}$$

- Hessian of $f(\boldsymbol{x})$ is defined by

$$\nabla^2 f(\boldsymbol{x}) = \boldsymbol{H}(\boldsymbol{x}) = \left\{ \frac{\partial^2 f}{\partial x_i \partial x_j} \right\}_{i,j=1,2,\ldots,n}$$

Note that Hessian is always square and symmetric.

- A point $\boldsymbol{x}$ is called a *stationary point* if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$. A geometric interpretation of the concept in the case of n = 1 or 2 is that $\boldsymbol{x}$ is a stationary point if the tangent (plane) of $f(\boldsymbol{x})$ at $\boldsymbol{x}$ is in parallel with the $\boldsymbol{x}$-axis (plane).

A. First-Order Necessary Condition

If $\boldsymbol{x}^*$ is a local minimum point (minimizer), then

$$\nabla f(\boldsymbol{x}^*) = \boldsymbol{0} \tag{B.2}$$

In other words, if $\boldsymbol{x}^*$ is a minimum point, then it must be a stationary point.

◊ A geometric interpretation of the 1st-order necessary condition when n = 1 or 2 is that the tangent plane of $f(\boldsymbol{x})$ at a minimizer must be in parallel to the $\boldsymbol{x}$-axis (plane).

◊ $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ is not sufficient for $\boldsymbol{x}^*$ to be a local minimizer.

B. Second-Order Necessary Conditions

If $\boldsymbol{x}^*$ is a local minimum point (minimizer), then

$$\nabla f(\boldsymbol{x}^*) = \mathbf{0} \quad \text{and} \quad \nabla^2 f(\boldsymbol{x}^*) \succeq \mathbf{0} \tag{B.3}$$

◊ $\nabla^2 f(\boldsymbol{x}^*) \succeq \mathbf{0}$ (i.e. the Hessian $\nabla^2 f(\boldsymbol{x}^*)$ is positive semidefinite if and only if the eigenvalues of $\nabla^2 f(\boldsymbol{x}^*)$ are nonnegative.

◊ $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$ and $\nabla^2 f(\boldsymbol{x}^*) \succeq \mathbf{0}$ are not sufficient for $\boldsymbol{x}^*$ to be a local minimizer.

C. ***Second-Order Sufficient Conditions***

Point $\boldsymbol{x}^*$ is a local minimizer, if

$$\nabla f(\boldsymbol{x}^*) = \mathbf{0} \quad \text{and} \quad \nabla^2 f(\boldsymbol{x}^*) \succ \mathbf{0} \tag{B.4}$$

D. ***Convex Sets and Convex Functions***

- A set $S \subseteq R^n$ is said to be convex if

$$\boldsymbol{x}_1, \boldsymbol{x}_2 \in S \Rightarrow \alpha \boldsymbol{x}_1 + (1-\alpha)\boldsymbol{x}_2 \in S \text{ for } 0 \le \alpha \le 1 \tag{B.5}$$

- A function $f(\boldsymbol{x})$ is said to be convex over convex set S if

$$f(\alpha \boldsymbol{x}_1 + (1-\alpha)\boldsymbol{x}_2) \le \alpha f(\boldsymbol{x}_1) + (1-\alpha) f(\boldsymbol{x}_2) \tag{B.6}$$

  for any $\boldsymbol{x}_1, \boldsymbol{x}_2 \in S$ and $0 \le \alpha \le 1$.

- A function $f(\boldsymbol{x}) \in C^1$ (i.e. continuously differentiable) is convex if and only if

$$f(\boldsymbol{x}_1) \ge f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T (\boldsymbol{x}_1 - \boldsymbol{x}) \tag{B.7}$$

for any $\boldsymbol{x}, \boldsymbol{x}_1 \in S$. Property (B.7) is an important one as it actually characterizes convex functions and has a geometrical interpretation – all tangents of $f(\boldsymbol{x})$ lie below the function graph $(\boldsymbol{x}, f(\boldsymbol{x}))$ as illustrated in Fig. B.1.

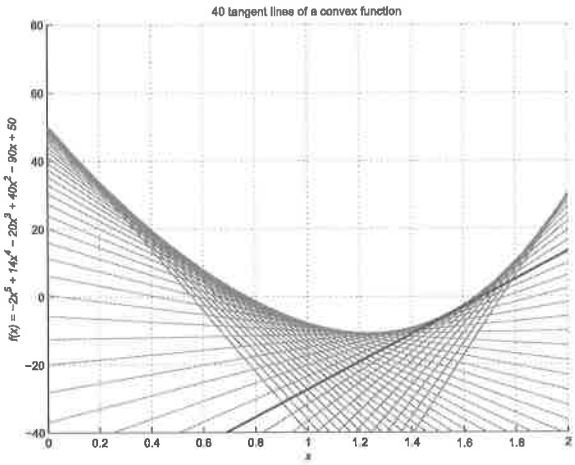


Figure B.1 All supporting tangents of a convex function lie underneath it.

- A function $f(\boldsymbol{x}) \in C^2$ (i.e. twice continuously differentiable) is convex over a convex set S (which contains more than one point) if and only if $\nabla^2 f(\boldsymbol{x}) \succeq \mathbf{0}$ for $\boldsymbol{x} \in S$.

- If function $f(\boldsymbol{x}) \in C^1$ is convex over a convex set S and $\boldsymbol{x}^*$ is a point in S that satisfies $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$, then $\boldsymbol{x}^*$ is a global minimizer in S. This is because for any $\boldsymbol{x} \in S$ we have

$$f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x}^*)^T (\boldsymbol{x} - \boldsymbol{x}^*) = f(\boldsymbol{x}^*)$$

◊ In other words, for smooth convex functions, the first-order necessary condition $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$ is sufficient for $\boldsymbol{x}^*$ to be a minimizer.

E. ***A General Structure of Unconstrained Optimization Algorithms***

Step 1: Choose an initial point $\boldsymbol{x}_0$ and a convergence tolerance ε. Set a counter $k = 0$ for the number of iterations.

Step 2: Determine a search direction $\boldsymbol{d}_k$ for reducing $f(\boldsymbol{x})$ from $f(\boldsymbol{x}_k)$.

Step 3: Determine a step size $\alpha_k \geq 0$ such that $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ is minimized, and construct

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k .$$

Step 4: If $\|\alpha_k \boldsymbol{d}_k\| < \varepsilon$, stop and output a solution $\boldsymbol{x}_{k+1}$, otherwise set $k := k + 1$ and repeat from Step 2.

F. ***A Backtracking Algorithm for Line Search (BLS)*** [6]

Assume $\boldsymbol{d}_k$ is a descent direction of $f(\boldsymbol{x})$ at $\boldsymbol{x}_k$.

Step 1: Select constants $\rho \in (0, 0.5)$ and $\gamma \in (0,1)$ (e.g. $\rho = 0.1$, $\gamma = 0.5$). Set $\alpha = 1$.

Step 2: While

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) > f(\boldsymbol{x}_k) + \rho \alpha \boldsymbol{g}_k^T \boldsymbol{d}_k \tag{B.8}$$

Set $\alpha = \gamma\alpha$.

Step 3: Output $\alpha_k = \alpha$.

◊ How BLS works: the right-hand side of (B.8) is a line in α passing through point (0, $f(\boldsymbol{x}_k)$) with a negative slop because $\boldsymbol{d}_k$ is a descent direction:

$$\rho \boldsymbol{g}_k^T \boldsymbol{d}_k = \rho \left. \frac{df(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)}{d\alpha} \right|_{\alpha=0} < 0$$

See Fig. B.2 for an illustration. Note from the figure that a value of α less than α_0 may be considered acceptable because such an α satisfies

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) < f(\boldsymbol{x}_k) + \rho \alpha \boldsymbol{g}_k^T \boldsymbol{d}_k < f(\boldsymbol{x}_k)$$

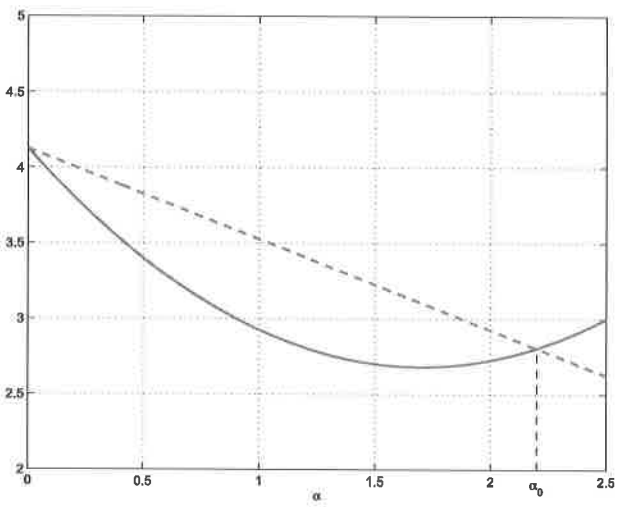


Figure B.2 Both $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ (blue curve) and $f(\boldsymbol{x}_k) + \rho\alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$ (red dashed line) are shown as functions of α. An acceptable α is found by gradually reducing α to a value less than α_0 (which is signified by $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) < f(\boldsymbol{x}_k) + \rho \alpha \boldsymbol{g}_k^T \boldsymbol{d}_k$).

• Two MATLAB functions, namely `bt_lsearch.m` and `inex_lsearch.m`, for line search are available from the link below:

http://www.ece.uvic.ca/~wslu/Talk.html

G. *The Steepest Descent Method (SDM)*

- The SDM was originated by A. Cauchy (1848). It remains to be one of the most popular minimization techniques because its simplicity although the algorithm is usually slower relative to another popular algorithm –Newton's algorithm.

- The SMD follows the general algorithmic structure described above, where the search direction is computed as

$$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) \tag{B.9}$$

This choice of $\boldsymbol{d}_k$ may be understood by the Taylor expansion of $f(\boldsymbol{x})$ at $\boldsymbol{x}_k$ up to its 1st-order:

$$f(\boldsymbol{x}_k + \boldsymbol{d}) \approx f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} \quad \text{(assuming } \|\boldsymbol{d}\| \text{ is small)}$$

From which we see the value of $f(\boldsymbol{x})$ is reduced at highest rate at $\boldsymbol{x}_k + \boldsymbol{d}_k$ if $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$ is taken.

H. *Why Is SDM Slow?*

The slow convergence of the SDM has to do with the "condition" of the Hessian at $\boldsymbol{x}_k$: the progress made by the kth SDM iteration relative to the preceding progress can be estimated as

$$f(x_{k+1}) - f(x^*) \le \left(\frac{1-r}{1+r}\right)^2 \left(f(x_k) - f(x^*)\right) \tag{B.10}$$

where r is the ratio of the smallest over largest eigenvalues of the Hessian at $\boldsymbol{x}_k$ (which is denoted by $\boldsymbol{H}_k$). From (B.10) the SDM is expected to be slow when ratio r is small, or stated in another way, when the *condition number* of $\boldsymbol{H}_k$ is large.

◊ The condition number of a square nonsingular matrix is the ratio of its largest over smallest eigenvalues. A square nonsingular matrix is said to be *ill-conditioned* if its condition number is very large.

I. *Scaling and Normalization*

The eigenvalues of Hessian of an objective function and, in turn, the performance of the steepest-descent method tend to depend to a large extent on the choice of variables. For example, in one and the same two-dimensional problem, whether the contours look nearly circular or extremely elliptical has to do with how the units of design components $\{x_i,\ i = 1,\ldots, n\}$ are chosen. Consequently, the rate of convergence can often be improved by scaling the variables properly through variable transformation. A possible approach to scaling is to let

$$\boldsymbol{x} = \boldsymbol{T}\boldsymbol{y}$$

where $\boldsymbol{T}$ is an $n \times n$ diagonal matrix, and then solve the problem

$$\underset{\boldsymbol{y}}{\text{minimize}} \quad h(\boldsymbol{y}) = f(\boldsymbol{x})\big|_{\boldsymbol{x}=\boldsymbol{T}\boldsymbol{y}}$$

The gradient and Hessian of the new problem are calculated as

$$\boldsymbol{g}_h = \boldsymbol{T}^T \boldsymbol{g}_x \text{ and } \boldsymbol{H}_h = \boldsymbol{T}^T \boldsymbol{H}\boldsymbol{T} \tag{B.11}$$

respectively. We note that both the steepest-descent direction and the eigenvalues associated with the transformed problem are altered, hence the performance of an SDM is expected to be improved by adequate scaling.

The choice of $\boldsymbol{T}$ tends to depend heavily on the problem at hand and, as a result, no general rules can be stated. For many problems in machine learning, however, the variables are always related to real-world quantities (e.g. number of rooms in a typical house, age of a house, total living area of a house in feet^2, etc.) and are known to vary in certain ranges. In cases like these, it is often beneficial to normalize each variable x, assuming to be in the range $[a, b]$, to a standard range say $[-0.5, 0.5]$, by linear transform

$$\bar{x} = \frac{x-\mu}{b-a} \quad \text{with} \quad \mu = \frac{a+b}{2} \tag{B.12}$$

J. Newton's Method

Newton's method is well known for its fast convergence at a cost of increased complexity.

- Newton's method also follows the general algorithmic structure as outlined in Sec. B.1.1.E, where the search direction is computed as

$$\boldsymbol{d}_k = -\left(\nabla^2 f(\boldsymbol{x}_k)\right)^{-1} \nabla f(\boldsymbol{x}_k), \quad \text{i.e.,} \quad \boldsymbol{d}_k = -\boldsymbol{H}_k^{-1} \boldsymbol{g}_k \tag{B.13}$$

◊ This choice of $\boldsymbol{d}_k$ may be understood as the minimizer of the 2nd-order approximate of $f(\boldsymbol{x})$ in a small vicinity of $\boldsymbol{x}_k$:

$$f(\boldsymbol{x}_k + \boldsymbol{d}) \approx f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} + \tfrac{1}{2} \boldsymbol{d}^T \nabla^2 f(\boldsymbol{x}_k) \boldsymbol{d}$$

Figure B.3 illustrates the first two Newton steps for the convex function

$$f(x) = x^2 + 0.05e^{-x} + 8$$

With $x_0 = -5$, the first two Newton iterations yield $x_1 = -3.15$ and $x_2 = -0.7929$ while the global minimizer is at $x^* = 0.0242$.

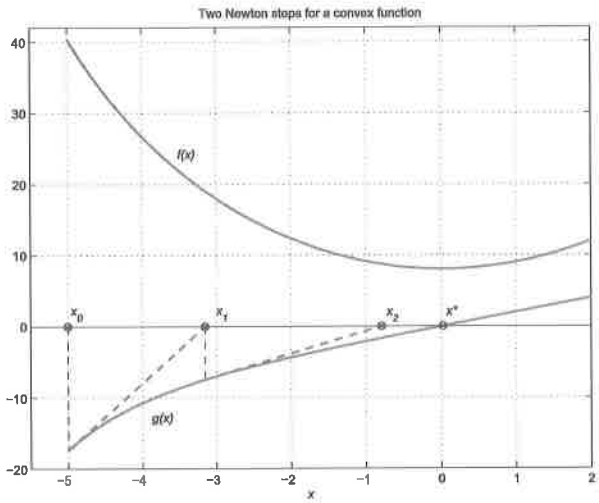


Figure B.3 Two Newton steps for a convex function.

- It is important to note that Newton's direction given by (1.65) is effective only when $\boldsymbol{H}_k$ is *positive definite*. In case $\boldsymbol{H}_k$ is not positive definite, the search direction in Newton's method is modified to

$$\boldsymbol{d}_k = -\hat{\boldsymbol{H}}_k^{-1}\boldsymbol{g}_k$$

where $\hat{\boldsymbol{H}}_k = \dfrac{\boldsymbol{H}_k + \beta \boldsymbol{I}}{1+\beta}$ with a sufficiently large β so that $\boldsymbol{H}_k + \beta \boldsymbol{I} \succ \boldsymbol{0}$.

K. Quasi-Newton Methods

One may take a unifying look at the SDM and Newton method by writing their search direction as

$$\boldsymbol{d}_k = -\boldsymbol{S}_k \boldsymbol{g}_k \tag{B.14}$$

where

$$\boldsymbol{S}_k = \begin{cases} \boldsymbol{I} & \text{for steepest descent} \\ \boldsymbol{H}_k^{-1} & \text{for Newton} \end{cases}$$

Newton's method is known for its fast convergence and solution accuracy. However, this fast convergence is achieved at the cost of increased computational complexity which has largely to do with the computation of the inverse Hessian in every Newton iteration. Quasi-Newton algorithms are developed to provide convergent rates comparable with that of Newton algorithm with reduced complexity. Actually quasi-Newton algorithms are based only on gradient information and do not require explicit evaluation of the Hessian and its inverse.

Quasi-Newton algorithms follow the general algorithmic structure described earlier, where search direction $\boldsymbol{d}_k$ is evaluated using (B.14) where matrix $\boldsymbol{S}_k$ is updated in each iteration using gradient information. Two most well-known quasi-Newton algorithms are those developed by Davidon, Fletcher, and Powell (DFP), and by Broyden, Fletcher, Goldfarb, and Shanno (BFGS),

respectively. The BFGS and DFP formulas for updating matrix $\boldsymbol{S}_k$ are given below.

◊ Quantities required: $\boldsymbol{\delta}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \quad \boldsymbol{\gamma}_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$

◊ Davidon-Fletcher-Powell (DFP) updating formula: $\boldsymbol{S}_0 = \boldsymbol{I}$, and

$$\boldsymbol{S}_{k+1} = \boldsymbol{S}_k + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\delta}_k^T \boldsymbol{\gamma}_k} - \frac{\boldsymbol{S}_k \boldsymbol{\gamma}_k \boldsymbol{\gamma}_k^T \boldsymbol{S}_k}{\boldsymbol{\gamma}_k^T \boldsymbol{S}_k \boldsymbol{\gamma}_k} \tag{B.15}$$

◊ Broyden-Fletcher-Goldfarb-Shanno (BFGS) updating formula: $\boldsymbol{S}_0 = \boldsymbol{I}$, and

$$\boldsymbol{S}_{k+1} = \boldsymbol{S}_k + \left(1 + \frac{\boldsymbol{\gamma}_k^T \boldsymbol{S}_k \boldsymbol{\gamma}_k}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T \boldsymbol{S}_k + \boldsymbol{S}_k \boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \tag{B.16}$$

B.1.2 Constrained Optimization

A general constrained optimization problem assumes the form

$$\begin{aligned} &\text{minimize} \quad f(\boldsymbol{x}) \\ &\text{subject to}: \; a_i(\boldsymbol{x}) = 0 \quad \text{for } i = 1,2,\ldots,p \\ &\qquad\qquad\quad c_j(\boldsymbol{x}) \le 0 \quad \text{for } j = 1,2,\ldots,q \end{aligned} \tag{B.17a-c}$$

◊ Feasible region: $\{\boldsymbol{x} : a_i(\boldsymbol{x}) = 0 \text{ for } i = 1,2,\ldots,p, \text{ and } c_j(\boldsymbol{x}) \le 0 \text{ for } j = 1,2,\ldots,q\}$

◊ A point $\boldsymbol{x}$ is said to be *feasible* if it is in the feasible region (inside or on the boundary).

◊ An inequality constraint $c_j(\boldsymbol{x}) \le 0$ is said to be *active* at a feasible point $\boldsymbol{x}$, if $c_j(\boldsymbol{x}) = 0$. An inequality constraint $c_j(\boldsymbol{x}) \le 0$ is said to be *inactive* at a feasible point $\boldsymbol{x}$ if $c_j(\boldsymbol{x}) < 0$.

A. Karush-Kuhn-Tucker (KKT) Conditions

The Karush-Kuhn-Tucker (KKT) conditions are a set of *first-order necessary conditions* for point $\boldsymbol{x}^*$ to be a local minimizer of problem (B.17). These conditions are fundamentally important in the theory and practice of constrained optimization. The KKT conditions can be stated as follows.

KKT Conditions

If $\boldsymbol{x}^*$ is a local minimizer of the constrained problem (1.69), then the following conditions, known as *KKT conditions*, are satisfied:

(a) $$a_i(\boldsymbol{x}^*) = 0 \quad \text{for } i = 1,2,\ldots,p \tag{B.18a}$$

(b) $$c_j(\boldsymbol{x}^*) \le 0 \quad \text{for } j = 1,2,\ldots,q \tag{B.18b}$$

(c) there exist Lagrange multipliers λ_i^* for $1 \le i \le p$ and μ_j^* for $1 \le j \le q$ such that

$$\nabla f(\boldsymbol{x}^*) + \sum_{i=1}^{p} \lambda_i^* \nabla a_i(\boldsymbol{x}^*) + \sum_{j=1}^{q} \mu_j^* \nabla c_j(\boldsymbol{x}^*) = \boldsymbol{0} \tag{B.18c}$$

(d) $$\mu_j^* c_j(\boldsymbol{x}^*) = 0 \quad \text{for } j = 1,2,\ldots,q \tag{B.18d}$$

(e) $$\mu_j^* \ge 0 \quad \text{for } j = 1,2,\ldots,q \tag{B.18e}$$

◊ Conditions (B.18a) and (B.18b) are merely the constraints imposed in the problem, see (B.17b) and (B.17c).

◊ Condition (B.18c) states that at a local minimizer the gradient of the objective function is a linear combination of the gradients of the constraint functions, and the coefficients in the combination are (up to a sign) Lagrange multipliers .

◊ If we define the *Lagrangian* of problem (B.17) as

$$L(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu}) = f(\boldsymbol{x}) + \sum_{i=1}^{p} \lambda_i a_i(\boldsymbol{x}) + \sum_{j=1}^{q} \mu_j c_j(\boldsymbol{x}) \tag{B.19}$$

then (B.18c) can be expressed as

$$\nabla_x L(\boldsymbol{x}^*,\boldsymbol{\lambda}^*,\boldsymbol{\mu}^*) = \mathbf{0} \tag{B.20}$$

◊ Conditions in (B.18d) are called *complementarity conditions*. It follows that if the kth inequality constraint is inactive at $\boldsymbol{x}^*$, then $\mu_k^* = 0$. In other words, the equation in (B.18c) only includes those terms $\nabla c_j(\boldsymbol{x}^*)$ with $c_j(\boldsymbol{x})$ active at $\boldsymbol{x}^*$.

◊ The total number of equations in the KKT conditions, namely p (from (B.18a)) + n (from (B.18c)) + q (from (B.18d)) , are equal to the total number of components in unknown vectors $\{\boldsymbol{x}^*,\boldsymbol{\lambda}^*,\boldsymbol{\mu}^*\}$.

B. ***Convex Programming Problems***

Definition B.1 *Convex programming*

A constrained minimization problem (B.17) is said to be a *convex programming* (CP) problem if the objective function is convex and the feasible region defined by the constraints, i.e., $\{\boldsymbol{x}: a_i(\boldsymbol{x}) = 0$ for $i = 1, 2, \ldots, p$, and $c_j(\boldsymbol{x}) \leq 0$ for $j = 1, 2, \ldots, q\}$ is a convex set. In words, a CP problem is a problem of minimizing a convex function over a convex region. ■

- It can be shown that the feasible region $\{\boldsymbol{x}: a_i(\boldsymbol{x}) = 0$ for $i = 1, 2, \ldots, p$, and $c_j(\boldsymbol{x}) \leq 0$ for $j = 1, 2, \ldots, q\}$ is convex if (i) the equality constraint functions $a_i(\boldsymbol{x})$ are *affine*, and (ii) the inequality constraint functions $c_j(\boldsymbol{x})$ are *convex*.

Hence a general CP problem assumes the form

$$\begin{aligned} &\text{minimize} \quad f(\boldsymbol{x}) \\ &\text{subject to}: \ \boldsymbol{a}_i^T \boldsymbol{x} = b_i \quad \text{for } i = 1,2,\ldots,p \\ &\qquad\qquad\quad c_j(\boldsymbol{x}) \leq 0 \quad \text{for } j = 1,2,\ldots,q \end{aligned} \tag{B.21a-c}$$

where $f(\boldsymbol{x})$ and $c_j(\boldsymbol{x})$ for $j = 1, 2, \ldots, q$ are convex.

- An important property of CP problems is that the KKT conditions become *both necessary and sufficient*. Therefore, for CP problems $\{\boldsymbol{x}^*,\boldsymbol{\lambda}^*,\boldsymbol{\mu}^*\}$ satisfying the KKT conditions gives a local solution (minimizer) $\boldsymbol{x}^*$.

- Another important property of CP problems is that any local minimizer of a CP problem is a global minimizer. By combining the above two properties, one concludes that x^* is a global minimizer of CP problem (B.21) if and only if the KKT conditions are satisfied.

C. *Important Classes of CP Problems*

- ***Linear programming*** (*LP*)

Linear programming problems are of great importance both theoretically and practically. Many problems in science and engineering can be accurately or approximately modeled as LP problems.

The *standard-form* LP problem is given by

$$\begin{aligned} &\text{minimize} \quad f(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \\ &\text{subject to:} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ &\qquad\qquad\quad \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{B.22a-c}$$

Another class of popular LP problem know as *alternative-form* LP problems, assumes the form

$$\begin{aligned} &\text{minimize} \quad f(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \\ &\text{subject to:} \quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \end{aligned} \tag{B.23a-b}$$

- ***Quadratic programming*** (*QP*)

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{H}\boldsymbol{x} + \boldsymbol{p}^T\boldsymbol{x} \\ &\text{subject to:} \quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \end{aligned} \tag{B.24a-b}$$

where $\boldsymbol{H}$ is symmetric and positive semidefinite ($\boldsymbol{H} \succeq \boldsymbol{0}$).

- ***QP with quadratic constraints***

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{H}\boldsymbol{x} + \boldsymbol{p}^T\boldsymbol{x} \\ &\text{subject to:} \quad \boldsymbol{x}^T\boldsymbol{Q}_i\boldsymbol{x} + \boldsymbol{q}_i^T\boldsymbol{x} + r_i \leq 0 \quad \text{for } 1 \leq i \leq q \end{aligned} \tag{B.25a-b}$$

where $\boldsymbol{H}$ and $\boldsymbol{Q}_i$ are symmetric and positive semidefinite.

- ***Semidefinite programming*** (*SDP*)

◊ **Primal SDP**

$$\begin{aligned} &\text{minimize} \quad \boldsymbol{C}\cdot\boldsymbol{X} \\ &\text{subject to:} \quad \boldsymbol{A}_i\cdot\boldsymbol{X} = b_i \quad \text{for } 1 \leq i \leq p \\ &\qquad\qquad\quad \boldsymbol{X} \succeq \boldsymbol{0} \end{aligned} \tag{B.26a-c}$$

where variable $\boldsymbol{X}$ and data $\boldsymbol{A}_i$ and $\boldsymbol{C}$ are symmetric matrices and $\boldsymbol{C}\cdot\boldsymbol{X}$ denotes the inner product in the space of real symmetric matrices defined by

$$\boldsymbol{C}\cdot\boldsymbol{X} = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}x_{i,j}$$

◊ **Dual SDP**

$$\begin{aligned} &\text{minimize} && \boldsymbol{c}^T\boldsymbol{x} \\ &\text{subject to:} && \boldsymbol{F}(\boldsymbol{x}) \succeq \boldsymbol{0} \end{aligned} \tag{B.27a-b}$$

where $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{F}_0 + \sum_{i=1}^{p} x_i \boldsymbol{F}_i$.

- ***Second-order cone programming*** (*SOCP*)

$$\begin{aligned} &\text{minimize} && \boldsymbol{b}^T\boldsymbol{x} \\ &\text{subject to:} && \left\| \boldsymbol{A}_j^T\boldsymbol{x} + \boldsymbol{c}_j \right\| \le \boldsymbol{b}_j^T\boldsymbol{x} + d_j \quad \text{for } 1 \le j \le q \end{aligned} \tag{B.28a-b}$$

D. ***Lagrange Dual*** [6]

Duality is an important concept associated with constrained convex problems. To study the Lagrange dual, we need a concept called *Lagrange dual function*. Consider the general CP problem

$$\begin{aligned} &\text{minimize} && f(\boldsymbol{x}) \\ &\text{subject to:} && \boldsymbol{a}_i^T\boldsymbol{x} = b_i \quad \text{for } 1 \le i \le p \\ & && c_j(\boldsymbol{x}) \le 0 \quad \text{for } 1 \le j \le q \end{aligned} \tag{B.29a-c}$$

where $f(\boldsymbol{x})$ and $c_j(\boldsymbol{x})$ are convex and recall its Lagrangian

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\boldsymbol{x}) + \sum_{i=1}^{p} \lambda_i \left(\boldsymbol{a}_i^T\boldsymbol{x} - b_i \right) + \sum_{j=1}^{q} \mu_i c_j(\boldsymbol{x})$$

Definition B.2 *Lagrange dual function*

The *Lagrange dual function* of problem (B.29) is defined as

$$q(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{B.30}$$

for $\boldsymbol{\lambda} \in R^p$ and $\boldsymbol{\mu} \in R^q$ with $\boldsymbol{\mu} \ge \boldsymbol{0}$. Note that the Lagrangian is *convex* with respect to $\boldsymbol{x}$. If $L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ is unbounded from below for some $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$, then the value of $q(\boldsymbol{\lambda}, \boldsymbol{\mu})$ is assigned to $-\infty$.

Property 1 $q(\boldsymbol{\lambda}, \boldsymbol{\mu})$ is a concave function with respect to $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$. The property follows from that fact that for $\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2 \in R^p$ and $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2 \in R^q$ with $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2 \ge \boldsymbol{0}$ and for $t \in (0,1)$, we have

$$q(t\boldsymbol{\lambda}_1+(1-t)\boldsymbol{\lambda}_2, t\boldsymbol{\mu}_1+(1-t)\boldsymbol{\mu}_2) = \inf_{\boldsymbol{x}} L(\boldsymbol{x}, t\boldsymbol{\lambda}_1+(1-t)\boldsymbol{\lambda}_1, t\boldsymbol{\mu}_1+(1-t)\boldsymbol{\mu}_2)$$

$$= \inf_{\boldsymbol{x}}\left[(t+1-t)f(\boldsymbol{x})+\sum_{i=1}^{p}(t\lambda_{1,i}+(1-t)\lambda_{2,i})(a_i^T\boldsymbol{x}-b_i)+\sum_{j=1}^{q}(t\mu_{1,j}+(1-t)\mu_{2,j})c_j(\boldsymbol{x})\right]$$

$$\geq t\cdot\inf_{\boldsymbol{x}}\left[f(\boldsymbol{x})+\sum_{i=1}^{p}\lambda_{1,i}(a_i^T\boldsymbol{x}-b_i)+\sum_{j=1}^{q}\mu_{1,j}c_j(\boldsymbol{x})\right]+(1-t)\cdot\inf_{\boldsymbol{x}}\left[f(\boldsymbol{x})+\sum_{i=1}^{p}\lambda_{2,i}(a_i^T\boldsymbol{x}-b_i)+\sum_{j=1}^{q}\mu_{2,j}c_j(\boldsymbol{x})\right]$$

$$= t\cdot q(\boldsymbol{\lambda}_1,\boldsymbol{\mu}_1)+(1-t)\cdot q(\boldsymbol{\lambda}_2,\boldsymbol{\mu}_2)$$

Definition B.3 *Lagrange dual problem*

The *Lagrange dual problem* with respect to problem (B.29) is defined as

$$\begin{aligned} &\underset{\boldsymbol{\lambda},\boldsymbol{\mu}}{\text{maximize}} \quad q(\boldsymbol{\lambda},\boldsymbol{\mu}) \\ &\text{subject to:} \quad \boldsymbol{\mu}\geq \mathbf{0} \end{aligned} \tag{B.31a-b}$$

Property 2 For any $\boldsymbol{x}$ feasible for problem (B.29) and $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$ feasible for problem (B.31), we have

$$f(\boldsymbol{x})\geq q(\boldsymbol{\lambda},\boldsymbol{\mu}) \tag{B.32}$$

This is because

$$L(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu}) = f(\boldsymbol{x})+\sum_{i=1}^{p}\lambda_i\left(\boldsymbol{a}_i^T\boldsymbol{x}-b_i\right)+\sum_{j=1}^{q}\mu_i c_j(\boldsymbol{x}) = f(\boldsymbol{x})+\sum_{j=1}^{q}\mu_i c_j(\boldsymbol{x})\leq f(\boldsymbol{x})$$

thus

$$q(\boldsymbol{\lambda},\boldsymbol{\mu}) = \inf_{\boldsymbol{x}} L(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu})\leq L(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu})\leq f(\boldsymbol{x})$$

We call the convex minimization problem in (B.29) the *primal* problem and the concave maximization problem in (B.31) the *dual* problem. The property (B.32) leads naturally to the concept of *duality gap* between the primal and dual objectives, which us defined by

$$\delta(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu}) = f(\boldsymbol{x})-q(\boldsymbol{\lambda},\boldsymbol{\mu}) \tag{B.33}$$

It follows that for feasible $\{\boldsymbol{x},\boldsymbol{\lambda}, \boldsymbol{\mu}\}$ the duality gap is always nonnegative.

Property 3 Let $\boldsymbol{x}^*$ be a solution of the primal problem in (B.29). Then the dual function at any feasible $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$ serves as a lower bound of the optimal value of the primal objective, $f(\boldsymbol{x}^*)$, namely,

$$f(\boldsymbol{x}^*)\geq q(\boldsymbol{\lambda},\boldsymbol{\mu}) \tag{B.34}$$

This property follows immediately from (B.32) by taking the minimum of $f(\boldsymbol{x})$ on its left-hand side.

- A question that naturally arises is what is the tightest lower bound the dual function $q(\boldsymbol{\lambda},\boldsymbol{\mu})$

can offer? Obviously the answer is found by maximizing the dual function $q(\boldsymbol{\lambda},\boldsymbol{\mu})$ on the right-hand side of (B.34) subject to $\boldsymbol{\mu} \geq \mathbf{0}$, which is exactly the Lagrange dual problem as formulated in (B.31). Therefore, if we denote the solution of (B.31) by $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$, then we have

$$f(\boldsymbol{x}^*) \geq q(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \tag{B.35}$$

Based on (B.35), we now introduce the concept of *strong* and *weak* duality as follows.

Definition B.4 *Strong and weak duality*

Let $\boldsymbol{x}^*$ and $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ be solutions of primal problem (1.81) and dual problem (1.83), respectively. We say strong duality holds if $f(\boldsymbol{x}^*) = q(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$, i.e., the optimal duality gap is zero; and a weak duality holds if $f(\boldsymbol{x}^*) > q(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$. ■

- It can be shown that if the primal problem is strictly feasible, i.e., there exists $\boldsymbol{x}$ satisfying

$$\begin{aligned} &\boldsymbol{a}_i^T \boldsymbol{x} = b_i \quad \text{for } 1 \leq i \leq p \\ &c_j(\boldsymbol{x}) < 0 \quad \text{for } 1 \leq j \leq q \end{aligned}$$

(this is to say that the interior of the feasible region of problem (B.29) is nonempty), then strong duality holds, i.e., the optimal duality gap is zero.

Example B.1 Find the Lagrange dual of the LP problem

$$\begin{aligned} &\text{minimize} \quad \boldsymbol{c}^T \boldsymbol{x} \\ &\text{subject to:} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \ \boldsymbol{x} \geq \mathbf{0} \end{aligned}$$

Solution We write $\boldsymbol{x} \geq \mathbf{0}$ as $-\boldsymbol{x} \leq \mathbf{0}$ hence the Lagrangian of the LP problem is given by

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \boldsymbol{c}^T \boldsymbol{x} + (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T \boldsymbol{\lambda} - \boldsymbol{x}^T \boldsymbol{\mu}$$

thus

$$q(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\boldsymbol{x}} \left\{ \boldsymbol{c}^T \boldsymbol{x} + (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T \boldsymbol{\lambda} - \boldsymbol{x}^T \boldsymbol{\mu} \right\} = \inf_{\boldsymbol{x}} \left\{ (\boldsymbol{c} + \boldsymbol{A}^T \boldsymbol{\lambda} - \boldsymbol{\mu})^T \boldsymbol{x} - \boldsymbol{b}^T \boldsymbol{\lambda} \right\} \tag{B.36}$$

For given $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$ such that $\boldsymbol{c} + \boldsymbol{A}^T \boldsymbol{\lambda} - \boldsymbol{\mu} \neq \mathbf{0}$, by (B.36) we have $q(\boldsymbol{\lambda}, \boldsymbol{\mu}) = -\infty$. Therefore to deal with a well-defined dual function $q(\boldsymbol{\lambda}, \boldsymbol{\mu})$ we assume $\boldsymbol{c} + \boldsymbol{A}^T \boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{0}$ which leads to

$$q(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\boldsymbol{x}} (-\boldsymbol{b}^T \boldsymbol{\lambda}) = -\boldsymbol{b}^T \boldsymbol{\lambda}$$

and the Lagrange dual of the LP problem in question is given by

$$\begin{aligned} &\underset{\boldsymbol{\lambda}, \boldsymbol{\mu}}{\text{maximize}} \quad -\boldsymbol{b}^T \boldsymbol{\lambda} \\ &\text{subject to:} \quad \boldsymbol{\mu} \geq \mathbf{0} \end{aligned}$$

Since $c + A^T\lambda - \mu = 0$, $\mu = c + A^T\lambda$ so the above problem becomes

$$\begin{aligned} &\underset{\lambda}{\text{maximize}} && -b^T\lambda \\ &\text{subject to:} && -c - A^T\lambda \le 0 \end{aligned}$$

which is equivalent to

$$\begin{aligned} &\underset{\lambda}{\text{minimize}} && b^T\lambda \\ &\text{subject to:} && (-A^T)\lambda \le c \end{aligned}$$

Example B.2 Find the Lagrange dual of the QP problem

$$\begin{aligned} &\text{minimize} && \tfrac{1}{2}x^T Hx + p^T x \\ &\text{subject to:} && Ax \le b \end{aligned} \tag{B.37a-b}$$

where H is positive definite.

Solution The Lagrangian of the QP problem is given by

$$L(x,\mu) = \tfrac{1}{2}x^T Hx + p^T x + \mu^T(Ax - b)$$

Hence

$$q(\mu) = \inf_x \left\{\tfrac{1}{2}x^T Hx + p^T x + \mu^T(Ax - b)\right\} \tag{B.38}$$

where the infimum (defined as the largest lower bound of the objective function) is attained at $x = -H^{-1}(p + A^T\mu)$. By substituting this solution into (B.38), we obtain

$$q(\mu) = -\tfrac{1}{2}\mu^T AH^{-1}A^T\mu - \mu^T(AH^{-1}p + b) - \tfrac{1}{2}p^T H^{-1}p \tag{B.39}$$

If we let $P = AH^{-1}A^T$, $t = AH^{-1}p + b$ and neglect the constant term in (B.39), the dual function becomes $q(\mu) = -\tfrac{1}{2}\mu^T P\mu - \mu^T t$, hence the Lagrange dual of (B.37) is given by

$$\begin{aligned} &\text{minimize} && \tfrac{1}{2}\mu^T P\mu + \mu^T t \\ &\text{subject to:} && \mu \ge 0 \end{aligned} \tag{B.40}$$

Note that by definition matrix P in (B.40) is positive definite, therefore the dual problem is also a convex QP problem, but with simpler constraints in comparison with the primal problem in (1.89). In addition, if the number of constraints involved in the primal problem is smaller than n, so is the size of the dual problem. ■

E. CVX

• *The material of this section is largely based on the User Guide version 1.22 of the software written by M. Grant and S. Boyd.*

• **What is CVX?**

Designed by Michael Grant and Stephen Boyd, with input from Yinyu Ye, CVX is a

modeling system for disciplined convex programming that are convex optimization problems described by a limited set of construction rules. CVX solves standard problems such as LP, QP, SOCP, and SDP problems. Compared to directly using a solver such as SeDuMi for these problems, CVX greatly simplifies the task of specifying the problem. CVX also solves more complex convex optimization problems, including many involving nonsmooth functions, such as l_1 norm.

- **Examples**

Example B.3 We first consider the most basic convex optimization problem, least-squares. In a least-squares problem, we seek $\boldsymbol{x} \in R^n$ that minimizes $\| \boldsymbol{Ax} - \boldsymbol{b} \|_2$, where $\boldsymbol{A} \in R^{m \times n}$ is skinny and full rank (i.e., $m \geq n$ and rank($\boldsymbol{A}$) = n). Let us create some test problem data for m, n, $\boldsymbol{A}$, and $\boldsymbol{b}$ in MATLAB:

```
m = 16; n = 8;
A = randn(m,n);
b = randn(m,1);
```

(We chose small values of m and n to keep the output readable.) Then the least squares solution $\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$ is easily computed using the backslash operator:

```
x_ls = A\b;
```

Using cvx, the same problem can be solved as follows:

```
cvx_begin
   variable x(n);
   minimize( norm(A*x-b) );
cvx_end
```

When MATLAB reaches command cvx_end, the least-squares problem is solved, and the Matlab variable $\boldsymbol{x}$ is overwritten with the solution of the least-squares problem, i.e., $(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$. Now x is an ordinary length-n numerical vector, identical to what would be obtained in the traditional approach, at least to within the accuracy of the solver. In addition, two additional Matlab variables are created:

- cvx_optval, which contains the value of the objective function; i.e., $\| \boldsymbol{Ax} - \boldsymbol{b} \|_2$;
- cvx_status, which contains a string describing the status of the calculation. In this case, cvx_status would contain the string Solved. See Appendix C of the user guide for a list of the possible values of cvx_status and their meaning.
- cvx_slvtol: the tolerance level achieved by the solver.
- cvx_slvitr: the number of iterations taken by the solver.

Example B.4 Suppose we wish to add some simple upper and lower bounds to the least-squares problem above. That is, we wish to solve the CP problem

$$\begin{array}{ll} \text{minimize} & \| \boldsymbol{Ax} - \boldsymbol{b} \|_2 \\ \text{subject to:} & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \end{array}$$

where $\boldsymbol{l}$ and $\boldsymbol{u}$ are given data, vectors with the same dimension as the variable $\boldsymbol{x}$. We can no longer

use the simple backslash notation to solve this problem, but it can be transformed into a QP, which can be solved without difficulty if you have some form of QP software available.

Let us provide some numeric values for $\boldsymbol{l}$ and $\boldsymbol{u}$:

```
bnds = randn(n,2);
l = min(bnds,[],2);
u = max(bnds,[],2);
```

Then if you have the MATLAB Optimization Toolbox, you can use the quadprog function to solve the problem as follows:

```
x_qp = quadprog(2*A'*A,-2*A'*b,[],[],[],[],l,u );
```

This actually minimizes the square of the norm, which is the same as minimizing the norm itself. In contrast, the `cvx` specification is given by

```
cvx_begin
   variable x(n);
   minimize(norm(A*x-b));
   subject to
   l <= x <= u;
cvx_end
```

Example B.5 `cvx` supports constraints more complex than simple bounds as seen in Example B.4. For example, let us define new matrices $\boldsymbol{C}$ and $\boldsymbol{d}$ in MATLAB as follows:

```
p = 4;
C = randn(p,n);
d = randn(p,1);
```

Now let us add an equality constraint and a nonlinear inequality constraint to the original least-squares problem:

```
cvx_begin
   variable x(n);
   minimize(norm(A*x-b));
   subject to
   C*x == d;
   norm(x,Inf) <= 1;
cvx_end
```

Example B.6 For our final example in this section, let us show how traditional MATLAB code and cvx specifications can be mixed to form and solve multiple optimization problems. The following code solves the problem of minimizing $\|\boldsymbol{Ax}-\boldsymbol{b}\|_2+\gamma\|\boldsymbol{x}\|_1$, for a logarithmically spaced vector of (positive) values of γ. This gives us points on the optimal trade-off curve between $\|\boldsymbol{Ax}-\boldsymbol{b}\|_2$ and $\|\boldsymbol{x}\|_1$. An example of this curve is given in Fig. B.4.

```
gamma = logspace(-2,2,20);
l2norm = zeros(size(gamma));
l1norm = zeros(size(gamma));
```

```
fprintf(1, ' gamma norm(x,1) norm(A*x-b)\n');
fprintf(1, '-------------------------------------\n');
for k = 1:length(gamma),
   fprintf(1,'%8.4e', gamma(k));
   cvx_begin quiet
      variable x(n);
      minimize(norm(A*x-b)+gamma(k)*norm(x,1));
   cvx_end
   l1norm(k) = norm(x,1);
   l2norm(k) = norm(A*x-b);
   fprintf(1,' %8.4e %8.4e\n',l1norm(k),l2norm(k));
end
plot(l1norm, l2norm);
xlabel('norm (x,1)');
ylabel('norm (A*x-b)');
grid
```

- **Basic rules**

(1) All `cvx` models must be preceded by command `cvx_begin` and terminated with commend `cvx_end`.

Command `cvx_begin` accepts several variants. For example, `cvx_begin quiet` prevents the model from producing screen output while it is been solved.

(2) Data Types for Variables

All variables must be declared using command `variable` (or `variables`). Variables can be real or complex; and scalar, vector, matrix, or n-dimensional array. Variables can be matrices with structure,

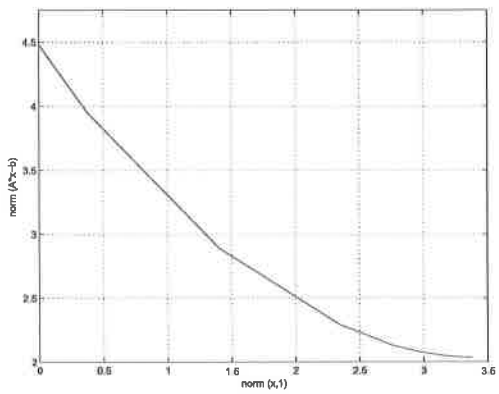


Fig. B.4

like symmetry or bandedness. The variable structure is given by including descriptive keywords following the name and size of the variable. Here are some examples of `cvx` variable:

```
variable w(50) complex;
variable X(20,10);
variable Y(50,50) symmetric;
variable Z(100,100) hermitian toeplitz;
```

- **Structure Keywords**

Version 1.22 of `cvx` supports the following structure keywords:

`banded(lb,ub)` `complex` `diagonal` `hankel` `hermitian` `lower_bidiagonal` `lower_hessenberg` `lower_triangular` `scaled_identity` `skew_symmetric` `symmetric` `toeplitz` `tridiagonal` `upper_bidiagonal` `upper_hankel` `upper_hessenberg` `upper_triangular`

(3) Objective functions

Declaring an objective function requires the use of the `minimize` or `maximize` function, as appropriate. The objective function in a call to minimize must be convex; the objective function in a call to maximize must be concave. At most one objective function may be declared in a given `cvx` specification, and the objective function must have a scalar value.

If no objective function is specified, the problem is interpreted as a *feasibility* problem, which is the same as performing a minimization with the objective function set to zero. In this case, `cvx_optval` is either `0`, if a feasible point is found, or `+Inf`, if the constraints are not feasible.

(4) Constraints

Version 1.22 of `cvx` supports the following constraint types:

- Equality == constraints, where both the left- and right-hand sides are affine functions of the optimization variables.
- Less-than <= inequality constraints, where the left-hand expression is convex, and the right-hand expression is concave.
- Greater-than >= constraints, where the left-hand expression is concave, and the right-hand expression is convex.

These equality and inequality operators work for arrays. When both sides of the constraint are arrays of the same size, the constraint is imposed elementwise. For example, if `a` and `b` are $m \times n$ matrices, then `a<=b` is interpreted by `cvx` as mn (scalar) inequalities, i.e., each entry of a must be less than or equal to the corresponding entry of `b`. `cvx` also handles cases where one side is a scalar and the other is an array. This is interpreted as a constraint for each element of the array, with the (same) scalar appearing on the other side. As an example, if `a` is an $m \times n$ matrix, then `a>=0` is interpreted as mn inequalities: each element of the matrix must be nonnegative.

(5) Functions

The base `cvx` function library includes a variety of convex, concave, and affine functions which accept `cvx` variables or expressions as arguments. Many are common MATLAB functions such as `sum`, `trace`, `diag`, `sqrt`, `max`, and `min`, re-implemented as needed to support `cvx`; others are new functions not found in MATLAB. A complete list of the functions in the base library can be found in §B of the user guide. It's also possible to add your own new functions; see §5 of the user guide.

(6) Sets

◊ `cvx` supports the definition and use of convex sets. The base library includes the cone of positive semidefinite $n \times n$ matrices, the second-order or Lorentz cone, and various norm balls. A complete list of sets supplied in the base library is given in §B of the user guide.

◊ Since MATLAB does not have a set membership operator, `cvx` adopts a slightly different syntax to require that an expression is in a set. To represent a set, a function is used to return an unnamed variable that is required to be in the set. Consider, for example, $\boldsymbol{S}_+^n$, the cone of symmetric positive semidefinite $n \times n$ matrices. In `cvx`, this is represented by function `semidefinite(n)`, which returns an unnamed new variable, that is constrained to be positive semidefinite. To require that the matrix expression `X` be symmetric positive semidefinite, the syntax `X == semidefinite(n)` is used. The literal meaning of this is that `X` is constrained to be equal to some unnamed variable, which is required to be an $n \times n$ symmetric positive semidefinite matrix. This is equivalent to saying that `X` must be symmetric positive semidefinite.

As an example, consider the constraint that a (matrix) variable `X` is a correlation matrix, i.e., it is symmetric, has unit diagonal elements, and is positive semidefinite.

◊ In `cvx` we can declare such a variable and impose such constraints using

```
variable X(n,n) symmetric;
X == semidefinite(n);
diag(X) == ones(n,1);
```

The second line here imposes the constraint that `X` be positive semidefinite (you can read `'=='` here as `'is'`, so the second line can be read as `'X is positive semidefinite'`). The left-hand side of the third line is a vector containing the diagonal elements of `X`, whose elements we require to be equal to one. Incidentally, `cvx` allows us to simplify the third line to

```
diag(X) == 1;
```

because `cvx` follows the MATLAB convention of handling array/scalar comparisons by comparing each element of the array independently with the scalar.

◊ Sets can be combined in affine expressions, and we can constrain an affine expression to be in a convex set. For example, we can impose constraints of the form

```
A*X*A'-X == B*semidefinite(n)*B';
```

where `X` is an $n \times n$ symmetric variable matrix, and $\boldsymbol{A}$ and $\boldsymbol{B}$ are $n \times n$ constant matrices. This constraint requires that $\boldsymbol{AXA}^T - \boldsymbol{X} = \boldsymbol{BYB}^T$, for some $Y \in \boldsymbol{S}_+^n$.

Example B.7 Standard-form LP

```
function x = lps_cvx(c,A,b)
n = length(c);
cvx_begin quiet
  variable x(n);
  minimize(c'*x);
  subject to
  A*x == b;
  x >= 0;
cvx_end
```

Example B.8 Alternative-form LP

```
function x = lpa_cvx(c,A,b)
```

```
n = length(c);
cvx_begin quiet
  variable x(n);
  minimize(c'*x);
  subject to
  A*x >= b;
cvx_end
```

Example B.9 Convex QP

```
function x = qp_cvx(H,p,A,b,E,d)
n = length(p);
cvx_begin quiet
  variable x(n);
  minimize(0.5*x'*H*x+x'*p);
  subject to
  A*x == b;
  E*x >= d;
cvx_end
```

Example B.10 SDP

```
% Program: sdp_cvx.m
% To solve semidefinite programming (SDP) problem
%           minimize    c'*x
%           subject to: F0 + x(1)*F1 + ... + x(p)*Fp >= 0
% using cvx, where ">= 0" means being positive semidefinite,
% and x = [x1 x2 ... xp]'.
% Input:
% c: constant vector of length p defining objective function.
% F = [F0 F1 ... Fp] with each Fi a symmetric square matrix.
% Output:
% x: solution of the SDP problem.
% Written by W.-S. Lu, University of Victoria.
% Last modified: Oct. 29, 2012.
% Example: Solve the SDP problem involved in Example 14.1 from PO, where
% c = [0 0 0 -1]';
% F = [F0 F1 F2 F3 F4]
%  = [-2.0   0.5   0.6   0  -1  0  0 0 -1 0  0  0 -1  0  0;
%     0.5  -2.0  -0.4  -1   0  0  0 0  0 0  0 -1  0 -1  0;
%     0.6  -0.4  -3.0   0   0  0 -1 0  0 0 -1  0  0  0 -1];
% x = sdp_cvx(c,F);
function x = sdp_cvx(c,F)
p = length(c);
[n,m] = size(F);
Fx = F(:,1:n);
cvx_begin quiet
  variable x(p);
```

```
  minimize(c'*x);
  subject to
  for i = 1:p,
     Fx = Fx + x(i)*F(:,(i*n+1):((i+1)*n));
  end
  Fx == semidefinite(n);
cvx_end
```

Example B.11 SOCP

```
% Program: socp_cvx.m
% To solve second-order cone programming (SOCP) problem
%        minimize b'*x
%        subject to: ||Ai'*x + ci|| <= bi'*x + di
%                for i = 1, ..., q
% using cvx, where x = [x1 x2 ... xm]' and Ai is a matrix of size m by ni.
Thus there are a total of q
% 2nd-order (Lorentz) cones, each with size ni.
% Input:
% A = [A1 A2 ... Aq]
% B = [b1 b2 ... bq]
% b: constant vector of length m defining objective function
% c = [c1; c2; ...; cq]
% d = [d1 d2 ... dq]
% k = [n1 n2 ... nq]
% Output:
% x: solution of the SOCP problem.
% Written by W.-S. Lu, University of Victoria.
% Last modified: Oct. 30, 2012.
% Example: Solve the SOCP problem in Example 14.5 from PO, where
% b = [1 0 0 0 0]';
% A = [0  0  0    0  0  0;
%    -1  0  0.5   0  0  0;
%     0  1  0    1  0  0;
%     1  0  0    0 -0.7071 -0.3536;
%     0  -1  0    0 -0.7071  0.3536];
% z = zeros(5,1);B = [b z z];
% c = [0 0 -0.5 0 4.2426 -0.7071]';
% d = [0 1 1];
% k = [2 2 2];
% x = socp_cvx(A,B,b,c,d,k);
function x = socp_cvx(A,B,b,c,d,k)
m = length(b);
q = length(d);
t = 0;
cvx_begin quiet
  variable x(m);
```

```
  minimize(b'*x);
  subject to
  for i = 1:q,
     Ai = A(:,(t+1):(t+k(i)));
     bi = B(:,i);
     ci = c((t+1):(t+k(i)));
     di = d(i);
     norm(Ai'*x+ci) <= bi'*x + di;
     t = t + k(i);
  end
cvx_end
```

Appendix C: A proof of the Hoeffding Inequality

The proof presented below follows R. Nowak's 2007 course notes on statistical learning theory. It consists of three steps.

Step 1 *Markov inequality*

Let z be a nonnegative random variable and t be a positive constant. Then

$$P(z \geq t) \leq \frac{E(z)}{t} \tag{C.1}$$

Proof: Let p be the density function of random variable z, by definition we can write

$$E(z) = \int_0^\infty yp(y)dy \geq \int_t^\infty yp(y)dy \geq t\int_t^\infty p(y)dy = tP(z \geq t)$$

which implies (C.1). ■

Now let u be a random variable with mean μ and variance σ^2, then $(u-\mu)^2$ is a nonnegative random variable whose mean is equal to σ^2. By applying (C.1) with $z = (u-\mu)^2$, we obtain

$$P[(u-\mu)^2 \geq t] \leq \frac{\sigma^2}{t} \tag{C.2}$$

Furthermore, let $u_1, u_2, \ldots, u_N$ be i.i.d. random variables, each with mean μ and variance σ^2, and let $u = \frac{1}{N}\sum_{n=1}^{n} u_n$. Obviously, $E(u) = \mu$ and $E[(u-\mu)^2] = \frac{\sigma^2}{N}$, hence (C.2) implies that

$$P[(u-\mu)^2 \geq t] \leq \frac{\sigma^2}{Nt} \tag{C.3}$$

As yet another application of the Markov inequality, let z be a nonnegative random variable and t and s be positive constants. Since $z \geq t$ if and only if $e^{sz} \geq e^{st}$, applying (C.1) to random variable e^{sz} yields

$$P(z \geq t) = P\left(e^{sz} \geq e^{st}\right) \leq e^{-st}E(e^{sz}) \tag{C.4}$$

Step 2 *Chernoff bound*

Now consider random variable $\sum_{n=1}^{N}\left(e(h(\boldsymbol{x}_n), y_n) - E_{\text{out}}(h)\right)$ and apply (C.4), we obtain

$$P\left(\sum_{n=1}^{N}\left(e(h(\boldsymbol{x}_n),y_n)-E_{\text{out}}(h)\right)\geq t\right)\leq e^{-st}E\left[e^{s\sum_{n=1}^{N}\left(e(h(\boldsymbol{x}_n),y_n)-E_{\text{out}}(h)\right)}\right]$$
$$=e^{-st}\prod_{n=1}^{N}E\left[e^{s\left(e(h(\boldsymbol{x}_n),y_n)-E_{\text{out}}(h)\right)}\right] \tag{C.5}$$

The last equality is due to the fact that in statistical learning it is often assumed that the training data $\mathcal{D}=\{(\boldsymbol{x}_n,y_n),n=1,2,...,N\}$ are i.i.d., hence $e(h(\boldsymbol{x}_n),y_n)$ are i.i.d. random variables.

Step 3 *Hoeffding inequality*

A general form of Hoeffding's inequality can be stated as follows: Let u_1, u_2, ..., u_N be independent and bounded random variables that satisfy $u_n\in[a_n,b_n]$ with probability 1. Let $\hat{z}=\sum_{n=1}^{N}u_n$. Then for any $t>0$, we have

$$P\left(|\hat{z}-E[\hat{z}]|\geq t\right)\leq 2e^{\frac{-2t^2}{\sum_{i=1}^{n}(b_i-a_i)^2}} \tag{C.6}$$

Proof: A critical component of the proof is to show that for a random variable z with $E[z]=0$ and $a\leq z\leq b$, $E\left[e^{sz}\right]$ has an upper bound

$$E\left[e^{sz}\right]\leq e^{\frac{s^2(b-a)^2}{8}} \tag{C.7}$$

To show (C.7), we use convexity of e^{sz} to write

$$e^{sz}\leq\frac{z-a}{b-a}e^{sb}+\frac{b-z}{b-a}e^{sa}\quad\text{for } a\leq z\leq b$$

It follows that

$$\begin{aligned}E\left[e^{sz}\right]&\leq E\left[\frac{z-a}{b-a}\right]e^{sb}+E\left[\frac{b-z}{b-a}\right]e^{sa}\\&=\frac{b}{b-a}e^{sa}-\frac{a}{b-a}e^{sb}\\&=\left(1-\theta+\theta e^{s(b-a)}\right)e^{-\theta s(b-a)}\quad\text{with }\theta=-\frac{a}{b-a}\end{aligned} \tag{C.8}$$

By defining $\phi(v)=-\theta v+\ln(1-\theta+\theta e^{v})$ with $v=s(b-a)$, (C.8) becomes

$$E\left[e^{sz}\right]\leq e^{\phi(v)} \tag{C.9}$$

Using calculus it can be shown that

$$\phi(v) \le \frac{s^2(b-a)^2}{8}$$

which in conjunction with (C.8) yields (C.7). It follows that

$$P\left(\hat{z} - E[\hat{z}] \ge t\right) \le e^{-st} \prod_{n=1}^{N} E\left[e^{s(u_n - E(u_n))}\right] \le e^{-st} e^{s^2 \sum_{n=1}^{N} \frac{(b_n - a_n)^2}{8}}$$

By choosing

$$s = \frac{4t}{\sum_{n=1}^{N} (b_n - a_n)^2}$$

we obtain

$$P\left(\hat{z} - E[\hat{z}] \ge t\right) \le e^{\frac{-2t^2}{\sum_{i=1}^{n} (b_i - a_i)^2}}$$

Similarly, we can show

$$P\left(E[\hat{z}] - \hat{z} \ge t\right) \le e^{\frac{-2t^2}{\sum_{i=1}^{n} (b_i - a_i)^2}}$$

combining the last two inequalities completes the proof. By applying the Hoeffding inequality with

$$u_n = e(h(\mathbf{x}_n), y_n) - E_{\text{out}}(h)$$

we obtain

$$P\left(|E_{\text{in}}(h) - E_{\text{out}}(h)| \ge \varepsilon\right) = P\left(\tfrac{1}{N} |\hat{z} - E(\hat{z})| \ge \varepsilon\right) \le 2e^{-2\varepsilon^2 N}$$

which is (1.21). ■

Problems

A.1 [7] Suppose we have three colored boxes r (red), b (blue), and g (green). Box r contains 3 apples, 4 oranges, and 3 limes; box b contains 1 apple, 1 orange, and 0 limes; and box g contains 3 apples, 3 oranges, and 4 limes. If a box is chosen at random with probabilities $p(r) = 0.2$, $p(b) = 0.2$, and $p(g) = 0.6$, and a piece of fruit is removed from the box (with equal probability of selecting any of the items in the box), then

(i) what is the probability of selecting an apple?

(ii) If we observe that the selected fruit is an orange, what is the probability that it came from the green box?

A.2 [7] There are 2 opaque bags. One bag has two blue balls and the other has one blue ball and one red ball. Suppose you pick a bag at random, and then pick a ball from that bag at random. You found the ball you picked was a blue ball. Now you pick the second ball from that same bag. Compute the probability that this ball is also blue?

A.3 Suppose we have a box that contains A red balls and B blue balls, one takes balls out of the box one by one, what is the probability that the balls that remain in the box are all red balls?

A.4 Prove the equality in Eq. (A.27).

A.5 Let x be a random variable with mean (expectation) $E[x] = \mu$ and variance $\mathrm{Var}[x] = \sigma^2$.

Compute and express the mean and variance of random variable $ax + b$ in terms of μ and σ^2, where a and b are constants.

A.6 Let x denote a set of temperature readings in Celsius (C). Treating x as a random variable, its average (expectation) and variance are known to be 22.5° C and $\sigma^2 = 0.4$. If we convert x to a set of temperature readings y in Fahrenheit (F) using $F = (9/5)C + 32$, what are the average and variance of the temperature data in Fahrenheit?

A.6 Let ξ_1 and ξ_2 be two independent random variables, find the distribution function of random variable $\xi = \xi_1 + \xi_2$ where

(i) ξ_1 and ξ_2 are uniform distribution over intervals (–5, 1) and (1, 5), respectively.

(ii) ξ_1 and ξ_2 are random variables with density functions $\varphi_1(x) = \varphi_2(x) = \dfrac{1}{2\alpha} e^{-|x|/\alpha}$ with some $\alpha > 0$.

B.1 Consider the convex QP problem

$$\underset{x}{\text{minimize}} \quad \frac{1}{2} \boldsymbol{x}^T \begin{bmatrix} 10 & -9 \\ -9 & 8.15 \end{bmatrix} \boldsymbol{x} + \boldsymbol{x}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{P1.1}$$

(i) Apply SDM to solve the problem with initial point $\boldsymbol{x}_0 = [1\ 1]^T$. Exact line search is carried out using the closed-form formula

$$\alpha_k = \frac{g_k^T g_k}{g_k^T H g_k}$$

where $\boldsymbol{H}$ is the Hessian of the objective function and $\boldsymbol{g}_k$ is the gradient of the objective function at $\boldsymbol{x}_k$. How many iterations the SDM has to take to achieve a solution $\boldsymbol{x}_s$ with the accuracy of $\|\boldsymbol{x}_s - \boldsymbol{x}^*\|_2 \le 10^{-5}$ where $\boldsymbol{x}^*$ is the true minimizer of the problem which is known to be $\boldsymbol{x}^* = [-16.3 \quad -18]^T$?

(ii) Now solve the problem in part (i) by scaling the variable $\boldsymbol{x}$ with $\boldsymbol{x} = \boldsymbol{T}\boldsymbol{y}$ first, where matrix $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \begin{bmatrix} 0.25 & 4.5 \\ 0 & 5 \end{bmatrix}$$

Working with the scaled QP problem, how many SDM iterations are needed to achieve a solution with the same accuracy as in part (i)?

Explain why such a scaling matrix can improve SDM's performance? If this $\boldsymbol{T}$ was not given, how do you find a similar $\boldsymbol{T}$ to properly scale an ill-conditioned QP problem?

To earn credit, include the solution details, your MATLAB code, and all numerical results.

B.2 Consider the one-variable *polynomial curve fitting* problem where a polynomial model of the form

$$y(x, \boldsymbol{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m = \sum_{i=0}^{m} w_i x^i \qquad \text{(P1.1)}$$

is used to fit a data set $\{(x_k, y_k), \text{for } k = 1, \ldots, n\}$, where $\boldsymbol{w} = [w_0 \quad w_1 \quad \cdots \quad w_m]^T$. The function $y(x, \boldsymbol{w})$ is a polynomial of order m in x, whose behavior for a fixed order m is determined by its coefficients $\{w_i, i = 0, 1, \ldots, m\}$. Therefore, the *polynomial curve fitting* problem is about to find "optimal" $\{w_i, i = 0, 1, \ldots, m\}$ such that the polynomial $y(x, \boldsymbol{w})$, when evaluated at $\{x_k, k = 1, 2, \ldots, n\}$, produces $\{y(x_k, \boldsymbol{w}), k = 1, \ldots, n\}$ that is the *closest* to the given data $\{y_k, k = 1, 2, \ldots, n\}$ in a certain sense. The *least-squares* (LS) polynomial curve fitting is one of the most popular approach to deal with polynomial modeling problems, which in the present case amounts to finding an $(m + 1)$-dimensional $\boldsymbol{w}$ that solves the unconstrained problem

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad e(\boldsymbol{w}) = \frac{1}{2} \sum_{k=1}^{n} |y(x_k, \boldsymbol{w}) - y_k|^2 \qquad \text{(P1.2)}$$

where $y(x, \boldsymbol{w})$ is given by (P1.1). Show that the optimal $\boldsymbol{w}$ satisfies the linear system of equations

$\boldsymbol{A}\boldsymbol{w} = \boldsymbol{t}$ where, for real-valued data set $\{(x_k, y_k), \text{for } k = 1, \ldots, n\}$, $\boldsymbol{A} = (a_{i,j}) \in R^{(m+1)\times(m+1)}$ with i, $j = 0, 1, \ldots, m$; and $\boldsymbol{t} = (t_i) \in R^{(m+1)\times 1}$ with $i = 0, 1, \ldots, m$ are determined by the data set as

$$a_{i,j} = \sum_{k=1}^{n} (x_k)^{i+j} \quad \text{and} \quad t_i = \sum_{k=1}^{n} (x_k)^i y_k \tag{P1.3}$$

To get credit, include all derivation details.

B.3 The purpose of this problem is to perform numerical evaluation of the LS data fitting method.

(i) Data preparation: the data to be used in what follows was generated from a 2nd order polynomial model that were contaminated by a small amount of Gaussian noise. Use the MATLAB code below to produce the data.

```
w_target = [9.5   -5.6   1];   % here the vector is associated with a poly with increasing order.
p_target = fliplr(w_target);   % function polyval requires poly coefficients with decreasing order.
x = [0.5   2.5   3   3.95   4.05];
randn('state',16)
y = polyval(p_target,x) + 0.4*randn(1,5);
```

To see how the noisy data are related to the target polynomial, use MATLAB to plot the following:

```
t = 0:4.5/999:4.5;
z = polyval(p_target,t);
figure(1)
plot(t,z,'k')
hold on
plot(x,y,'ro','linewidth',1.5)
```

(ii) Apply the polynomial data fitting method from Prob. B.2 to the data set {x, y} generated in part (i) with a 2nd – order polynomial $y(x, \boldsymbol{w})$ (i.e. $m = 2$, see Eq. (P1.1)). To earn credit:

- Compute and report the numerical values of $\boldsymbol{w}$ for the $y(x, \boldsymbol{w})$ that solves the minimization problem in (P1.2);
- Plot $y(t, \boldsymbol{w})$ versus t = 0:4.5/999:4.5 in Figure 1 generated in part (i) with the data set also displayed in the figure;
- Evaluate the approximation error, namely the value of the minimized objective function $e(\boldsymbol{w})$ (see (P1.2)) and comment the performance of the 2nd-order model obtained.

B.4 Repeat Problem B.3 with a 4th-order polynomial $y(x, \boldsymbol{w})$.

B.5 Use CVX to solve the following LP and convex QP problems

(i)

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) = 2x_1 + 9x_2 + 3x_3 \\ \text{subject to:} \quad & -2x_1 + 2x_2 + x_3 - x_4 = 1 \\ & x_1 + 4x_2 - x_3 - x_5 = 1 \\ & x_i \geq 0 \quad \text{for } i = 1, 2, \ldots, 5 \end{aligned}$$

(ii)

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) = x_1 + x_2 \\ \text{subject to:} \quad & x_1 \leq 2 \\ & x_2 \leq 2 \\ & -x_2 \leq 5 \\ & -2x_1 + x_2 \leq 2 \\ & 2x_1 + x_2 \leq 4 \end{aligned}$$

(iii)

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) = (x_1 - x_3)^2 + (x_2 - x_4)^2 \\ \text{subject to:} \quad & -x_1 \leq 0 \\ & -x_2 \leq 0 \\ & x_1 + 2x_2 \leq 2 \\ & -x_4 \leq -2 \\ & -x_3 - x_4 \leq -3 \\ & x_3 + 2x_4 \leq 6 \end{aligned}$$

To earn credit, include CVX code and report numerical values of the solutions with an accuracy of no-less-than-eight decimal places.